



Greenplum

The Data Computing Division of EMC

EMC²

where information lives[®]

Greenplum Database
Administrator Guide
Version 3.3.7

Greenplum Database Administrator Guide 3.3.7 - Contents

Preface	1
About This Guide.....	1
Greenplum Database Documentation	2
Document Conventions.....	2
Contact Us	4

Section I: Introduction to Greenplum

Chapter 1: About the Greenplum Architecture	6
About the Greenplum Master.....	7
About the Greenplum Segments.....	7
About the Greenplum Interconnect	7
About Redundancy and Failover in Greenplum Database	8
About Segment Mirroring.....	8
About Master Mirroring	9
About Interconnect Redundancy	9
About Parallel Data Loading	10
About Management and Monitoring	10
Chapter 2: About Distributed Databases	12
Understanding How Data is Stored.....	12
Understanding Greenplum Distribution Policies.....	13
Chapter 3: Summary of Greenplum Features	14
System Requirements	14
Feature Summary	15
Greenplum SQL Standard Conformance.....	15
Greenplum and PostgreSQL Compatibility	19
Chapter 4: About Greenplum Query Processing	26
Understanding Greenplum Query Plans	26
Understanding Parallel Query Execution	28

Section II: Installation and Initialization

Chapter 5: Preparing for a Greenplum Installation	30
About Greenplum and Hardware Platforms.....	30
Example Segment Host Hardware Stack	30
Disk Layout	31
Network Layout	32
Required OS System Settings for Greenplum	34
Linux Parameter Settings.....	35
Solaris 10 x86 Parameter Settings.....	35
Verifying OS Settings.....	35
Estimating Storage Capacity	36
Calculating Usable Disk Capacity	36
Calculating User Data Size.....	37
Calculating Space Requirements for Metadata and Logs	37
Validating Hardware Performance	37
Validating Disk I/O and Memory Bandwidth	38
Validating Network Performance	39

Chapter 6: Installing the Greenplum Software	40
Overview	40
Installing Greenplum Database on the Master Host	40
Running the Installer	41
Configuring Your Installation on the Master Host.....	42
Setting OS Tuning Parameters.....	42
Designating a Greenplum User	42
Designating a Greenplum Group (optional)	42
Changing Ownership of Your Greenplum Installation	43
Configuring Your Environment Variables	43
Creating the Data Directory on the Master Host.....	43
Installing Greenplum Database on the Segment Hosts	44
Setting OS Tuning Parameters.....	44
Setting Up a Trusted Host Environment	44
Copying the Greenplum Software to the Segment Hosts.....	46
Creating the Data Storage Areas on the Segment Hosts	47
Synchronizing System Clocks	48
Next Steps	48
Uninstalling Greenplum Database.....	48
Chapter 7: Configuring Localization Settings	50
About Locale Support in Greenplum Database.....	50
Locale Behavior	51
Troubleshooting Locales.....	52
Character Set Support.....	52
Setting the Character Set	54
Character Set Conversion Between Server and Client	54
Chapter 8: Initializing a Greenplum Database System	57
Overview.....	57
Initializing Greenplum Database.....	57
Creating a Host List File	58
Creating the Greenplum Database Configuration File	59
Running the Initialization Utility	59
Setting the Master Data Directory Environment Variable	61
Next Steps	61
Allowing Client Connections	61
Creating Databases and Loading Data	61
Chapter 9: Greenplum Database Demo Programs	62
Greenplum Database Demo (gpdemo)	62
Before You Begin	62
Running Greenplum Database Demo (gpdemo)	62
Running MIVP	63
About TPCB	64
Removing Greenplum Database Demo (gpdemo).....	65
 Section III: Access Control and Security	
Chapter 10: Managing Roles and Privileges	67
Security Best Practices for Roles and Privileges.....	67
Creating New Roles (Users).....	68

Altering Role Attributes.....	68
Creating Groups (Role Membership).....	69
Managing Object Privileges	70
Simulating Row and Column Level Access Control	71
Encrypting Data	71
Chapter 11: Configuring Client Authentication	73
Allowing Connections to Greenplum Database.....	73
Editing the pg_hba.conf File.....	74
Limiting Concurrent Connections.....	75
Encrypting Client/Server Connections	76
Chapter 12: Accessing the Database	77
Establishing a Database Session	77
Supported Client Applications.....	78
Greenplum Database Client Applications.....	78
pgAdmin III for Greenplum Database	79
Database Application Interfaces.....	82
Third-Party Client Tools	83
Troubleshooting Connection Problems	84
Chapter 13: Managing Workload and Resources	85
Overview of Greenplum Workload Management	85
How Resource Queues Work in Greenplum Database.....	85
Steps to Enable Workload Management	86
Configuring Workload Management.....	87
Creating Resource Queues	87
Creating Queues with an Active Threshold	88
Creating Queues with a Cost Threshold	88
Assigning Roles (Users) to a Resource Queue.....	89
Removing a Role from a Resource Queue	89
Modifying Resource Queues.....	89
Altering a Resource Queue.....	89
Dropping a Resource Queue	90
Checking Resource Queue Status	90
Viewing Resource Queue Status.....	90
Viewing Resource Queue Statistics	90
Viewing the Roles Assigned to a Resource Queue	91
Viewing the Waiting Queries for a Resource Queue.....	91
Clearing a Waiting Statement From a Resource Queue	92

Section IV: Database Administration

Chapter 14: Defining Database Objects	94
Creating and Managing Databases	94
About Template Databases	94
Creating a Database	94
Viewing the List of Databases	95
Altering a Database	95
Dropping a Database	95
Creating and Managing Schemas.....	96
The Default 'Public' Schema.....	96

Creating a Schema	96
Schema Search Paths	96
Dropping a Schema	97
System Schemas	97
Creating and Managing Tables	98
Creating a Table	98
Altering a Table	105
Dropping a Table	106
Partitioning Large Tables	107
Understanding Table Partitioning in Greenplum Database	107
Deciding on a Table Partitioning Strategy	108
Creating Partitioned Tables	109
Loading Partitioned Tables	112
Verifying Your Partition Strategy	113
Viewing Your Partition Design	114
Maintaining Partitioned Tables	114
Creating and Using Sequences	118
Creating a Sequence	118
Using a Sequence	118
Altering a Sequence	119
Dropping a Sequence	119
Using Indexes in Greenplum Database	119
Index Types	121
Creating an Index	122
Examining Index Usage	122
Managing Indexes	123
Dropping an Index	123
Creating and Managing Views	124
Creating Views	124
Dropping Views	124
Chapter 15: Managing Data	125
About Concurrency Control in Greenplum Database	125
Inserting New Rows	126
Updating Existing Rows	127
Deleting Rows	127
Truncating a Table	128
Working With Transactions	128
Transaction Isolation Levels	128
Vacuuming the Database	130
Configuring the Free Space Map	130
Chapter 16: Querying Data	131
Defining Queries	131
SQL Lexicon	131
SQL Value Expressions	131
Using Functions and Operators	140
Using Functions in Greenplum Database	140
User-Defined Functions	141
Built-in Functions and Operators	141

Query Profiling	148
Reading EXPLAIN Output	149
Reading EXPLAIN ANALYZE Output	150
What to Look for in a Query Plan	151
Chapter 17: Parallel Data Loading	153
About External Tables and Web Tables	153
External Tables and Query Planner Statistics	153
Creating and Using External Tables	154
Formatting of Input Data	155
Using the Greenplum File Server (gpfdist)	156
Defining an External Table	159
Loading Data From an External Table	160
Investigating Load Errors.....	160
Creating and Using Web Tables	161
Defining Command-Based Web Tables.....	162
Defining URL-Based Web Tables	163
Loading Data from Web Tables	163
Non-Parallel Data Loading	163
Loading Data with COPY	163
Loading Data with INSERT	164
Other Data Loading Performance Tips	164

Section V: System Administration

Chapter 18: Starting and Stopping Greenplum	167
Overview.....	167
Starting Greenplum Database	167
Restarting Greenplum Database	167
Uploading Configuration File Changes Only	168
Starting the Master in Maintenance Mode	168
Stopping Greenplum Database	168
Chapter 19: Configuring Your Greenplum System	170
About Greenplum Master, Global, and Local Parameters	170
Setting Configuration Parameters.....	170
Setting a Local Configuration Parameter	171
Setting a Global or Master Configuration Parameter	171
Configuration Parameter Categories	172
File Location Parameters.....	173
Connection and Authentication Parameters.....	173
System Resource Consumption Parameters	174
Write Ahead Log Parameters	175
Query Tuning Parameters	176
Error Reporting and Logging Parameters	178
Runtime Statistics Collection Parameters	178
Automatic Statistics Collection Parameters	179
Automatic Vacuuming Parameters	179
Client Connection Default Parameters.....	180
Lock Management Parameters	180
Workload Management Parameters	180

External Table Parameters	180
Append-Only Table Parameters.....	181
Past PostgreSQL Version Compatibility Parameters.....	181
Greenplum Array Configuration Parameters.....	181
Chapter 20: Enabling High Availability Features	183
Overview of High Availability in Greenplum Database.....	183
Overview of Segment Mirroring	183
Overview of Master Mirroring.....	184
Overview of Fault Detection and Recovery	185
Enabling Mirroring in Greenplum Database.....	186
Enabling Segment Mirroring.....	186
Enabling Master Mirroring	187
Setting the Fault Operational Mode	188
Knowing When a Segment is Down	189
Checking for Failed Segments.....	189
Checking the Log Files	190
Recovering a Failed Segment	190
Recovering From Segment Failures.....	191
Recovering a Failed Master.....	193
Chapter 21: Backing Up and Restoring Databases.....	195
Overview of Backup and Restore Operations	195
About Parallel Backups	195
About Non-Parallel Backups.....	196
About Parallel Restores.....	196
About Non-Parallel Restores	197
Backing Up a Database	197
Backing Up a Database with gp_dump.....	198
Automating Parallel Backups with gpccrondump.....	199
Restoring From Parallel Backup Files	200
Restoring a Database with gp_restore	200
Restoring a Database Using gpdbrestore	201
Restoring to a Different Greenplum System Configuration	202
Rebuilding a New Greenplum System From Backup	203
Chapter 22: Expanding a Greenplum System	205
Planning Greenplum System Expansion.....	205
System Expansion Overview	205
System Expansion Checklist	206
Planning New Hardware Platforms	207
Planning Initialization of New Segments	208
Planning Table Redistribution.....	209
Preparing and Adding Nodes	211
Adding New Nodes to the Trusted Host Environment	211
Verifying OS Settings.....	213
Validating Disk I/O and Memory Bandwidth	213
Integrating New Hardware into the System	214
Initializing New Segments.....	214
Creating an Input File for System Expansion	214
Running gpexpand to Initialize New Segments	217

Rolling Back an Expansion Setup	218
Redistributing Tables.....	218
Ranking Tables for Redistribution	218
Redistributing Tables Using gpexpand.....	219
Monitoring Table Redistribution.....	219
Removing the Expansion Schema.....	220
Chapter 23: Monitoring a Greenplum System.....	221
Monitoring Database Activity and Performance.....	221
Monitoring System State	221
Checking System Status and Configuration.....	221
Checking Disk Space Usage	222
Checking for Data Distribution Skew.....	223
Viewing the Database Server Log Files	224
Log File Format.....	224
Searching the Greenplum Database Server Log Files	225
Using the Jetpack Administrative Interface.....	226
Chapter 24: Routine System Maintenance Tasks.....	227
Routine Vacuum and Analyze	227
Transaction ID Management	227
System Catalog Maintenance	227
Vacuum and Analyze for Query Optimization	228
Routine Reindexing	228
Managing Greenplum Database Log Files	229
Database Server Log Files	229
Management Utility Log Files	229

Section VI: Performance Tuning

Chapter 25: Defining Database Performance	231
Understanding the Performance Factors	231
System Resources	231
Workload	231
Throughput.....	231
Contention.....	232
Optimization	232
Determining Acceptable Performance	232
Baseline Hardware Performance	232
Performance Benchmarks	232
Chapter 26: Common Causes of Performance Issues.....	234
Identifying Hardware and Segment Failures	234
Managing Workload.....	235
Avoiding Contention	235
Maintaining Database Statistics.....	235
Identifying Statistics Problems in Query Plans	235
Tuning Statistics Collection	236
Optimizing Data Distribution	236
Optimizing Your Database Design.....	236
Greenplum Database Maximum Limits.....	237

Chapter 27: Investigating a Performance Problem	238
Checking System State	238
Checking Database Activity	238
Checking for Active Sessions (Workload)	238
Checking for Locks (Contention)	238
Checking Query Status and System Utilization.....	239
Troubleshooting Problem Queries	239
Investigating Error Messages	239
Gathering Information for Greenplum Support.....	240

Section VII: Extending Greenplum Database

Chapter 28: Using Greenplum MapReduce	242
About Greenplum MapReduce	242
The Basics of MapReduce.....	242
How Greenplum MapReduce Works.....	243
Programming Greenplum MapReduce.....	244
Defining Inputs.....	244
Defining Map Functions.....	247
Defining Reduce Functions.....	249
Defining Outputs.....	251
Defining Tasks	252
Putting Together a Complete MapReduce Specification	253
Submitting MapReduce Jobs for Execution	253
Troubleshooting Problems with MapReduce Jobs	254
Language Does Not Exist	254
Generic Python Iterator Error	254
Function Defined Using Wrong MODE.....	255

Section VIII: References

Appendix A: SQL Command Reference	259
SQL Syntax Summary	261
ABORT	286
ALTER AGGREGATE	287
ALTER CONVERSION	289
ALTER DATABASE.....	290
ALTER DOMAIN	292
ALTER FUNCTION	294
ALTER GROUP	297
ALTER INDEX	298
ALTER LANGUAGE	300
ALTER OPERATOR	301
ALTER OPERATOR CLASS	302
ALTER RESOURCE QUEUE	303
ALTER ROLE	305
ALTER SCHEMA	308
ALTER SEQUENCE	309
ALTER TABLE	312
ALTER TABLESPACE	324
ALTER TRIGGER	325

ALTER TYPE.....	326
ALTER USER.....	327
ANALYZE.....	328
BEGIN.....	330
CHECKPOINT.....	332
CLOSE.....	333
CLUSTER.....	334
COMMENT.....	337
COMMIT.....	340
COPY.....	341
CREATE AGGREGATE.....	350
CREATE CAST.....	354
CREATE CONVERSION.....	357
CREATE DATABASE.....	359
CREATE DOMAIN.....	361
CREATE EXTERNAL TABLE.....	363
CREATE FUNCTION.....	375
CREATE GROUP.....	381
CREATE INDEX.....	382
CREATE LANGUAGE.....	386
CREATE OPERATOR.....	389
CREATE OPERATOR CLASS.....	394
CREATE RESOURCE QUEUE.....	399
CREATE ROLE.....	402
CREATE RULE.....	406
CREATE SCHEMA.....	409
CREATE SEQUENCE.....	411
CREATE TABLE.....	415
CREATE TABLE AS.....	426
CREATE TABLESPACE.....	430
CREATE TRIGGER.....	433
CREATE TYPE.....	436
CREATE USER.....	443
CREATE VIEW.....	444
DEALLOCATE.....	447
DECLARE.....	448
DELETE.....	451
DROP AGGREGATE.....	454
DROP CAST.....	455
DROP CONVERSION.....	456
DROP DATABASE.....	457
DROP DOMAIN.....	458
DROP EXTERNAL TABLE.....	459
DROP FUNCTION.....	460
DROP GROUP.....	462
DROP INDEX.....	463
DROP LANGUAGE.....	464
DROP OPERATOR.....	465
DROP OPERATOR CLASS.....	467

DROP OWNED	469
DROP RESOURCE QUEUE	471
DROP ROLE	473
DROP RULE	474
DROP SCHEMA	475
DROP SEQUENCE	476
DROP TABLE	477
DROP TABLESPACE	478
DROP TRIGGER	479
DROP TYPE.....	480
DROP USER.....	481
DROP VIEW	482
END	483
EXECUTE.....	484
EXPLAIN.....	485
FETCH	488
GRANT	492
INSERT	497
LOAD	499
LOCK.....	500
MOVE	504
PREPARE	506
REASSIGN OWNED.....	509
REINDEX	510
RELEASE SAVEPOINT	512
RESET	513
REVOKE	514
ROLLBACK.....	517
ROLLBACK TO SAVEPOINT	518
SAVEPOINT	520
SELECT	522
SELECT INTO	538
SET	540
SET ROLE.....	542
SET SESSION AUTHORIZATION	544
SET TRANSACTION	546
SHOW	549
START TRANSACTION	550
TRUNCATE	552
UPDATE.....	553
VACUUM.....	557
VALUES.....	560
Appendix B: Management Utility Reference.....	563
Backend Server Programs	564
Management Utility Summary	565
gp_dump.....	583
gp_restore	588
gpactivatestandby	592

gpaddmirrors	595
gpchecknet	599
gpcheckos	602
gpcheckperf	604
gpcrondump	607
gpdbrestore	612
gpdeletesystem	615
gpdetective	617
gpexpand	619
gpfdist	623
gpinitstandby	625
gpinitssystem	628
gpload	633
gplogfilter	643
gpmapproduce	647
gpmigrator	650
gprebuildsystem	654
gprecoverseg	657
gpscp	661
gpsizecalc	663
gpskew	666
gpssh	669
gpssh-exkeys	671
gpstart	674
gpstate	676
gpstop	679
Appendix C: Client Utility Reference	682
Client Utility Summary	684
clusterdb	694
createdb	696
createlang	698
dropdb	700
droplang	702
dropuser	704
ecpg	706
pg_config	708
pg_dump	711
createuser	718
pg_dumpall	721
pg_restore	725
psql	730
reindexdb	751
vacuumdb	753
Appendix D: Server Configuration Parameters	755
add_missing_from	756
array_nulls	756
authentication_timeout	756
autovacuum	756

autovacuum_analyze_scale_factor	756
autovacuum_analyze_threshold	756
autovacuum_freeze_max_age.....	756
autovacuum_naptime.....	757
autovacuum_vacuum_cost_delay.....	757
autovacuum_vacuum_cost_limit	757
autovacuum_vacuum_scale_factor.....	757
autovacuum_vacuum_threshold.....	757
backslash_quote	757
bgwriter_all_maxpages	757
bgwriter_all_percent	758
bgwriter_delay	758
bgwriter_lru_maxpages.....	758
bgwriter_lru_percent.....	758
block_size	758
bonjour_name.....	758
check_function_bodies	758
checkpoint_segments.....	758
checkpoint_timeout.....	758
checkpoint_warning	759
client_encoding	759
client_min_messages	759
commit_delay	759
commit_siblings	759
config_file	759
constraint_exclusion.....	760
cpu_index_tuple_cost	760
cpu_operator_cost	760
cpu_tuple_cost.....	760
cursor_tuple_fraction	760
custom_variable_classes	760
data_directory.....	760
DateStyle	760
db_user_namespace	761
deadlock_timeout.....	761
debug_assertions	761
debug_pretty_print	761
debug_print_parse	761
debug_print_plan	761
debug_print_prelim_plan	761
debug_print_rewritten.....	761
debug_print_slice_table	761
default_statistics_target.....	762
default_tablespace	762
default_transaction_isolation.....	762
default_transaction_read_only	762
default_with_oids.....	762
dynamic_library_path.....	762
effective_cache_size	762

enable_bitmapscan	763
enable_groupagg	763
enable_hashagg	763
enable_hashjoin	763
enable_indexscan.....	763
enable_mergejoin	763
enable_nestloop.....	763
enable_seqscan.....	763
enable_sort.....	764
enable_tidscan	764
escape_string_warning.....	764
explain_pretty_print.....	764
external_pid_file	764
extra_float_digits	764
from_collapse_limit.....	764
fsynch	764
full_page_writes.....	765
gin_fuzzy_search_limit.....	765
gp_adjust_selectivity_for_outerjoins	765
gp_analyze_relative_error.....	765
gp_autostats_mode	766
gp_autostats_on_change_threshold	766
gp_cached_segworkers_threshold.....	766
gp_command_count.....	766
gp_connections_per_thread	767
gp_debug_linger	767
gp_enable_adaptive_nestloop	767
gp_enable_agg_distinct	767
gp_enable_agg_distinct_pruning.....	767
gp_enable_fallback_plan	767
gp_enable_fast_sri.....	767
gp_enable_gpperfmon	767
gp_enable_groupect_distinct_gather	768
gp_enable_groupect_distinct_pruning.....	768
gp_enable_multiphase_agg.....	768
gp_enable_predicate_propagation.....	768
gp_enable_preunique.....	768
gp_enable_sequential_window_plans	768
gp_enable_sort_distinct	768
gp_enable_sort_limit.....	769
gp_external_enable_exec.....	769
gp_external_grant_privileges	769
gp_external_max_segs	769
gp_fault_action	769
gp_fts_probe_interval	769
gp_fts_probe_threadcount	769
gp_gpperfmon_send_interval.....	769
gp_hashagg_compress_spill_files.....	770
gp_hashjoin_tuples_per_bucket.....	770

gp_interconnect_hash_multiplier.....	770
gp_interconnect_queue_depth	770
gp_interconnect_setup_timeout.....	770
gp_interconnect_type.....	770
gp_log_format	771
gp_log_gang	771
gp_log_interconnect.....	771
gp_max_csv_line_length.....	771
gp_max_local_distributed_cache.....	771
gp_max_packet_size.....	771
gp_motion_cost_per_row.....	771
gp_reject_percent_threshold.....	771
gp_reraise_signal.....	772
gp_role	772
gp_safefswritesize.....	772
gp_segment_connect_timeout	772
gp_segments_for_planner.....	772
gp_session_id	772
gp_set_proc_affinity	772
gp_set_read_only	772
gp_statistics_pullup_from_child_ partition.....	772
gp_statistics_use_fkeys	773
gp_use_dispatch_agent	773
gp_vmem_protect_gang_cache_limit.....	773
gp_vmem_protect_limit	773
gpperfmon_port.....	773
hba_file.....	773
ident_file.....	773
integer_datetimes	774
IntervalStyle	774
join_collapse_limit	774
krb_caseins_users.....	774
krb_server_hostname	774
krb_server_keyfile	774
krb_srvname.....	774
lc_collate.....	775
lc_ctype	775
lc_messages	775
lc_monetary	775
lc_numeric	775
lc_time.....	775
listen_addresses	775
local_preload_libraries	776
log_autostats	776
log_connections	776
log_disconnections.....	776
log_dispatch_stats	776
log_duration.....	776

log_error_verbosity.....	776
log_executor_stats.....	776
log_filename.....	776
log_hostname.....	776
log_min_duration_statement.....	777
log_min_error_statement.....	777
log_min_messages.....	777
log_parser_stats.....	777
log_planner_stats.....	777
log_rotation_age.....	777
log_rotation_size.....	778
log_statement.....	778
log_statement_stats.....	778
log_timezone.....	778
log_truncate_on_rotation.....	778
maintenance_work_mem.....	779
max_appendonly_tables.....	779
max_connections.....	779
max_files_per_process.....	779
max_fsm_pages.....	779
max_fsm_relations.....	780
max_function_args.....	780
max_identifier_length.....	780
max_index_keys.....	780
max_locks_per_transaction.....	780
max_prepared_transactions.....	780
max_resource_portals_per_transaction.....	780
max_resource_queues.....	781
max_stack_depth.....	781
password_encryption.....	781
port.....	781
random_page_cost.....	781
regex_flavor.....	781
resource_cleanup_gangs_on_wait.....	781
resource_scheduler.....	781
resource_select_only.....	782
search_path.....	782
seq_page_cost.....	782
server_encoding.....	782
server_version.....	782
server_version_num.....	782
shared_buffers.....	782
shared_preload_libraries.....	783
silent_mode.....	783
sql_inheritance.....	783
ssl.....	783
standard_conforming_strings.....	783
statement_timeout.....	783
stats_block_level.....	784

stats_command_string.....	784
stats_queue_level.....	784
stats_reset_server_on_start.....	784
stats_row_level.....	784
stats_start_collector.....	784
superuser_reserved_connections.....	784
tcp_keepalives_count.....	784
tcp_keepalives_idle.....	784
tcp_keepalives_interval.....	785
temp_buffers.....	785
TimeZone.....	785
timezone_abbreviations.....	785
transaction_isolation.....	785
transaction_read_only.....	785
transform_null_equals.....	785
unix_socket_directory.....	786
unix_socket_group.....	786
unix_socket_permissions.....	786
update_process_title.....	786
vacuum_cost_delay.....	786
vacuum_cost_limit.....	786
vacuum_cost_page_dirty.....	786
vacuum_cost_page_hit.....	786
vacuum_cost_page_miss.....	786
vacuum_freeze_min_age.....	787
wal_buffers.....	787
wal_synch_method.....	787
work_mem.....	787
Appendix E: Initialization Configuration File Reference.....	788
Required Parameters.....	788
ARRAY_NAME.....	788
MACHINE_LIST_FILE.....	788
SEG_PREFIX.....	788
PORT_BASE.....	788
DATA_DIRECTORY.....	788
MASTER_HOSTNAME.....	789
MASTER_DIRECTORY.....	789
MASTER_PORT.....	789
TRUSTED_SHELL.....	789
CHECK_POINT_SEGMENTS.....	789
ENCODING.....	789
Optional Parameters.....	790
DATABASE_NAME.....	790
Optional Parameters for Mirror Segments.....	790
MIRROR_PORT_BASE.....	790
MIRROR_DATA_DIRECTORY.....	790
Appendix F: Greenplum MapReduce Specification.....	791
Greenplum MapReduce Document Format.....	791

Greenplum MapReduce Document Schema	793
Example Greenplum MapReduce Document.....	803
MapReduce Flow Diagram.....	810
Appendix G: Greenplum Environment Variables.....	812
Required Environment Variables.....	812
GPHOME.....	812
PATH.....	812
LD_LIBRARY_PATH.....	812
MASTER_DATA_DIRECTORY.....	812
Optional Environment Variables	813
PGDATABASE	813
PGHOST	813
PGHOSTADDR	813
PGPASSWORD.....	813
PGPASSFILE	813
PGOPTIONS.....	813
PGPORT.....	813
PGUSER	813
PGDATESTYLE	813
PGTZ.....	813
PGCLIENTENCODING.....	813
Appendix H: Greenplum Database Data Types.....	814
Appendix I: System Catalog Reference	817
gp_configuration	820
gp_configuration_history.....	821
gp_db_interfaces	822
gp_interfaces	823
gp_distributed_log	824
gp_distributed_xacts.....	825
gp_distribution_policy	826
gpexpand.status	827
gpexpand.status_detail	828
gpexpand.expansion_progress	830
gp_id	831
gp_master_mirroring	832
gp_transaction_log.....	833
gp_pgdatabase	834
gp_version_at_initdb.....	835
pg_aggregate.....	836
pg_am	837
pg_amop.....	839
pg_amproc.....	840
pg_attrdef.....	841
pg_attribute	842
pg_auth_members	844
pg_authid.....	845
pg_autovacuum	846

pg_cast.....	847
pg_class.....	848
pg_constraint.....	851
pg_conversion.....	852
pg_database.....	853
pg_depend.....	855
pg_description.....	856
pg_exttable.....	857
pg_index.....	858
pg_inherits.....	860
pg_language.....	861
pg_largeobject.....	862
pg_listener.....	863
pg_locks.....	864
pg_namespace.....	866
pg_opclass.....	867
pg_operator.....	868
pg_partition.....	869
pg_partitions.....	870
pg_partition_columns.....	872
pg_partition_rule.....	873
pg_partition_templates.....	874
pg_pltemplate.....	875
pg_proc.....	876
pg_type.....	878
pg_resqueue.....	881
pg_resqueue_status.....	882
pg_rewrite.....	883
pg_roles.....	884
pg_shdepend.....	885
pg_shdescription.....	886
pg_stat_activity.....	887
pg_statistic.....	888
pg_stat_resqueues.....	890
pg_tablespace.....	891
pg_trigger.....	892
pg_window.....	893
Appendix J: SQL 2008 Optional Feature Compliance.....	895
Glossary.....	916

Copyright © 2008 by Greenplum, Inc. All rights reserved.

This publication pertains to Greenplum software and to any subsequent release until otherwise indicated in new editions or technical notes.

Information in this document is subject to change without notice. The software described herein is furnished under a license agreement, and it may be used or copied only in accordance with the terms of that agreement. Upgrades are provided only at regularly scheduled software release dates. No part of this publication may be reproduced, transmitted, or translated in any form or by any means, electronic, mechanical, manual, optical, or otherwise, without the prior written permission of Greenplum, Inc. Greenplum makes no warranty of any kind with respect to the completeness or accuracy of this manual.

Greenplum, the Greenplum logo, and Greenplum Database are trademarks of Greenplum, Inc. PostgreSQL is a trademark of Marc Fournier held in trust for The PostgreSQL Development Group. All other company and product names used herein may be trademarks or registered trademarks of their respective companies.

Some Greenplum Database management tools make use of the Apache Portable Runtime library (<http://www.apache.org/licenses>) and the Eventlib library Copyright 2000-2004 Niels Provos and are subject to their respective license agreements.

Preface

This guide provides information for system administrators and database super users responsible for administering a Greenplum Database system.

- [About This Guide](#)
- [Greenplum Database Documentation](#)
- [Contact Us](#)

About This Guide

This guide provides information and instructions for installing, configuring and maintaining a Greenplum Database system. This guide is intended for system and database administrators responsible for managing a Greenplum Database system.

This guide assumes knowledge of Linux/Unix system administration, database management systems, database administration, and structured query language (SQL).

Because Greenplum Database is based on PostgreSQL 8.2.14, this guide assumes some familiarity with PostgreSQL. Links and cross-references to [PostgreSQL documentation](#) are provided throughout this guide for features that are similar to those in Greenplum Database.

This guide contains the following main sections:

- [Section I, “Introduction to Greenplum”](#) explains the distributed architecture and parallel processing concepts of Greenplum Database.
- [Section II, “Installation and Initialization”](#) provides instructions for getting a Greenplum Database system up and running.
- [Section III, “Access Control and Security”](#) explains how clients connect to a Greenplum Database system, and how to configure access control and workload management.
- [Section IV, “Database Administration”](#) explains how to do basic database administration tasks such as defining database objects, loading data, writing queries and managing data.
- [Section V, “System Administration”](#) explains the various system administration tasks of Greenplum Database such as configuring the server, monitoring system activity, enabling high-availability, backing up and restoring databases, and other routine system administration tasks.
- [Section VI, “Performance Tuning”](#) provides guidance on identifying and troubleshooting the most common causes of performance issues in Greenplum Database.
- [Section VII, “Extending Greenplum Database”](#) describes how to extend the functionality of Greenplum Database by developing your own functions and programs.
- [Section VIII, “References”](#) contains reference documentation for SQL commands, command-line utilities, client programs, system catalogs, and configuration parameters.

Greenplum Database Documentation

The Greenplum Database documentation is provided in PDF format. The *Administrator Guide* can be found in the `$GPHOME/docs` directory of your Greenplum Database installation. Both the *Administrator Guide* and the release notes are available for download on the [Greenplum Network](#).

The Greenplum Database documentation is intended as a supplement to the [PostgreSQL 8.2.14 documentation](#), upon which Greenplum Database is based.

The following documents are provided with this release of Greenplum Database:

- **Release Notes** — The release notes explain the new features and any known issues associated with this release.
- **Greenplum Database Administrator Guide** — This guide contains conceptual, instructional and reference information for database and system administrators responsible for installing, configuring and administering a Greenplum Database system.

Document Conventions

The following conventions are used throughout the Greenplum Database documentation to help you identify certain types of information.

Text Conventions

Table 0.1 Text Conventions

Text Convention	Usage	Examples
bold	Button, menu, tab, page, and field names in GUI applications	Click Cancel to exit the page without saving your changes.
<i>italics</i>	New terms where they are defined Database objects, such as schema, table, or columns names	The <i>master instance</i> is the postmaster process that accepts client connections. Catalog information for Greenplum Database resides in the <i>pg_catalog</i> schema.
monospace	File names and path names Programs and executables Command names and syntax Parameter names	Edit the <code>postgresql.conf</code> file. Use <code>gpstart</code> to start Greenplum Database.
<i>monospace italics</i>	Variable information within file paths and file names Variable information within command syntax	<code>/home/gpadmin/config_file</code> <code>COPY tablename FROM</code> <code>'filename'</code>

Table 0.1 Text Conventions

Text Convention	Usage	Examples
monospace bold	Used to call attention to a particular part of a command, parameter, or code snippet.	Change the host name, port, and database name in the JDBC connection URL: jdbc:postgresql:// host:5432/m ydb
UPPERCASE	Environment variables SQL commands Keyboard keys	Make sure that the Java /bin directory is in your \$PATH. SELECT * FROM my_table; Press CTRL+C to escape.

Command Syntax Conventions

Table 0.2 Command Syntax Conventions

Text Convention	Usage	Examples
{ }	Within command syntax, curly braces group related command options. Do not type the curly braces.	FROM { 'filename' STDIN }
[]	Within command syntax, square brackets denote optional arguments. Do not type the brackets.	TRUNCATE [TABLE] name
...	Within command syntax, an ellipsis denotes repetition of a command, variable, or option. Do not type the ellipsis.	DROP TABLE name [, ...]
	Within command syntax, the pipe symbol denotes an “OR” relationship. Do not type the pipe symbol.	VACUUM [FULL FREEZE]
\$ <i>system_command</i> # <i>root_system_command</i> => <i>gpdb_command</i> =# <i>su_gpdb_command</i>	Denotes a command prompt - do not type the prompt symbol. \$ and # denote terminal command prompts. => and =# denote Greenplum Database interactive program command prompts (psql or gpssh, for example).	\$ createdb mydatabase # chown gpadmin -R /datadir => SELECT * FROM mytable; =# SELECT * FROM pg_database;

Naming Conventions and Acronyms

Table 0.3 Naming Conventions and Acronyms

Convention	Meaning
\$GPHOME	The base directory where Greenplum Database is installed, for example: <code>/usr/local/greenplum-db-3.3.7.x</code>
gpadmin	The default name for the Greenplum Database super user.
KB	Kilobytes (2^{10})
MB	Megabytes (2^{20})
GB	Gigabytes (2^{30})
TB	Terabytes (2^{40})

Contact Us

To contact Greenplum customer support, open a support incident. Authorized Customer Greenplum Subject Matter Experts and Central Point of Contact Administrators can log a support incident on the [support portal](#). If you are a Greenplum Subject Matter Expert or Central Point of Contact at your company, and do not have access, please contact entitlement@greenplum.com.

Section I: Introduction to Greenplum

Greenplum Database is a massively parallel processing (MPP) database server based on PostgreSQL open-source technology. MPP (also known as a *shared nothing* architecture) refers to systems with two or more processors which cooperate to carry out an operation - each processor with its own memory, operating system and disks. Greenplum leverages this high-performance system architecture to distribute the load of multi-terabyte data warehouses, and is able to use all of a system's resources in parallel to process a query.

Greenplum Database is essentially several PostgreSQL database instances acting together as one cohesive database management system. It is based on PostgreSQL 8.2.14, and in most cases is very similar to PostgreSQL with regards to SQL support, features, configuration options, and end-user functionality. Database users interact with Greenplum Database as they would a regular PostgreSQL DBMS.

The internals of PostgreSQL have been modified or supplemented to support the parallel structure of Greenplum Database. For example the system catalog, query planner, optimizer, query executor, and transaction manager components have been modified and enhanced to be able to execute queries in parallel across all of the PostgreSQL database instances at once. The Greenplum *interconnect* (the networking layer) enables communication between the distinct PostgreSQL instances and allows the system behave as one logical database.

Greenplum Database also includes features designed to optimize PostgreSQL for business intelligence (BI) workloads. For example, Greenplum has added parallel data loading (external tables), resource management, query optimizations and storage enhancements which are not found in regular PostgreSQL. Many features and optimizations developed by Greenplum do make their way back into the PostgreSQL community. For example, table partitioning is a feature developed by Greenplum which is now in standard PostgreSQL.

To learn more about Greenplum Database, refer to the following topics:

[About the Greenplum Architecture](#)

[About Distributed Databases](#)

[About Greenplum Query Processing](#)

[Summary of Greenplum Features](#)

1. About the Greenplum Architecture

Greenplum Database is able to handle the storage and processing of large amounts of data by distributing the load across several servers or *hosts*. A database in Greenplum is actually an *array* of individual PostgreSQL databases, all working together to present a single database image. The *master* is the entry point to the Greenplum Database system. It is the database instance where clients connect and submit SQL statements. The master coordinates the work with the other database instances in the system, the *segments*, which handle data processing and storage.

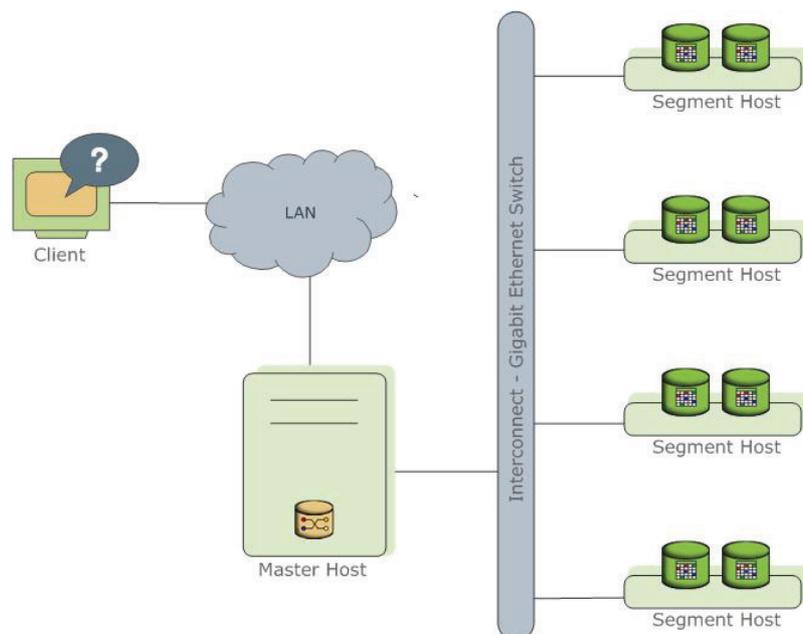


Figure 1.1 High-Level Greenplum Database Architecture

This section describes all of the components that comprise a Greenplum Database system, and how they work together:

- [About the Greenplum Master](#)
- [About the Greenplum Segments](#)
- [About the Greenplum Interconnect](#)
- [About Redundancy and Failover in Greenplum Database](#)
- [About Parallel Data Loading](#)
- [About Management and Monitoring](#)

About the Greenplum Master

The *master* is the entry point to the Greenplum Database system. It is the database process that accepts client connections and processes the SQL commands issued by the users of the system.

Since Greenplum Database is based on PostgreSQL, end-users interact with Greenplum Database (through the master) as they would a typical PostgreSQL database. They can connect to the database using client programs such as `psql` or application programming interfaces (APIs) such as JDBC or ODBC.

The master is where the *global system catalog* resides (the set of system tables that contain metadata about the Greenplum Database system itself), however the master does not contain any user data. Data resides only on the *segments*. The master does the work of authenticating client connections, processing the incoming SQL commands, distributing the work load between the segments, coordinating the results returned by each of the segments, and presenting the final results to the client program.

About the Greenplum Segments

In Greenplum Database, the *segments* are where the data is stored and where the majority of query processing takes place. User-defined tables and their indexes are distributed across the available number of segments in the Greenplum Database system, each segment containing a distinct portion of the data. Segment instances are the database server processes that serve segments. Users do not interact directly with the segments in a Greenplum Database system, but do so through the master.

In the recommended Greenplum Database hardware configuration, there is one active segment per effective CPU or CPU core. For example, if your segment hosts have two dual-core processors, you would have four primary segments per host.

About the Greenplum Interconnect

The *interconnect* is the networking layer of Greenplum Database. When a user connects to a database and issues a query, processes are created on each of the segments to handle the work of that query (see [“Understanding Parallel Query Execution”](#) on page 28). The *interconnect* refers to the inter-process communication between the segments, as well as the network infrastructure on which this communication relies. The interconnect uses a standard Gigabit Ethernet switching fabric.

By default, the interconnect uses UDP (User Datagram Protocol) to send messages over the network. The Greenplum software does the additional packet verification and checking not performed by UDP, so the reliability is equivalent to TCP (Transmission Control Protocol), and the performance and scalability exceeds TCP. With TCP, Greenplum has a scalability limit of 1000 segment instances. To remove this limit, UDP is now the default protocol for the interconnect.

About Redundancy and Failover in Greenplum Database

Greenplum Database has deployment options to provide for a system without a single point of failure. This section explains the redundancy components of Greenplum Database.

- [About Segment Mirroring](#)
- [About Master Mirroring](#)
- [About Interconnect Redundancy](#)

About Segment Mirroring

When you deploy your Greenplum Database system, you have the option to configure *mirror* segments. Mirror segments allow database queries to fail over to a backup segment if the primary segment becomes unavailable. To configure mirroring, you must have enough hosts in your Greenplum Database system so that the secondary segment always resides on a different host than its primary. [Figure 1.2](#) shows how table data is distributed across the segments when mirroring is configured. The mirror segment always resides on a different host than its primary segment.

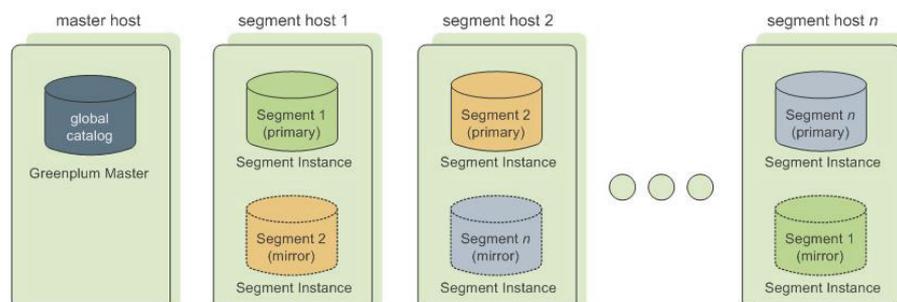


Figure 1.2 Data Mirroring in Greenplum Database

Segment Failover and Recovery

When mirroring is enabled in a Greenplum Database system, the system will automatically fail over to the mirror copy whenever a primary copy becomes unavailable. A Greenplum Database system can remain operational if a segment instance or host goes down as long as all portions of data are available on the remaining active segments.

Whenever the master cannot connect to a segment instance, it marks that segment instance as *invalid* in the Greenplum Database system catalog. The segment instance will remain invalid and out of operation until steps are taken to bring that segment back online. Once a segment is back online, the master will mark it as valid again the next time it successfully connects to the segment.

Recovery of a failed segment depends on the configured *fault operational mode*. If the system is running in *read-only* mode (the default), users will not be able to issue DDL or DML commands when there are failed segments in the system. In read-only mode, you can recover a segment with only a short interruption of service. If the system is running in *continue* mode, all operations will continue as long as there is one active segment instance alive per portion of data. In this mode, the data on the failed segment must be reconciled with the active segment before it can be brought back into operation. The system must be shutdown to recover failed segments.

If you do not have mirroring enabled, the system will automatically shutdown if a segment instance becomes invalid. You must recover all failed segments before operations can continue.

About Master Mirroring

You can also optionally deploy a *backup* or *mirror* of the master instance. A backup master host serves as a *warm standby* in the event of the primary master host becoming unoperational. The standby master is kept up to date by a transaction log replication process, which runs on the standby master host and keeps the data between the primary and standby master hosts synchronized. If the primary master fails, the log replication process is shutdown, and the standby master can be activated in its place. Upon activation of the standby master, the replicated logs are used to reconstruct the state of the master host at the time of the last successfully committed transaction.

Since the master does not contain any user data, only the system catalog tables need to be synchronized between the primary and backup copies. These tables are not updated frequently, but when they are, changes are automatically copied over to the standby master so that it is always kept current with the primary.

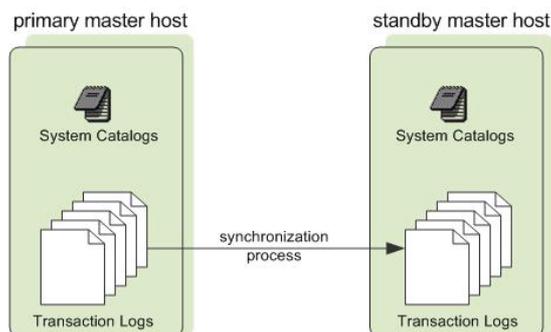


Figure 1.3 Master Mirroring in Greenplum Database

About Interconnect Redundancy

The *interconnect* refers to the inter-process communication between the segments, as well as the network infrastructure on which this communication relies. A highly available interconnect can be achieved by deploying dual Gigabit Ethernet switches on your network, and redundant Gigabit connections to the Greenplum Database host servers.

About Parallel Data Loading

One challenge of large scale, multi-terabyte data warehouses is getting large amounts of data loaded within a given maintenance window. Greenplum supports fast, parallel data loading with its external tables feature. External tables can also be accessed in ‘single row error isolation’ mode, allowing administrators to filter out bad rows during a load operation into a separate error table, while still loading properly formatted rows. Administrators can control the acceptable error threshold for a load operation, giving them control over the quality and flow of data into the database.

By using external tables in conjunction with Greenplum Database’s parallel file server (`gpfdist`), administrators can achieve maximum parallelism and load bandwidth from their Greenplum Database system. Greenplum has demonstrated load rates in excess of 2 TB an hour.

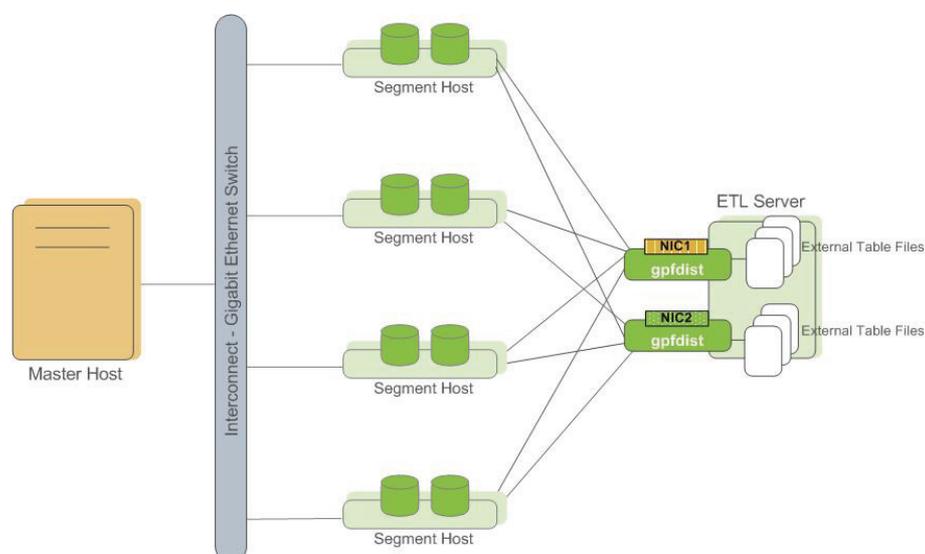


Figure 1.4 External Tables Using Greenplum Parallel File Server (`gpfdist`)

About Management and Monitoring

Management of a Greenplum Database system is performed using a series of command-line utilities, which are located in `$GPHOME/bin`. Greenplum provides utilities for the following Greenplum Database administration tasks:

- Installing Greenplum Database on an Array
- Initializing a Greenplum Database System
- Starting and Stopping Greenplum Database
- Adding or Removing a Host
- Expanding the Array and Redistributing Tables among New Segments
- Managing Recovery for Failed Segment Instances
- Managing Failover and Recovery for a Failed Master Instance

- Backing Up and Restoring a Database (in Parallel)
- Loading Data in Parallel
- System State Reporting

Greenplum also provides an optional performance monitoring feature that administrators can install and enable with Greenplum Database. To use the Greenplum Performance Monitor, each host in your Greenplum Database array must have a monitor agent installed. When you start the Greenplum Performance Monitor, the agents begin collecting data on queries and system utilization. Segment agents send their data to the Greenplum master at regular intervals (typically every 15 seconds). Users can query the Greenplum Performance Monitor database to see query and system performance data for both active queries and historical queries. Greenplum Performance Monitor also has a graphical web-based user interface for viewing these performance metrics.

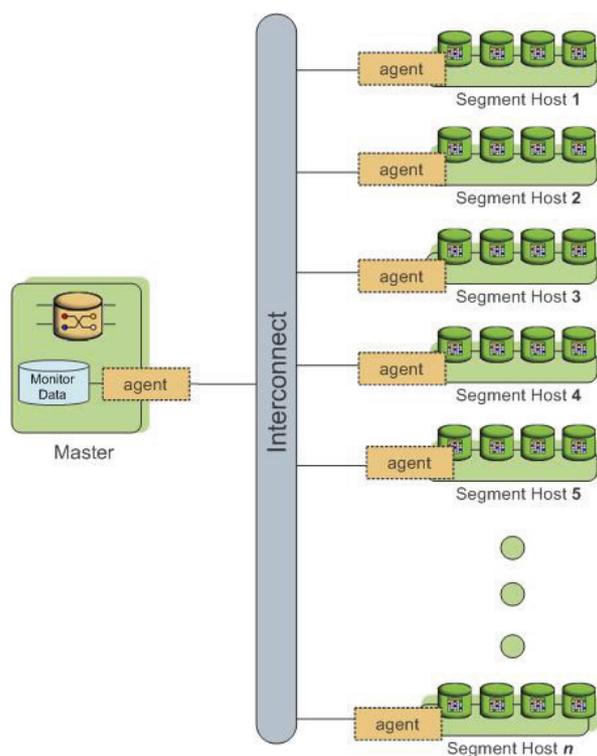


Figure 1.5 Greenplum Performance Monitor Architecture

2. About Distributed Databases

Greenplum is a *distributed database* system. This means that the data stored in the database system is physically located on more than one database server (referred to as *segments* in Greenplum). These individual database servers are connected by a communications network (referred to as the *interconnect* in Greenplum). An essential feature of a true distributed database is that users and client programs work as if they were accessing one single database on a local machine (in Greenplum, this entry-point database is referred to as the *master*). The fact that the database is distributed across several machines is seamless to the users of the system.

Understanding How Data is Stored

To understand how Greenplum Database stores data across the various hosts and segment instances, consider the following simple logical database. In [Figure 2.1](#), primary keys are shown in bold font and foreign key relationships are indicated by a line from the foreign key in the referring relation to the primary key of the referenced relation. In data warehouse terminology, this is referred to as a *star schema*. In this type of database schema, the sale table is usually called a *fact table* and the other tables (customer, vendor, product) are usually called the *dimension tables*.

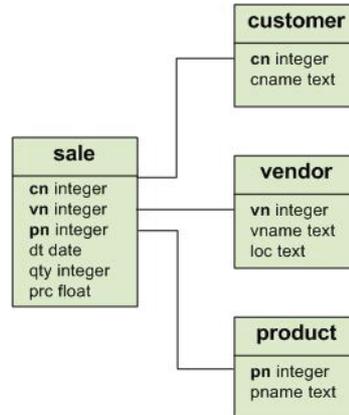


Figure 2.1 Sample Database Star Schema

In Greenplum Database all tables are *distributed*, which means a table is divided into non-overlapping sets of rows or parts. Each part resides on a single database known as a *segment* within the Greenplum Database system. The parts are distributed across all of the available segments using a sophisticated hashing algorithm. Database administrators choose the hash key (one or more table columns) when defining the table.

The Greenplum Database physical database implements the logical database on an array of individual database instances — a *master instance* and two or more *segment instances*. The master instance does not contain any user data, only the global catalog tables. The segment instances contain disjoint parts (collections of rows) for each distributed table.

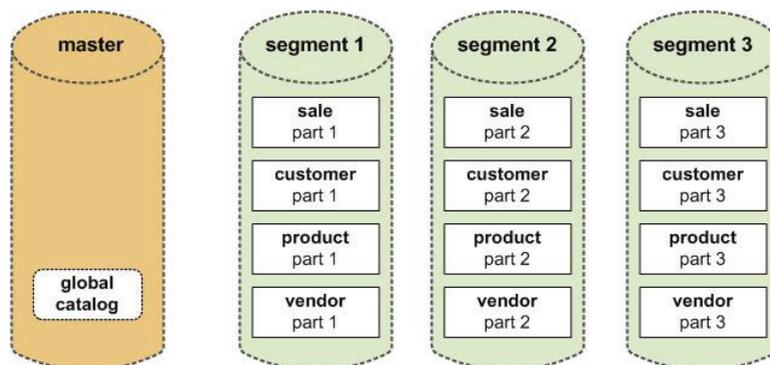


Figure 2.2 Table Distribution in a Greenplum Database Physical Database

Understanding Greenplum Distribution Policies

When you create or alter a table in Greenplum Database, there is an additional `DISTRIBUTED` clause to define the *distribution policy* of the table. The distribution policy determines how to divide the rows of a table across the Greenplum segments. Greenplum Database provides two types of distribution policy:

Hash Distribution - With hash distribution, one or more table columns is used as the *distribution key* for the table. The distribution key is used by a hashing algorithm to assign each row to a particular segment. Keys of the same value will always hash to the same segment. Choosing a unique distribution key, such as a primary key, will ensure the most even data distribution. Hash distribution is the default distribution policy for a table. If a `DISTRIBUTED` clause is not supplied, then either the `PRIMARY KEY` (if the table has one) or the first column of the table will be used as the table distribution key.

Random Distribution - With random distribution, rows are sent to the segments as they come in, cycling across the segments in a round-robin fashion. Rows with columns having the same values will not necessarily be located on the same segment. Although a random distribution ensures even data distribution, there are performance advantages to choosing a hash distribution policy whenever possible.

3. Summary of Greenplum Features

This section provides a high-level overview of the system requirements and feature set of Greenplum Database. It contains the following topics:

- [System Requirements](#)
- [Feature Summary](#)

System Requirements

The following table lists minimum recommended specifications for servers intended to support Greenplum Database in a production environment. Greenplum also provides hardware build guides for its certified hardware platforms. It is recommended that you work with a Greenplum Systems Engineer to review your anticipated environment to ensure an appropriate hardware configuration for Greenplum Database.

Table 3.1 System Prerequisites for Greenplum Database 3.3.7

Operating System	SUSE Linux SLES 10 update 2 CentOS 5.0 or higher RedHat Enterprise Linux 4.0 or higher Solaris x86 v10 update 4
Filesystems	<ul style="list-style-type: none"> • xfs required for data storage on SUSE Linux and Red Hat (ext3 supported for root filesystem) • zfs required for data storage on Solaris (ufs supported for root filesystem)
Minimum CPU	Pentium Pro compatible (P3/Athlon and above)
Minimum Memory	16 GB RAM per server
Disk Requirements	<ul style="list-style-type: none"> • 150MB per host for Greenplum installation • Approximately 300MB per segment instance for meta data • Appropriate free space for data with disks at no more than 70% capacity • High-speed, local storage
Network Requirements	Gigabit Ethernet within the array Dedicated, non-blocking switch
Software and Utilities	bash shell GNU tar GNU zip GCC runtime libraries (glibc, etc.) GNU readline (Solaris only) ¹

1. On Solaris platforms, you must have GNU Readline in your environment to support interactive Greenplum administrative utilities such as [gpssh](#). Certified readline packages are available for download from the Greenplum Network.

Feature Summary

This section explains the SQL features that are currently supported in Greenplum Database.

- [Greenplum SQL Standard Conformance](#)
- [Greenplum and PostgreSQL Compatibility](#)

Greenplum SQL Standard Conformance

The SQL language was first formally standardized in 1986 by the American National Standards Institute (ANSI) as SQL 1986. Subsequent versions of the SQL standard have been released by ANSI and as International Organization for Standardization (ISO) standards: SQL 1989, SQL 1992, SQL 1999, SQL 2003, SQL 2006, and finally SQL 2008, which is the current SQL standard. The official name of the standard is ISO/IEC 9075-14:2008. In general, each new version adds more features, although occasionally features are deprecated or removed.

It is important to note that there are no commercial database systems that are fully compliant with the SQL standard. Greenplum Database is almost fully compliant with the SQL 1992 standard, with most of the features from SQL 1999. Several features from SQL 2003 have also been implemented (most notably the SQL OLAP features).

This section addresses the important conformance issues of Greenplum Database as they relate to the SQL standards. For a feature-by-feature list of Greenplum's support of the latest SQL standard, see [Appendix J, "SQL 2008 Optional Feature Compliance"](#).

Core SQL Conformance

In the process of building a parallel, shared-nothing database system and query optimizer, certain common SQL constructs are not currently implemented in Greenplum Database. The following SQL constructs are not supported:

1. `UPDATE` statements that update the distribution key columns of a hash-distributed Greenplum table. There is currently no way for the system to redistribute a row to a different segment when its hash value changes.
2. `UPDATE` and `DELETE` statements that require data to move from one segment to another. This restricts the use of joins in update and delete operations to hash-distributed tables that have the same distribution key column(s), and the join condition must specify equality on the distribution key column(s).
3. Correlated subqueries that Greenplum's parallel optimizer cannot internally rewrite into non-correlated joins. Most simple uses of correlated subqueries do work. Those that do not can be manually rewritten using outer joins.
4. Certain rare cases of multi-row subqueries that Greenplum's parallel optimizer cannot internally rewrite into equijoins.
5. Some set returning subqueries in `EXISTS` or `NOT EXISTS` clauses that Greenplum's parallel optimizer cannot rewrite into joins.

6. UNION ALL of joined tables with subqueries.
7. Set-returning functions in the FROM clause of a subquery.
8. Backwards scrolling cursors, including the use of FETCH PRIOR, FETCH FIRST, FETCH ABOLUTE, and FETCH RELATIVE.
9. In CREATE TABLE statements (on hash-distributed tables): a UNIQUE or PRIMARY KEY clause must include all of (or a superset of) the distribution key columns. Because of this restriction, only one UNIQUE clause or PRIMARY KEY clause is allowed in a CREATE TABLE statement. UNIQUE or PRIMARY KEY clauses are not allowed on randomly-distributed tables.
10. CREATE UNIQUE INDEX statements that do not contain all of (or a superset of) the distribution key columns. CREATE UNIQUE INDEX is not allowed on randomly-distributed tables.
11. VOLATILE or STABLE functions cannot execute on the segments, and so are generally limited to being passed literal values as the arguments to their parameters.
12. Triggers are not supported since they typically rely on the use of VOLATILE functions.
13. Referential integrity constraints (foreign keys) are not enforced in Greenplum Database. Users can declare foreign keys and this information is kept in the system catalog, however.
14. Sequence manipulation functions CURRVAL and LASTVAL.
15. DELETE WHERE CURRENT OF and UPDATE WHERE CURRENT OF (positioned delete and positioned update operations).

SQL 1992 Conformance

The following features of SQL 1992 are not supported in Greenplum Database:

1. NATIONAL CHARACTER (NCHAR) and NATIONAL CHARACTER VARYING (NVARCHAR). Users can declare the NCHAR and NVARCHAR types, however they are just synonyms for CHAR and VARCHAR in Greenplum Database.
2. CREATE ASSERTION statement.
3. INTERVAL literals are supported in Greenplum Database, but do not conform to the standard.
4. GET DIAGNOSTICS statement.
5. GRANT INSERT or UPDATE privileges on columns. Privileges can only be granted on tables in Greenplum Database.

6. GLOBAL TEMPORARY TABLES and LOCAL TEMPORARY TABLES. Greenplum TEMPORARY TABLES do not conform to the SQL standard, but many commercial database systems have implemented temporary tables in the same way. Greenplum temporary tables are the same as VOLATILE TABLES in Teradata.
7. UNIQUE predicate.
8. MATCH PARTIAL for referential integrity checks (most likely will not be implemented in Greenplum Database).

SQL 1999 Conformance

The following features of SQL 1999 are not supported in Greenplum Database:

1. Large Object data types: BLOB, CLOB, NCLOB. However, the BYTEA and TEXT columns can store very large amounts of data in Greenplum Database (hundreds of megabytes).
2. Recursive WITH clause or the WITH RECURSIVE clause (recursive queries). Non-recursive WITH clauses can easily be rewritten by moving the common table expression into the FROM clause as a derived table.
3. MODULE (SQL client modules).
4. CREATE PROCEDURE (SQL/PSM). This can be worked around in Greenplum Database by creating a FUNCTION that returns void, and invoking the function as follows:


```
SELECT myfunc(args);
```
5. The PostgreSQL/Greenplum function definition language (PL/PGSQL) is a subset of Oracle's PL/SQL, rather than being compatible with the SQL/PSM function definition language. Greenplum Database also supports function definitions written in Python, Perl, and R.
6. BIT and BIT VARYING data types (intentionally omitted). These were deprecated in SQL 2003, and replaced in SQL 2008.
7. Greenplum supports identifiers up to 63 characters long. The SQL standard requires support for identifiers up to 128 characters long.
8. Prepared transactions (PREPARE TRANSACTION, COMMIT PREPARED, ROLLBACK PREPARED). This also means Greenplum does not support XA Transactions (2 phase commit coordination of database transactions with external transactions).
9. CHARACTER SET option on the definition of CHAR() or VARCHAR() columns.
10. Specification of CHARACTERS or OCTETS (BYTES) on the length of a CHAR() or VARCHAR() column. For example, VARCHAR(15 CHARACTERS) or VARCHAR(15 OCTETS) or VARCHAR(15 BYTES).
11. CURRENT_SCHEMA function.
12. CREATE DISTINCT TYPE statement. CREATE DOMAIN can be used as a work-around in Greenplum.

13. The *explicit table* construct.

SQL 2003 Conformance

The following features of SQL 2003 are not supported in Greenplum Database:

- 1.** XML data type (PostgreSQL does support this).
- 2.** MERGE statements.
- 3.** IDENTITY columns and the associated GENERATED ALWAYS/GENERATED BY DEFAULT clause. The SERIAL or BIGSERIAL data types are very similar to INT or BIGINT GENERATED BY DEFAULT AS IDENTITY.
- 4.** MULTISET modifiers on data types.
- 5.** ROW data type.
- 6.** Greenplum Database syntax for using sequences is non-standard. For example, `nextval('seq')` is used in Greenplum instead of the standard `NEXT VALUE FOR seq`.
- 7.** GENERATED ALWAYS AS columns. Views can be used as a work-around.
- 8.** The sample clause (TABLESAMPLE) on SELECT statements. The `random()` function can be used as a work-around to get random samples from tables.
- 9.** NULLS FIRST/NULLS LAST clause on SELECT statements and subqueries (nulls are always last in Greenplum Database).
- 10.** The *partitioned join tables* construct (PARTITION BY in a join).
- 11.** GRANT SELECT privileges on columns. Privileges can only be granted on tables in Greenplum Database. Views can be used as a work-around.
- 12.** For CREATE TABLE x (LIKE(y)) statements, Greenplum does not support the [INCLUDING|EXCLUDING] [DEFAULTS|CONSTRAINTS|INDEXES] clauses.
- 13.** Greenplum array data types are almost SQL standard compliant with some exceptions. Generally customers should not encounter any problems using them.

SQL 2008 Conformance

The following features of SQL 2008 are not supported in Greenplum Database:

- 1.** BINARY and VARBINARY data types. BYTEA can be used in place of VARBINARY in Greenplum Database.
- 2.** FETCH FIRST or FETCH NEXT clause for SELECT, for example:

```
SELECT id, name FROM tabl ORDER BY id OFFSET 20 ROWS FETCH
NEXT 10 ROWS ONLY;
```

 Greenplum has LIMIT and LIMIT OFFSET clauses instead.

3. The `ORDER BY` clause is ignored in views and subqueries unless a `LIMIT` clause is also used. This is intentional, as the Greenplum optimizer cannot determine when it is safe to avoid the sort, causing an unexpected performance impact for such `ORDER BY` clauses. To work around, you can specify a really large `LIMIT`. For example: `SELECT * FROM mytable ORDER BY 1 LIMIT 999999999`
4. The `row subquery` construct is not supported.
5. `TRUNCATE TABLE` does not accept the `CONTINUE IDENTITY` and `RESTART IDENTITY` clauses.
6. `CREATE FOREIGN DATA WRAPPER (SQL/MED)`.
7. `CREATE SERVER (SQL/MED)`.
8. `CREATE USER MAPPING (SQL/MED)`.

Greenplum and PostgreSQL Compatibility

Greenplum Database is based on PostgreSQL 8.2 with a few features added in from the 8.3 release. To support the distributed nature and typical workload of a Greenplum Database system, some SQL commands have been added or modified, and there are a few PostgreSQL features that are not supported. Greenplum has also added features not found in PostgreSQL, such as physical data distribution, parallel query optimization, external tables, resource queues for workload management and enhanced table partitioning. For full SQL syntax and references, see “[SQL Command Reference](#)” on page 259.

Table 3.2 SQL Support in Greenplum Database

SQL Command	Supported in Greenplum	Modifications, Limitations, Exceptions
ALTER AGGREGATE	YES	
ALTER CONVERSION	YES	
ALTER DATABASE	YES	
ALTER DOMAIN	YES	
ALTER FUNCTION	YES	
ALTER GROUP	YES	Deprecated as of PostgreSQL 8.1 - see ALTER ROLE
ALTER INDEX	YES	
ALTER LANGUAGE	YES	
ALTER OPERATOR	YES	
ALTER OPERATOR CLASS	NO	
ALTER RESOURCE QUEUE	YES	Greenplum Database workload management feature - not in PostgreSQL 8.2.
ALTER ROLE	YES	Greenplum Database Clauses: <code>RESOURCE QUEUE queue_name none</code>

Table 3.2 SQL Support in Greenplum Database

SQL Command	Supported in Greenplum	Modifications, Limitations, Exceptions
ALTER SCHEMA	YES	
ALTER SEQUENCE	YES	
ALTER TABLE	YES	<p>Unsupported Clauses / Options: CLUSTER ON ENABLE/DISABLE TRIGGER</p> <p>Greenplum Database Clauses: ADD DROP RENAME SPLIT EXCHANGE PARTITION SET SUBPARTITION TEMPLATE SET WITH (REORGANIZE=true false) SET DISTRIBUTED BY</p>
ALTER TABLESPACE	NO	
ALTER TRIGGER	NO	
ALTER TYPE	YES	
ALTER USER	YES	Deprecated in PostgreSQL 8.1 - see ALTER ROLE
ANALYZE	YES	
BEGIN	YES	
CHECKPOINT	YES	
CLOSE	YES	
CLUSTER	YES	
COMMENT	YES	
COMMIT	YES	
COMMIT PREPARED	NO	
COPY	YES	<p>Modified Clauses: ESCAPE [AS] 'escape' 'OFF'</p> <p>Greenplum Database Clauses: [LOG ERRORS INTO <i>error_table</i>] SEGMENT REJECT LIMIT <i>count</i> [ROWS PERCENT]</p>
CREATE AGGREGATE	YES	<p>Unsupported Clauses / Options: [, SORTOP = <i>sort_operator</i>]</p> <p>Greenplum Database Clauses: [, PREFUNC = <i>prefunc</i>]</p> <p>Limitations: The functions used to implement the aggregate must be IMMUTABLE functions.</p>
CREATE CAST	YES	
CREATE CONSTRAINT TRIGGER	NO	

Table 3.2 SQL Support in Greenplum Database

SQL Command	Supported in Greenplum	Modifications, Limitations, Exceptions
CREATE CONVERSION	YES	
CREATE DATABASE	YES	
CREATE DOMAIN	YES	
CREATE EXTERNAL TABLE	YES	Greenplum Database parallel ETL feature - not in PostgreSQL 8.2.14.
CREATE FUNCTION	YES	Limitations: Functions defined as <code>STABLE</code> or <code>VOLATILE</code> can be executed in Greenplum Database provided that are executed on the master only. <code>STABLE</code> and <code>VOLATILE</code> functions cannot be used in statements that execute at the segment level.
CREATE GROUP	YES	Deprecated in PostgreSQL 8.1 - see CREATE ROLE
CREATE INDEX	YES	Greenplum Database Clauses: <code>USING bitmap</code> (bitmap indexes) Limitations: <code>UNIQUE</code> indexes are allowed only if they contain all of (or a superset of) the Greenplum distribution key columns. Unsupported Clauses / Options: <code>CONCURRENTLY</code> not allowed for bitmap indexes
CREATE LANGUAGE	YES	
CREATE OPERATOR	YES	Limitations: The function used to implement the operator must be an <code>IMMUTABLE</code> function.
CREATE OPERATOR CLASS	NO	
CREATE OPERATOR FAMILY	NO	
CREATE RESOURCE QUEUE	YES	Greenplum Database workload management feature - not in PostgreSQL 8.2.14.
CREATE ROLE	YES	Greenplum Database Clauses: <code>RESOURCE QUEUE queue_name none</code>
CREATE RULE	YES	
CREATE SCHEMA	YES	
CREATE SEQUENCE	YES	Limitations: <ul style="list-style-type: none"> • The <code>lastval</code> and <code>currval</code> functions are not supported. • The <code>setval</code> function only allowed in queries that do not operate on distributed data. • The <code>nextval</code> function not allowed in <code>UPDATE</code> or <code>DELETE</code> queries if mirrors are enabled.

Table 3.2 SQL Support in Greenplum Database

SQL Command	Supported in Greenplum	Modifications, Limitations, Exceptions
CREATE TABLE	YES	<p>Unsupported Clauses / Options:</p> <p>[GLOBAL LOCAL] REFERENCES FOREIGN KEY [DEFERRABLE NOT DEFERRABLE]</p> <p>Limited Clauses:</p> <ul style="list-style-type: none"> UNIQUE or PRIMARY KEY constraints are only allowed on hash-distributed tables (DISTRIBUTED BY), and the constraint columns must be the same as or a superset of the table's distribution key columns. <p>Greenplum Database Clauses:</p> <p>DISTRIBUTED BY (column, [...]) DISTRIBUTED RANDOMLY PARTITION BY type (column [, ...]) (partition_specification, [...]) WITH (appendonly=true [,compresslevel=value,blocksize=value])</p>
CREATE TABLE AS	YES	See CREATE TABLE
CREATE TABLESPACE	NO	<p>Greenplum Database Clauses:</p> <p>LOCATION 'segdir', 'segdir', ... MIRROR LOCATION 'segdir', 'segdir', ...</p>
CREATE TRIGGER	NO	
CREATE TYPE	YES	<p>Limitations:</p> <p>The functions used to implement a new base type must be IMMUTABLE functions.</p>
CREATE USER	YES	Deprecated in PostgreSQL 8.1 - see CREATE ROLE
CREATE VIEW	YES	
DEALLOCATE	YES	
DECLARE	YES	<p>Unsupported Clauses / Options:</p> <p>SCROLL FOR UPDATE [OF column [, ...]] }]</p> <p>Limitations:</p> <p>Cursors are non-updateable, and cannot be backward-scrolled. Forward scrolling is supported.</p>
DELETE	YES	<p>Unsupported Clauses / Options:</p> <p>RETURNING</p> <p>Limitations:</p> <ul style="list-style-type: none"> Joins must be on a common Greenplum distribution key (equijoins) Cannot use STABLE or VOLATILE functions in a DELETE statement if mirrors are enabled

Table 3.2 SQL Support in Greenplum Database

SQL Command	Supported in Greenplum	Modifications, Limitations, Exceptions
DROP AGGREGATE	YES	
DROP CAST	YES	
DROP CONVERSION	YES	
DROP DATABASE	YES	
DROP DOMAIN	YES	
DROP EXTERNAL TABLE	YES	Greenplum Database parallel ETL feature - not in PostgreSQL 8.2.14.
DROP FUNCTION	YES	
DROP GROUP	YES	Deprecated in PostgreSQL 8.1 - see DROP ROLE
DROP INDEX	YES	
DROP LANGUAGE	YES	
DROP OPERATOR	YES	
DROP OPERATOR CLASS	NO	
DROP OWNED	NO	
DROP RESOURCE QUEUE	YES	Greenplum Database workload management feature - not in PostgreSQL 8.2.14.
DROP ROLE	YES	
DROP RULE	YES	
DROP SCHEMA	YES	
DROP SEQUENCE	YES	
DROP TABLE	YES	
DROP TABLESPACE	NO	
DROP TRIGGER	NO	
DROP TYPE	YES	
DROP USER	YES	Deprecated in PostgreSQL 8.1 - see DROP ROLE
DROP VIEW	YES	
END	YES	
EXECUTE	YES	
EXPLAIN	YES	

Table 3.2 SQL Support in Greenplum Database

SQL Command	Supported in Greenplum	Modifications, Limitations, Exceptions
FETCH	YES	Unsupported Clauses / Options: LAST PRIOR BACKWARD BACKWARD ALL Limitations: Cannot fetch rows in a nonsequential fashion; backward scan is not supported.
GRANT	YES	
INSERT	YES	Unsupported Clauses / Options: RETURNING
LISTEN	NO	
LOAD	YES	
LOCK	YES	
MOVE	YES	See FETCH
NOTIFY	NO	
PREPARE	YES	
PREPARE TRANSACTION	NO	
REASSIGN OWNED	YES	
REINDEX	YES	
RELEASE SAVEPOINT	YES	
RESET	YES	
REVOKE	YES	
ROLLBACK	YES	
ROLLBACK PREPARED	NO	
ROLLBACK TO SAVEPOINT	YES	
SAVEPOINT	YES	

Table 3.2 SQL Support in Greenplum Database

SQL Command	Supported in Greenplum	Modifications, Limitations, Exceptions
SELECT	YES	<p>Limitations:</p> <ul style="list-style-type: none"> Limited use of VOLATILE and STABLE functions in FROM or WHERE clauses Limited use of correlated subquery expressions (See SELECT) Text search (Tsearch2) is not supported FETCH FIRST or FETCH NEXT clauses not supported <p>Greenplum Database Clauses (OLAP): [GROUP BY <i>grouping_element</i> [, ...]] [WINDOW <i>window_name</i> AS (<i>window_specification</i>)] [FILTER (WHERE <i>condition</i>)] applied to an aggregate function in the SELECT list</p>
SELECT INTO	YES	See SELECT
SET	YES	
SET CONSTRAINTS	NO	In PostgreSQL, this only applies to foreign key constraints, which are currently not enforced in Greenplum Database.
SET ROLE	YES	
SET SESSION AUTHORIZATION	YES	Deprecated as of PostgreSQL 8.1 - see SET ROLE
SET TRANSACTION	YES	
SHOW	YES	
START TRANSACTION	YES	
TRUNCATE	YES	
UNLISTEN	NO	
UPDATE	YES	<p>Unsupported Clauses: RETURNING</p> <p>Limitations:</p> <ul style="list-style-type: none"> SET not allowed for Greenplum distribution key columns. Joins must be on a common Greenplum distribution key (equijoins). Cannot use STABLE or VOLATILE functions in an UPDATE statement if mirrors are enabled.
VACUUM	YES	<p>Limitations: VACUUM FULL is not recommended in Greenplum Database.</p>
VALUES	YES	

4. About Greenplum Query Processing

Users issue queries to Greenplum Database just as they would to any database management system (DBMS). They connect to the database instance on the Greenplum master host using a client application (such as `psql`) and submit an SQL statement.

The query is received through the master, which parses the query, optimizes the query, and creates a parallel query plan. The master then dispatches the plan to all of the segments for execution. Each segment is then responsible for executing local database operations on its own particular set of data.

All database operations—such as table scans, joins, aggregations, and sorts—execute in parallel across the segments simultaneously. Each operation is performed on a segment database independent of the data associated with the other segment databases.

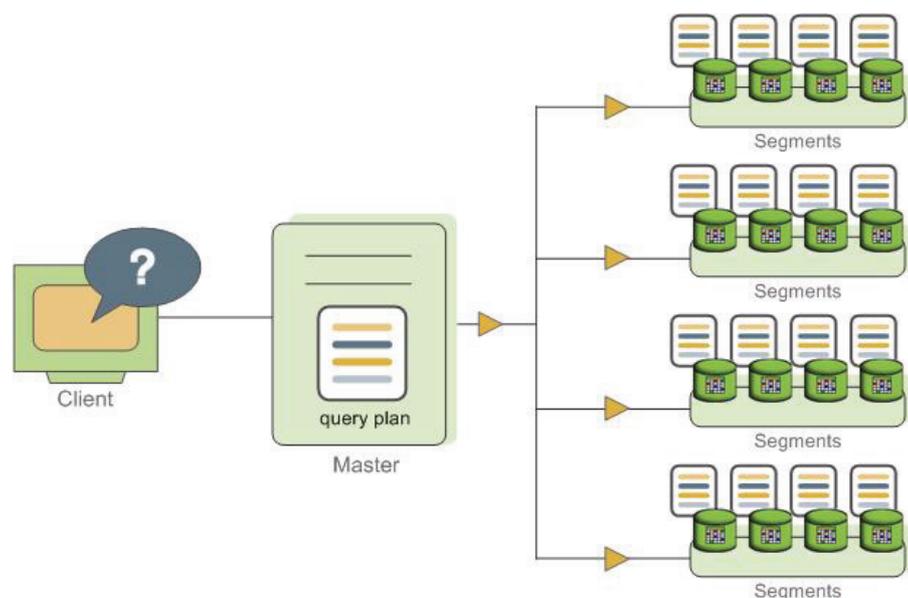


Figure 4.1 Dispatching the Parallel Query Plan

Understanding Greenplum Query Plans

A query plan is the set of operations that Greenplum Database will perform to produce the answer to a given query. Each *node* or step in the plan represents a database operation such as a table scan, join, aggregation or sort. Plans are read and executed from bottom to top.

Understanding Parallel Query Execution

Greenplum creates a number of database processes to handle the work of a query. On the master, the query worker process is called the *query dispatcher* (QD). The QD is responsible for creating and dispatching the query plan, and for accumulating and presenting the final results. On the segments, a query worker process is called a *query executor* (QE). A QE is responsible for completing its portion of work and communicating its intermediate results to the other worker processes.

For each *slice* of the query plan there is at least one worker process assigned. A worker process works on its assigned portion of the query plan independently. During query execution, each segment will have a number of processes working on the query in parallel.

Related processes that are working on the same portion of the query plan are referred to as *gangs*. As a portion of work is completed, tuples flow up the query plan from one gang of processes to the next. This inter-process communication between the segments is what is referred to as the *interconnect* component of Greenplum Database.

Figure 4.3 shows the query worker processes on the master and two segment instances for the query plan illustrated in Figure 4.2.

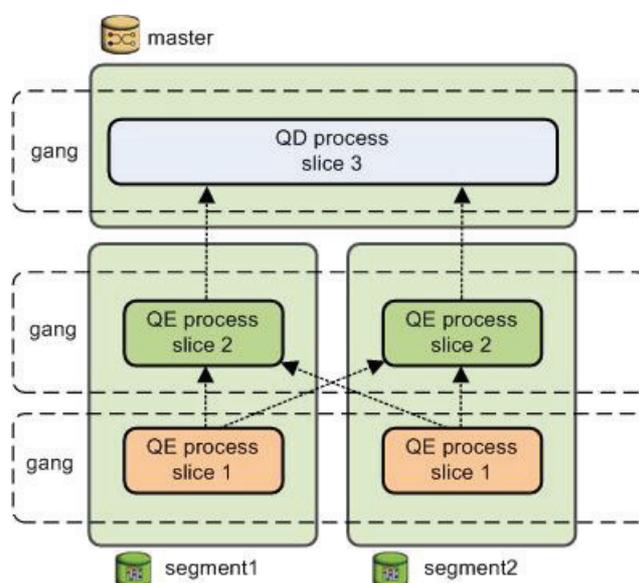


Figure 4.3 Query Worker Processes

Section II: Installation and Initialization

This section describes the tasks involved in getting a Greenplum Database system up and running:

- [Preparing for a Greenplum Installation](#) - Information about how hardware is typically configured for Greenplum Database, and instructions on validating your baseline hardware performance.
- [Installing the Greenplum Software](#) - Instructions on installing and configuring the Greenplum software on all hosts in your Greenplum Database array.
- [Configuring Localization Settings](#) - Information on the localization features of Greenplum Database. Locale settings must be configured prior to initializing your Greenplum Database system.
- [Initializing a Greenplum Database System](#) - Instructions for initializing a Greenplum Database system.
- [Greenplum Database Demo Programs](#) - Instructions on how to initialize a virtual Greenplum Database array with the master and segment instances all on one host. This configuration is for demonstration purposes only.

5. Preparing for a Greenplum Installation

To achieve the best performance from your Greenplum Database system, it is important to have servers that can support the storage and processing demands of the Greenplum Database software.

This chapter contains some recommendations for configuring your hardware to achieve the best performance possible with Greenplum Database. It contains the following topics:

- [About Greenplum and Hardware Platforms](#)
- [Required OS System Settings for Greenplum](#)
- [Estimating Storage Capacity](#)
- [Validating Hardware Performance](#)

About Greenplum and Hardware Platforms

Greenplum Database is a software-only solution, meaning that it runs on a variety of commodity server platforms from hardware vendors such as Sun, Dell and HP. The hardware and database software are not coupled as with some other data warehouse appliance vendors. However, as with any database, Greenplum's performance is dependent on the hardware on which it is installed. And because the database is distributed across multiple machines in a Greenplum Database system, the selection and configuration of hardware is even more important to achieving the best performance possible.

Greenplum is in the process of certifying a number of standard hardware platforms, and can provide guidance on selecting and configuring hardware to meet your database performance needs. Greenplum recommends that you work with our platform engineers prior to making a hardware purchase decision for Greenplum Database.

Example Segment Host Hardware Stack

Regardless of the hardware platform you choose, a production Greenplum Database processing node (a segment host) is typically configured as described in this section.

The segment hosts do the majority of database processing, so the segment host servers are configured in order to achieve the best performance possible from your Greenplum Database system. Greenplum Database's performance will be as fast as the slowest segment server in the array. Therefore, it is important to ensure that the underlying hardware and operating systems that are running Greenplum Database are all running at their optimal performance level. It is also advised that all segment hosts in a Greenplum Database array have identical hardware resources and configurations.

Segment hosts should also be dedicated to Greenplum Database operations only. To get the best query performance, you do not want Greenplum Database competing with other applications for machine or network resources.

The following diagram shows an example Greenplum Database segment host hardware stack. The number of effective CPUs on a host is the basis for determining how many primary Greenplum Database segment instances to deploy per segment host. This example shows a host with two effective CPUs (one dual-core CPU). Note that there is one primary segment instance (or primary/mirror pair if using mirroring) per CPU core.

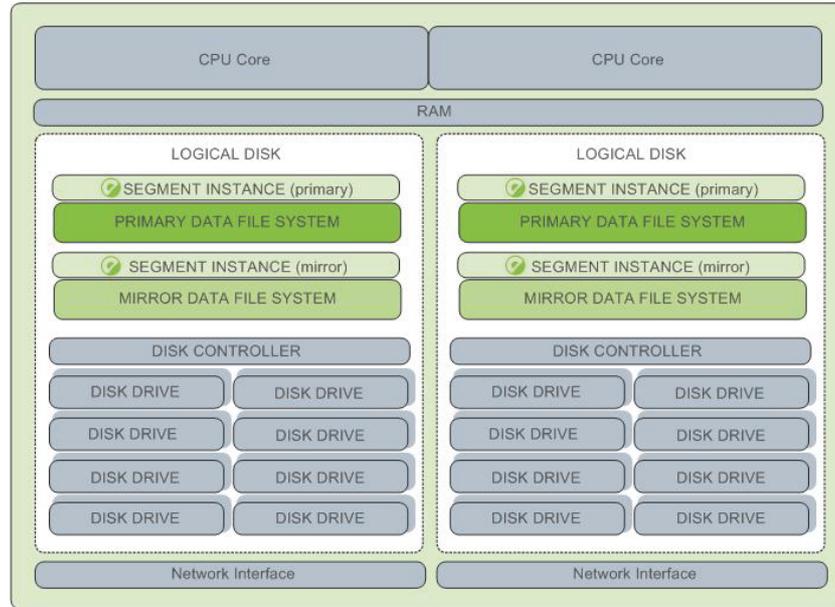


Figure 5.1 Example Greenplum Database Segment Host Configuration

Disk Layout

Each CPU should be mapped to a logical disk. A logical disk consists of one primary file system (and optionally a mirror file system) accessing a pool of physical disks through an I/O channel or disk controller. The logical disk and file system are provided by the operating system. Most operating systems provide the ability for a logical disk drive to use groups of physical disks arranged in RAID arrays. Greenplum Database performs best with a RAID-10 (mirrored) disk configuration, but also offers acceptable performance using RAID-5 (or RAID-Z). Whether you choose a

performance disk configuration (RAID-10) or a capacity disk configuration (RAID-5) configuration depends on your fault tolerance, performance, and capacity requirements.

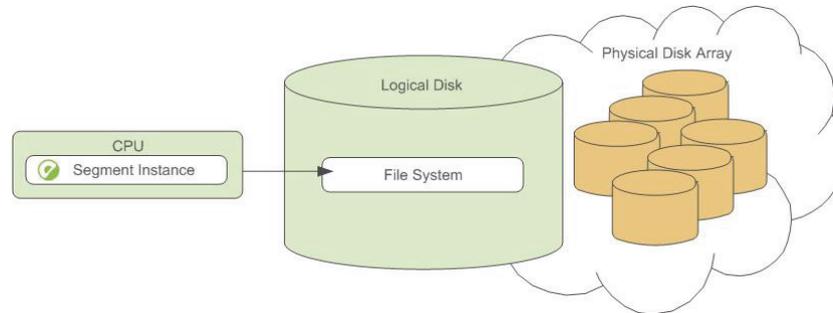


Figure 5.2 Logical Disk Layout in Greenplum Database

Network Layout

For the best possible performance, Greenplum recommends having one network interface for each primary segment instance on a segment host. If using mirroring, a primary/mirror pair would then share an interface. The master host would also have four network interfaces to the Greenplum Database array plus additional external network interfaces.

On each Greenplum Database segment host, you would then create separate host names for each network interface. For example, if a host has four network interfaces, then it would have four corresponding host names, each of which will map to a primary segment instance. You would also do the same for the master host, however, when you initialize your Greenplum Database array, only one master host name will be used within the array.

With this configuration, the TCP/IP stack within the operating system automatically selects the best path to the destination. Greenplum Database automatically balances the network destinations to maximize parallelism.

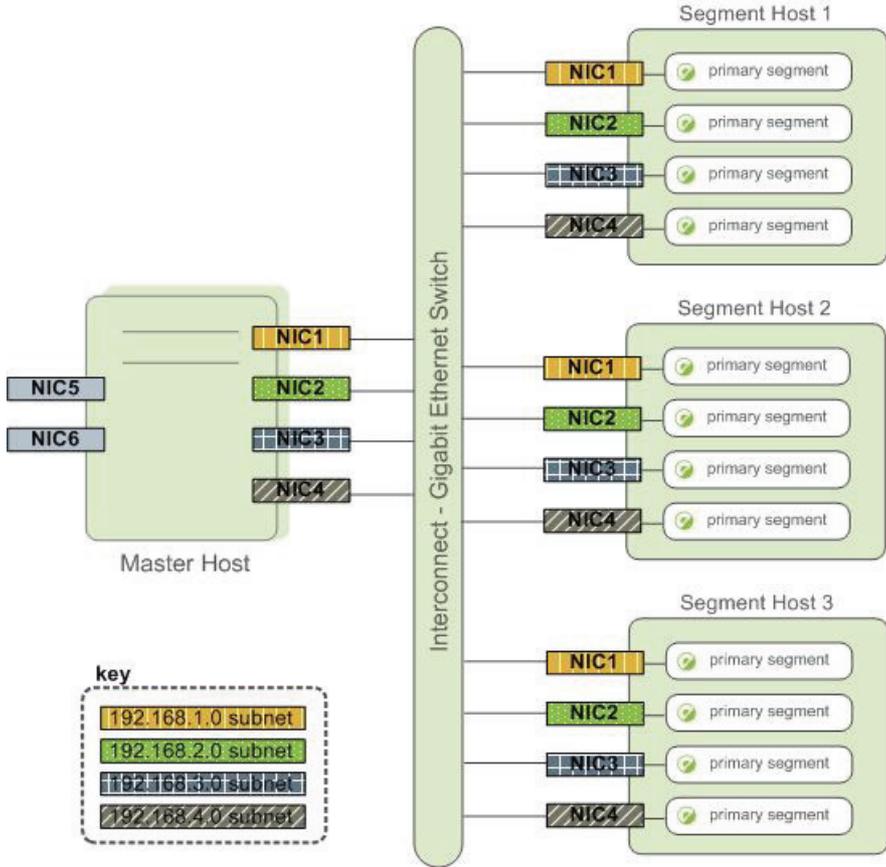


Figure 5.3 Example Network Interface Architecture

Redundant Switch Configuration

If using multiple Gigabit Ethernet switches within your Greenplum Database array, you should evenly divide the number of subnets between each switch. In our example configuration, if we had two switches, NICs 1 and 2 on each host would use switch 1 and NICs 3 and 4 on each host would use switch 2. For the master host, the host name bound to NIC 1 (and therefore using switch 1) is the effective master host name for the

array. Therefore, if deploying a warm standby master for redundancy purposes, the standby master should map to a NIC that uses a different switch than the primary master.

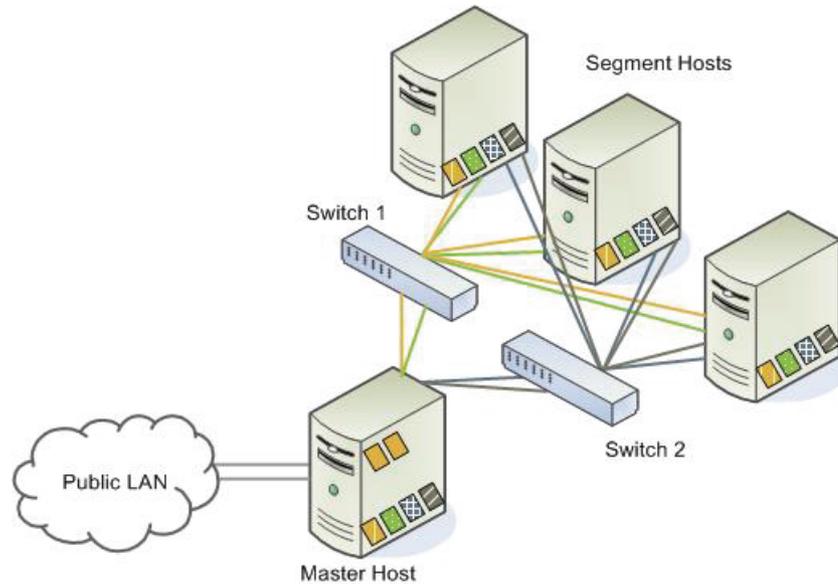


Figure 5.4 Example Switch Configuration

Required OS System Settings for Greenplum

A typical Greenplum Database installation can quickly exhaust certain operating system resource limits on a host. On most systems, the factory defaults are too low to support the processing needs of a database application such as Greenplum Database. This section provides information on OS tuning parameters, which should be set for each host (master and segments) in your Greenplum Database system. In general, the following categories of system parameters need to be altered:

- **Shared Memory** - A Greenplum Database instance will not work unless the shared memory segment for your kernel is properly sized. Most default OS installations have the shared memory values set too low for Greenplum Database. On Linux systems, you must also disable the OOM (out of memory) killer.
- **Network** - On high-volume Greenplum Database systems, certain network-related tuning parameters must be set to optimize network connections made by the Greenplum interconnect.
- **User Limits** - User limits control the resources available to processes started by a user's shell. Greenplum Database requires a higher limit on the allowed number of file descriptors that a single process can have open. The default settings may cause some Greenplum Database queries to fail because they will run out of file descriptors needed to process the query.

Linux Parameter Settings

Set the following parameters in the `/etc/sysctl.conf` file on each Greenplum host and reboot:

```
kernel.shmmax = 500000000
kernel.shmmni = 4096
kernel.shmall = 4000000000
kernel.sem = 250 64000 100 512
net.ipv4.tcp_tw_recycle=1
net.ipv4.tcp_max_syn_backlog=4096
net.core.netdev_max_backlog=10000
vm.overcommit_memory=2
```

Set the following parameters in the `/etc/security/limits.conf` file on each Greenplum host:

```
* soft nofile 65536
* hard nofile 65536
* soft nproc 131072
* hard nproc 131072
```

Solaris 10 x86 Parameter Settings

Set the following parameters in the `/etc/system` file on each Greenplum host and reboot:

```
set rlim_fd_max=262144
set rlim_fd_cur=65536
set shmsys:shminfo_shmmax=0x2000000
set semsys:seminfo_semmni=1024
```

In the `/etc/rc2.d` directory on each Greenplum host, add the following lines to a script file named `S##gpdb_ndd` where `##` is any two numbers. Make sure to set file permissions on this file so that it is executable by all Greenplum users (`gpadmin`):

```
/usr/sbin/ndd -set /dev/tcp tcp_conn_req_max_q 4096
/usr/sbin/ndd -set /dev/tcp tcp_conn_req_max_q0 4096
/usr/sbin/ndd -set /dev/tcp tcp_largest_anon_port 65535
/usr/sbin/ndd -set /dev/tcp tcp_smallest_anon_port 4096
/usr/sbin/ndd -set /dev/tcp tcp_time_wait_interval 1000
```

Verifying OS Settings

Greenplum provides a utility called `gpcheckos` that can be used to verify that all hosts in your array have the correct OS settings for running the Greenplum Database software. To run `gpcheckos`:

1. Log in on the master host as the user who will be running your Greenplum Database system (for example, `gpadmin`).

2. Create a host file that has the host names of all segment hosts to include in the verification test (one host name per line). Make sure there are no blank lines or extra spaces. If using a multi-NIC configuration, this file would just have a *single* host name per segment host. For example:

```
sdw1-1
sdw2-1
sdw3-1
```

3. Run the `gpcheckos` utility using the host file you just created. For example:


```
$ gpcheckos -f segment_host_file
```
4. Look for lines prefixed with `[FIX username@hostname]`. These lines will explain OS-level fixes that need to be made before you initialize Greenplum Database.
5. Run the `gpcheckos` utility again, this time only checking the master host. For example:


```
$ gpcheckos -h mdw-1
```

Estimating Storage Capacity

To estimate how much data your Greenplum Database system can accommodate, use the following measurements as guidelines. Also keep in mind that you may want to have extra space for landing backup files and data load files on each segment host.

- [Calculating Usable Disk Capacity](#)
- [Calculating User Data Size](#)
- [Calculating Space Requirements for Metadata and Logs](#)

Calculating Usable Disk Capacity

To calculate how much data a Greenplum system can hold, you have to calculate the usable disk capacity per segment host and then multiply that by the number of segment hosts in your Greenplum array. Start with the raw capacity of the physical disks on a segment host that are available for data storage (*raw_capacity*). Account for file system formatting overhead (roughly 10 percent) and the RAID level you are using. If using RAID-10 (the default configuration), the calculation would be:

$$(raw_capacity * 0.9) / 2 = formatted_disk_space$$

Once you have formatted RAID disk arrays (*formatted_disk_space*), you will need to calculate how much storage is actually available for user data (*U*). If using Greenplum mirrors for data redundancy, this would then double the size of your user data ($2 * U$). Greenplum also requires some space be reserved as a working area for active queries. The work space should be approximately one third the size of your user data ($work\ space = U/3$):

$$\text{With mirrors: } (2 * U) + U/3 = formatted_disk_space$$

$$\text{Without mirrors: } U + U/3 = formatted_disk_space$$

Calculating User Data Size

As with all databases, the size of your raw data will be slightly larger once it is loaded into the database. On average, raw data will be about 1.4 times larger on disk after it is loaded into the database, but could be smaller or larger depending on the data types you are using, table storage type, and so on.

- **Page Overhead** - When your data is loaded into Greenplum Database, it is divided into pages of 32KB each. Each page has 20 bytes of page overhead.
- **Row Overhead** - In a regular ‘heap’ storage table, each row of data has 24 bytes of row overhead. An ‘append-only’ storage table has only 4 bytes of row overhead.
- **Attribute Overhead** - For the data values itself, the size associated with each attribute value is dependent upon the data type chosen. As a general rule, you want to use the smallest data type possible to store your data (assuming you know the possible values a column will have).
- **Indexes** - In Greenplum Database, indexes are distributed across the segment hosts as is table data. The default index type in Greenplum Database is B-tree. Because index size depends on the number of unique values in the index and the data to be inserted, precalculating the exact size of an index is impossible. However, you can roughly estimate the size of an index using these formulas.

B-tree: $unique_values * (data_type_size + 24 \text{ bytes})$

Bitmap: $(unique_values * number_of_rows * 1 \text{ bit} * compression_ratio / 8) + (unique_values * 32)$

Calculating Space Requirements for Metadata and Logs

On each segment host, you will also want to account for space for Greenplum Database log files and metadata:

- **System Metadata** — For each Greenplum Database segment instance (primary or mirror) or master instance running on a host, estimate approximately 20 MB for the system catalogs and metadata.
- **Write Ahead Log** — For each Greenplum Database segment (primary or mirror) or master instance running on a host, allocate space for the write ahead log (WAL). The WAL is divided into segment files of 64 MB each. At most, the number of WAL files will be: $2 * checkpoint_segments + 1$. You can use this to estimate space requirements for WAL. The default *checkpoint_segments* setting for a Greenplum Database instance is 8, meaning 1088 MB WAL space allocated for each segment or master instance on a host.
- **Greenplum Database Log Files** — Each segment instance and the master instance generates database log files, which will grow over time. Sufficient space should be allocated for these log files, and some type of log rotation facility should be used to ensure that log files do not grow too large.

Validating Hardware Performance

This section provides instructions for testing the performance of your baseline hardware:

- [Validating Disk I/O and Memory Bandwidth](#)
- [Validating Network Performance](#)

Validating Disk I/O and Memory Bandwidth

Greenplum provides a management utility called `gpcheckperf`, which starts a session on each specified host and runs the following performance tests:

- **Disk I/O Test (`dd` test)** — To test the sequential throughput performance of a logical disk or file system, the utility uses the `dd` command, which is a standard UNIX utility. It times how long it takes to write and read a large file to and from disk and calculates your disk I/O performance in megabytes (MB) per second. By default, the file size that is used for the test is calculated at two times the total RAM on the host. This ensures that the test is truly testing disk I/O and not using the memory cache.
- **Memory Bandwidth Test (`stream`)** — To test memory bandwidth, the utility uses the STREAM benchmark program to measure sustainable memory bandwidth (in MB/s). This tests that your system is not limited in performance by the memory bandwidth of the system in relation to the computational performance of the CPU. In applications where the data set is large (as in Greenplum Database), low memory bandwidth is a major performance issue. If memory bandwidth is significantly lower than the theoretical bandwidth of the CPU, then it can cause the CPU to spend significant amounts of time waiting for data to arrive from system memory.

Before using `gpcheckperf`, you must have a trusted host setup between the hosts involved in the performance test. You can use the utility `gpssh-exkeys` to update the known host files and exchange public keys between hosts if you have not done so already.

When using `gpcheckperf` in a network divided into subnets, you must test each subnet separately. For example, if the network has four subnets and four network interfaces per host, run `gpcheckperf` four times with four separate host files.

Note that `gpcheckperf` calls to `gpssh` and `gpscp`, so these Greenplum utilities must be in your `$PATH`.

To run `gpcheckperf`

1. Create a host file that has the host names of all hosts to include in the performance test (one host name per line). Make sure there are no blank lines or extra spaces. For example:

```
sdw-1
sdw-2
sdw-3
```

2. Run the `gpcheckperf` utility using the host file you just created. Use the `-d` option to specify the file systems you want to test on each host (you must have write access to these directories). For example:

```
$ gpcheckperf -f seg_host_file -d /data1 -d /data2 -v
```

3. The utility may take a while to perform the tests as it is copying very large files between the hosts. When it is finished you will see the summary results for the Disk Write, Disk Read, and Stream tests.

If your network is divided into subnets, repeat this procedure with a separate host file for each subnet.

Validating Network Performance

Greenplum provides a utility called `gpchecknet` that tests network performance between the hosts in your Greenplum Database array. It runs the `netperf` benchmark program `TCP_STREAM` test, which transfers a 15 second stream of data between the hosts included in the test. By default, pairs of hosts are tested in parallel and the minimum, maximum, average and median network transfer rates are reported in megabytes (MB) per second. If this summary network transfer rate is slower than expected (less than 100 MB/s), you can run the network test serially using the `-r n` option to obtain per-host results. Serial mode is also recommended if you are testing an uneven number of hosts.

To run `gpchecknet`

1. Create a host file that has the host names of all hosts to include in the performance test (one host name per line). Make sure there are no blank lines or extra spaces.

For example:

```
mdw-1
sdw-1
sdw-2
sdw-3
```

2. Run the `gpchecknet` utility using the host file you just created. Use the `-d` option to specify a temporary test directory to copy the `netperf` program files (you must have write access to this directory). For example:

```
$ gpchecknet -f host_file -d /data1
```

3. The minimum, maximum, average and median network transfer rates are reported in megabytes (MB) per second. If this summary network transfer rate is slower than expected (less than 100 MB/s), you can rerun the network test in serial mode to determine which host(s) have a problem:

```
$ gpchecknet -f host_file -r n -d /data1
```

6. Installing the Greenplum Software

This chapter describes how to install the Greenplum Database software on all of the hosts that will comprise your Greenplum Database system. It contains the following topics:

- [Overview](#)
- [Installing Greenplum Database on the Master Host](#)
- [Configuring Your Installation on the Master Host](#)
- [Installing Greenplum Database on the Segment Hosts](#)
- [Next Steps](#)

Overview

Installing Greenplum Database involves installing and configuring the software on the master host first, then using utilities in your master installation to install and configure the segment hosts. After the Greenplum software is installed and configured on all hosts, you are then ready to initialize your Greenplum Database system.

For more information about hardware and OS configuration for a Greenplum Database host, see [Chapter 5, “Preparing for a Greenplum Installation”](#).

Installing Greenplum Database on the Master Host

This section explains how to install Greenplum Database on the master host. The master host is the machine by which clients and users access a Greenplum Database system. Setting up your Greenplum Database master involves the following tasks:

- [Running the Installer](#)
- [Designating a Greenplum User](#)
- [Configuring Your Environment Variables](#)
- [Creating the Data Directory on the Master Host](#)
- [Setting Up a Trusted Host Environment](#)

Installing a Standby Master Host

If you decide to also deploy a standby master, you should also perform the above tasks on the standby master host as well. Your primary and standby master hosts should be installed before initializing your Greenplum Database system. For more information about the standby master, see [“Enabling High Availability Features”](#) on page 183.

Running the Installer

1. Download or copy the installer file to the machine that will be the Greenplum Database master host. Installer files are available from Greenplum for RedHat (32-bit and 64-bit), Solaris 64-bit and SuSe Linux 64-bit platforms.
2. Unzip the installer file where *PLATFORM* is either `RHEL4-i386` (RedHat 32-bit), `RHEL4-x86_64` (RedHat 64-bit), `SOL-x86_64` (Solaris 64-bit) or `SuSE10-x86_64` (SuSe Linux 64 bit). For example:


```
# unzip greenplum-db-3.3.7.x-PLATFORM.zip
```
3. Launch the installer using `bash`. For example:


```
# /bin/bash greenplum-db-3.3.7.x-PLATFORM.bin
```
4. The installer will prompt you to accept the Greenplum Database license agreement. Type `yes` to accept the license agreement.
5. The installer will prompt you to provide an installation path. Press `ENTER` to accept the default install path (`/usr/local/greenplum-db-3.3.7.x`), or enter an absolute path to an install location. You must have write permissions to the location you specify.
6. The installer will install the Greenplum software and create a `greenplum-db` symbolic link one directory level above your version-specific Greenplum installation directory. The symbolic link is used to facilitate patch maintenance and upgrades between versions. The installed location is referred to as `$GPHOME`.
7. After installation, proceed to the instructions for “[Configuring Your Installation on the Master Host](#)” on page 42.

About Your Greenplum Database Installation

- `greenplum_path.sh` — This file contains the environment variables for Greenplum Database. See “[Configuring Your Environment Variables](#)” on page 43.
- `bin` — This directory contains the Greenplum Database management utilities. This directory also contains the PostgreSQL client and server programs, most of which are also used in Greenplum Database.
- `demo` — This directory contains the Greenplum demonstration programs.
- `docs` — The Greenplum Database documentation (PDF files).
- `ext` — Bundled programs (such as Python) used by some Greenplum Database utilities.
- `include` — The C header files for Greenplum Database.
- `lib` — Shared library files for Greenplum Database.
- `share` — Shared files for Greenplum Database.

Configuring Your Installation on the Master Host

After you have installed the Greenplum Database software on the master host, you must perform the following configurations:

- [Setting OS Tuning Parameters](#)
- [Designating a Greenplum User](#)
- [Designating a Greenplum Group \(optional\)](#)
- [Changing Ownership of Your Greenplum Installation](#)
- [Configuring Your Environment Variables](#)
- [Creating the Data Directory on the Master Host](#)

Setting OS Tuning Parameters

A typical Greenplum Database installation can quickly exhaust certain operating system resource limits on a host. On most systems, the factory defaults are too low to support the processing needs of a database application such as Greenplum. See [“Required OS System Settings for Greenplum”](#) on page 34 for the recommended OS parameter settings for your operating system. You must set the recommended OS parameters for both your primary and standby master host (if you choose to deploy a standby master).

Designating a Greenplum User

You cannot run a Greenplum Database database server as `root`. Greenplum recommends that you designate a user account that will own your Greenplum Database installation, and to always start and administer Greenplum Database as this user. For the purposes of this documentation, we will use the user name of `gpadmin`. You can choose any user name you like, but be sure to use the same user name consistently on all hosts in your Greenplum Database system.

To add a new user, for example, run the following commands as `root`:

```
# useradd gpadmin
# passwd gpadmin
# New password: <gpadmin_password>
# Retype new password: <gpadmin_password>
```

Designating a Greenplum Group (optional)

If your environment has multiple administrative users or if you will be running multiple Greenplum Database instances on the same array of machines, you may want to designate a group of users to own your Greenplum Database installation. For the purposes of this documentation, we will use the group name of `gpadmin` as well.

To add a new group, for example, run the following commands as `root`:

```
# groupadd gpadmin
# usermod -g gpadmin gp_user1
```

```
# usermod -g gpadmin gp_user2
```

Changing Ownership of Your Greenplum Installation

After creating your `gpadmin` user or group account, Greenplum recommends that you change the ownership of your Greenplum Database installation so that it is owned by the `gpadmin` user or group. For example, run the following command as `root`:

```
# chown -R gpadmin /usr/local/greenplum-db
# chgrp -R gpadmin /usr/local/greenplum-db
```

Configuring Your Environment Variables

As a convenience, a `greenplum_path.sh` file is provided in your `$GPHOME` directory following installation with environment variable settings for Greenplum Database. You can source this in the `gpadmin` user's startup shell profile (such as `.bashrc`), or in `/etc/profile` if you want to set the environment variables for all users.

For example, you could add a line similar to the following to your chosen profile files:

```
source /usr/local/greenplum-db/greenplum_path.sh
```

After editing the chosen profile file, source it as the correct user to make the changes active. For example:

```
$ source ~/.bashrc
```

or

```
$ source /etc/profile
```

Creating the Data Directory on the Master Host

Every Greenplum Database master and segment instance has a designated storage area on disk that is called the *data directory* location. This is the file system location where the database data will be stored. Each master and segment instance needs its own designated data directory storage location.

The data directory location on the master is different than those on the segments. The master does not store any user data, only the system catalog tables and system metadata are stored on the master instance, therefore you do not need to designate as much storage space as on the segments.

To create the data directory location on the master

1. Create or choose a directory that will serve as your master data storage area. This directory should have sufficient disk space for your data and be owned by the `gpadmin` user and group. For example, run the following commands as `root`:

```
# mkdir /data1/gpdb_p1
```

2. Change ownership of this directory to the `gpadmin` user and group. For example:

```
# chown gpadmin /data1/gpdb_p1
# chgrp gpadmin /data1/gpdb_p1
```

Installing Greenplum Database on the Segment Hosts

Once you have your master host installed and configured, you must then install the Greenplum Database DBMS software on each of the segment hosts. This involves the following tasks:

- [Setting OS Tuning Parameters](#)
- [Setting Up a Trusted Host Environment](#)
- [Copying the Greenplum Software to the Segment Hosts](#)
- [Creating the Data Storage Areas on the Segment Hosts](#)
- [Synchronizing System Clocks](#)

Setting OS Tuning Parameters

A typical Greenplum Database installation can quickly exhaust certain operating system resource limits on a host. On most systems, the factory defaults are too low to support the processing needs of a database application such as Greenplum. See [“Required OS System Settings for Greenplum”](#) on page 34 for the recommended OS parameter settings for your operating system. You must set the recommended OS tuning parameters on all segment hosts.

Setting Up a Trusted Host Environment

The Greenplum Database management scripts require that the same non-root user (`gpadmin`) be created on all hosts in the Greenplum Database system. The scripts must be able to connect as that user to all segment hosts without a password prompt.

Identify all of the segment hosts in Greenplum Database. For each host, you must:

1. Exchange SSH keys as `root`.
2. Create the `gpadmin` user account on each segment host (if it doesn't exist).
3. Exchange SSH keys as `gpadmin`.

Note: If using a multi-NIC configuration, make sure to exchange SSH keys using all of the configured host names for a segment host.

Exchanging SSH Keys Between Master and All Segment Hosts

Greenplum provides a utility called `gpssh-exkeys` that can facilitate the exchange of SSH keys between all hosts in your Greenplum Database array.

Greenplum recommends performing the key exchange process twice: once as `root` (for administration convenience) and once as the `gpadmin` user (required for the Greenplum management utilities). Perform the following tasks in this order:

- [“To exchange SSH keys as root”](#) on page 45
- [“To create the gpadmin user”](#) on page 45
- [“To exchange SSH keys as the gpadmin user”](#) on page 46

To exchange SSH keys as root

1. Create a host list file that has all of the host names in your Greenplum Database array (master, standby master and segment hosts) — one host name per line. If using a multi-NIC configuration, make sure to exchange SSH keys using all of the configured host names for a given host. Make sure there are no blank lines or extra spaces. For example:

```

mdw                OR      masterhost
sdw1-1             seghost1
sdw1-2             seghost2
sdw1-3             seghost3
sdw1-4
sdw2-1
sdw2-2
sdw2-3
sdw2-4
sdw3-1
sdw3-2
sdw3-3
sdw3-4

```

2. Log in as `root` on the master host, and source the `greenplum_path.sh` file from your Greenplum installation.

```

$ su -
# source /usr/local/greenplum-db/greenplum_path.sh

```

3. Run the `gpssh-exkeys` utility referencing the host list file (`all_hosts_file`) you just created. For example:

```

# gpssh-exkeys -f /home/gpadmin/all_hosts_file

```

4. `gpssh-exkeys` will check the remote hosts and perform the key exchange between all hosts. Enter the `root` user password when prompted. For example:

```

***Enter password for root@hostname: <root_password>

```

To create the gpadmin user

1. Create a second host list file that just has the host names of the segment hosts. If using a multi-NIC configuration, this file would just have a *single* host name per segment host — one host name per line. We will call this host list file `single_seg_hosts_file`.

2. Use `gpssh` to create the `gpadmin` user on all of the segment hosts (if it does not exist already). For example:

```

# gpssh -f single_seg_hosts_file '/usr/sbin/useradd gpadmin
-d /home/gpadmin -s /bin/bash'

```

3. Set the new `gpadmin` user's password. On Linux, you can do this on all segment hosts at once using `gpssh`. For example:

```
# gpssh -f single_seg_hosts_file 'echo gpadmin_password |
passwd gpadmin --stdin'
```

On Solaris, you must log in to each segment host and set the gpadmin user's password on each host. For example:

```
# ssh segment_hostname
# passwd gpadmin
# New password: <gpadmin_password>
# Retype new password: <gpadmin_password>
```

4. Verify that the gpadmin user has been created by looking for its home directory:

```
# gpssh -f single_seg_hosts_file ls -l /home
```

To exchange SSH keys as the gpadmin user

1. Log in as gpadmin, and run the `gpssh-exkeys` utility referencing the all host list file (`all_hosts_file`) you created earlier:

```
$ su - gpadmin
$ gpssh-exkeys -f /home/gpadmin/all_hosts_file
```

2. `gpssh-exkeys` will check the remote hosts and perform the key exchange between all hosts. Enter the gpadmin user password when prompted. For example:

```
***Enter password for gpadmin@hostname: <gpadmin_password>
```

Copying the Greenplum Software to the Segment Hosts

Now that your trusted host environment is established, you can use `gpscp` and `gpssh` from the master host to install the Greenplum Database software on all of the segment hosts at the same time.

NOTE: The Greenplum Database software must be installed in the *same location* on all hosts (the master host and all segment hosts must have the same `$GPHOME`).

To install Greenplum Database on the Segment Hosts

1. On the master host, create a tar file of your Greenplum Database installation. For example (running as `root`):

```
# su -
# cd /usr/local
# gtar -cvf /home/gpadmin/gp.tar greenplum-db-3.3.7.x
```

2. Copy the tar file to the segment hosts using `gpscp`. For example:

```
# source /usr/local/greenplum-db/greenplum_path.sh
# gpscp -f /home/gpadmin/single_seg_hosts_file
/home/gpadmin/gp.tar =:/usr/local
```

3. Start an interactive session in `gpssh`. For example:

```
# gpssh -f /home/gpadmin/single_seg_hosts_file
```

4. At the `gpssh` command prompt, untar the tar file in the installation directory on the segment hosts. For example:


```
=> gtar --directory /usr/local -xvf /usr/local/gp.tar
```
5. Confirm that the Greenplum Database directory was installed in the correct location (the same location as `$GPHOME` on your master host). For example:


```
=> ls /usr/local/greenplum-db-3.3.7.x
```
6. Create a `greenplum-db` symbolic link to point to the current version directory of your Greenplum Database software. For example:


```
=> ln -s /usr/local/greenplum-db-3.3.7.x
    /usr/local/greenplum-db
```
7. Change the ownership of the Greenplum Database install directory to the `gpadmin` user or group. For example:


```
=> chown -R gpadmin /usr/local/greenplum-db
    => chgrp -R gpadmin /usr/local/greenplum-db
```
8. Remove the tar file. For example:


```
=> rm /usr/local/gp.tar
```
9. Move on to the next task, “[Creating the Data Storage Areas on the Segment Hosts](#)” on page 47. Do not exit the `gpssh` interactive session.

Creating the Data Storage Areas on the Segment Hosts

Every Greenplum Database segment instance has a designated storage area on disk that is called the *data directory* location. This is the file system location where the database data will be stored. In Greenplum Database, each *segment instance* is a PostgreSQL database server with its own distinct data directory. A segment host will most likely have more than one segment instance.

The Greenplum Database initialization utility will create a unique data directory per segment instance at runtime. However the utility needs to know where to create it. Therefore, you must create one (or more) data storage areas on disk where the segment data directories will reside.

If deploying mirror segments, then a segment host will have the same number of mirror segment instances as primary segment instances. If you decide to deploy mirror segments, you may want to designate a separate storage area for the mirror data directories.

Keep in mind that the data directories of the segment instances are where the user data resides, so they must have enough disk space to accommodate your planned data capacity.

To create segment data storage areas

1. In `gpssh`, create or choose a directory that will serve as your primary segment storage area. This directory should have sufficient disk space for your data and be owned by the `gpadmin` user. For example (in a `gpssh` interactive session as `root`):

```
=> mkdir /data1
```

2. Change ownership of this directory to the `gpadmin` user or group. For example:

```
=> chown -R gpadmin /data1
```

```
=> chgrp -R gpadmin /data1
```

3. (optional) If deploying mirror segments, create or choose a directory that will serve as your mirror segment storage area. This directory should have sufficient disk space for your data and be owned by the `gpadmin` user or group. For example, run the following commands as `root`:

```
=> mkdir /data2
```

```
=> chown -R gpadmin /data2
```

```
=> chgrp -R gpadmin /data2
```

4. Exit `gpssh` interactive mode:

```
=> exit
```

Synchronizing System Clocks

Greenplum recommends that you synchronize the system clocks on all hosts in the array using NTP (Network Time Protocol) or a similar utility. See www.ntp.org for more information about NTP. To see if the system clocks are synchronized, run the `date` command using `gpssh`. For example:

```
$ gpssh -f single_seg_hosts_file -v date
```

If you have the NTP daemon installed on your segment hosts, you can use it to synchronize the system clocks. For example:

```
$ gpssh -f single_seg_hosts_file -v ntpd
```

Next Steps

After you have installed and verified the Greenplum Database software on all of the hosts in the system, you are ready to initialize Greenplum Database and begin using it. See “[Initializing a Greenplum Database System](#)” on page 57.

Uninstalling Greenplum Database

Greenplum provides the `gpdeletesystem` utility to facilitate the removal of a Greenplum Database instance. This utility stops all `postgres` processes and deletes all data directories on the master and segments.

After running `gpdeletesystem`, you can manually remove Greenplum artifacts such as the `$GPHOME` directories, the `greenplum-db` symbolic link, and the installation tar files.

To uninstall Greenplum Database:

1. Log in as `gpadmin` on the master host.

2. With Greenplum Database running, run `gpdeletesystem` and specify the master data directory for the Greenplum instance to be deleted. For example:

```
$ gpdeletesystem -d /data1/gpdb_p1
```

3. Start an interactive session in `gpssh`. For example:

```
$ gpssh -f /home/gpadmin/single_seg_hosts_file
```

4. At the `gpssh` command prompt, remove the `$GPHOME` directories and symbolic links on the segments. For example:

```
=> rm -rf /usr/local/greenplum-db-3.3.7.x  
=> rm /usr/local/greenplum-db
```

5. After exiting `gpssh`, remove the `$GPHOME` directory and symbolic link on the master. For example:

```
$ rm -rf /usr/local/greenplum-db-3.3.7.x  
$ rm /usr/local/greenplum-db
```

If you have a standby master deployed, log in to the standby master host and remove the `$GPHOME` directory.

6. Remove Greenplum environment variables. For example, if you sourced the `greenplum_path.sh` file in the startup shell profile, remove this line and re-source the profile.
7. Optionally, restore the defaults of any kernel parameters that were adjusted to meet Greenplum Database requirements.

7. Configuring Localization Settings

This chapter describes the available localization features of Greenplum Database. Greenplum Database supports localization with two approaches:

- Using the locale features of the operating system to provide locale-specific collation order, number formatting, and so on.
- Providing a number of different character sets defined in the Greenplum Database server, including multiple-byte character sets, to support storing text in all kinds of languages, and providing character set translation between client and server.

About Locale Support in Greenplum Database

Locale support refers to an application respecting cultural preferences regarding alphabets, sorting, number formatting, etc. Greenplum Database uses the standard ISO C and POSIX locale facilities provided by the server operating system. For additional information refer to the documentation of your operating system.

Locale support is automatically initialized when a Greenplum Database system is initialized. The initialization utility, `gpinitssystem`, will initialize the Greenplum array with the locale setting of its execution environment by default, so if your system is already set to use the locale that you want in your Greenplum Database system then there is nothing else you need to do. If you want to use a different locale (or you are not sure which locale your system is set to), you can instruct `gpinitssystem` exactly which locale to use by specifying the `-n locale` option. For example:

```
gpinitssystem -c gp_init_config -n sv_SE
```

This example sets the locale to Swedish (`sv`) as spoken in Sweden (`SE`). Other possibilities might be `en_US` (U.S. English) and `fr_CA` (French Canadian). If more than one character set can be useful for a locale then the specifications look like this: `cs_CZ.ISO8859-2`. What locales are available under what names on your system depends on what was provided by the operating system vendor and what was installed. On most systems, the command `locale -a` will provide a list of available locales.

Occasionally it is useful to mix rules from several locales, for example use English collation rules but Spanish messages. To support that, a set of locale subcategories exist that control only a certain aspect of the localization rules:

- `LC_COLLATE` — String sort order
- `LC_CTYPE` — Character classification (What is a letter? Its upper-case equivalent?)
- `LC_MESSAGES` — Language of messages
- `LC_MONETARY` — Formatting of currency amounts
- `LC_NUMERIC` — Formatting of numbers
- `LC_TIME` — Formatting of dates and times

If you want the system to behave as if it had no locale support, use the special locale `C` or `POSIX`.

The nature of some locale categories is that their value has to be fixed for the lifetime of a Greenplum Database system. That is, once `gpinitssystem` has run, you cannot change them anymore. `LC_COLLATE` and `LC_CTYPE` are those categories. They affect the sort order of indexes, so they must be kept fixed, or indexes on text columns will become corrupt. Greenplum Database enforces this by recording the values of `LC_COLLATE` and `LC_CTYPE` that are seen by `gpinitssystem`. The server automatically adopts those two values based on the locale that was chosen at initialization time.

The other locale categories can be changed as desired whenever the server is running by setting the server configuration variables that have the same name as the locale categories (see [Appendix D, “Server Configuration Parameters”](#)). The defaults that are chosen by `gpinitssystem` are written into the master and segment `postgresql.conf` configuration files to serve as defaults when the Greenplum Database system is started. If you delete these assignments from the master and each segment `postgresql.conf` files then the server will inherit the settings from its execution environment.

Note that the locale behavior of the server is determined by the environment variables seen by the server, not by the environment of any client. Therefore, be careful to configure the correct locale settings on each Greenplum Database host (master and segments) before starting the system. A consequence of this is that if client and server are set up in different locales, messages may appear in different languages depending on where they originated.

Inheriting the locale from the execution environment means the following on most operating systems: For a given locale category, say the collation, the following environment variables are consulted in this order until one is found to be set: `LC_ALL`, `LC_COLLATE` (the variable corresponding to the respective category), `LANG`. If none of these environment variables are set then the locale defaults to `C`.

Some message localization libraries also look at the environment variable `LANGUAGE` which overrides all other locale settings for the purpose of setting the language of messages. If in doubt, please refer to the documentation of your operating system, in particular the documentation about `gettext`, for more information.

Native language support (NLS), which enables messages to be translated to the user’s preferred language, is not enabled in Greenplum Database for languages other than English. This is independent of the other locale support.

Locale Behavior

The locale settings influence the following SQL features:

- Sort order in queries using `ORDER BY` on textual data
- The ability to use indexes with `LIKE` clauses
- The `upper`, `lower`, and `initcap` functions
- The `to_char` family of functions

The drawback of using locales other than `C` or `POSIX` in Greenplum Database is its performance impact. It slows character handling and prevents ordinary indexes from being used by `LIKE`. For this reason use locales only if you actually need them.

Troubleshooting Locales

If locale support does not work as expected, check that the locale support in your operating system is correctly configured. To check what locales are installed on your system, you may use the command `locale -a` if your operating system provides it.

Check that Greenplum Database is actually using the locale that you think it is.

`LC_COLLATE` and `LC_CTYPE` settings are determined at initialization time and cannot be changed without redoing `gpinitssystem`. Other locale settings including `LC_MESSAGES` and `LC_MONETARY` are initially determined by the operating system environment of the master and/or segment host, but can be changed after initialization by editing the `postgresql.conf` file of each Greenplum master and segment instance. You can check the active locale settings of the master host using the `SHOW` command. Note that every host in your Greenplum Database array should be using identical locale settings.

Character Set Support

The character set support in Greenplum Database allows you to store text in a variety of character sets, including single-byte character sets such as the ISO 8859 series and multiple-byte character sets such as EUC (Extended Unix Code), UTF-8, and Mule internal code. All supported character sets can be used transparently by clients, but a few are not supported for use within the server (that is, as a server-side encoding). The default character set is selected while initializing your Greenplum Database array using `gpinitssystem`. It can be overridden when you create a database, so you can have multiple databases each with a different character set.

Table 7.1 Greenplum Database Character Sets¹

Name	Description	Language	Server?	Bytes/Char	Aliases
BIG5	Big Five	Traditional Chinese	No	1-2	WIN950, Windows950
EUC_CN	Extended UNIX Code-CN	Simplified Chinese	Yes	1-3	
EUC_JP	Extended UNIX Code-JP	Japanese	Yes	1-3	
EUC_KR	Extended UNIX Code-KR	Korean	Yes	1-3	
EUC_TW	Extended UNIX Code-TW	Traditional Chinese, Taiwanese	Yes	1-3	
GB18030	National Standard	Chinese	No	1-2	
GBK	Extended National Standard	Simplified Chinese	No	1-2	WIN936, Windows936
ISO_8859_5	ISO 8859-5, ECMA 113	Latin/Cyrillic	Yes	1	
ISO_8859_6	ISO 8859-6, ECMA 114	Latin/Arabic	Yes	1	

Table 7.1 Greenplum Database Character Sets¹

Name	Description	Language	Server?	Bytes/Char	Aliases
ISO_8859_7	ISO 8859-7, ECMA 118	Latin/Greek	Yes	1	
ISO_8859_8	ISO 8859-8, ECMA 121	Latin/Hebrew	Yes	1	
JOHAB	JOHA	Korean (Hangul)	Yes	1-3	
KOI8	KOI8-R(U)	Cyrillic	Yes	1	KOI8R
LATIN1	ISO 8859-1, ECMA 94	Western European	Yes	1	ISO88591
LATIN2	ISO 8859-2, ECMA 94	Central European	Yes	1	ISO88592
LATIN3	ISO 8859-3, ECMA 94	South European	Yes	1	ISO88593
LATIN4	ISO 8859-4, ECMA 94	North European	Yes	1	ISO88594
LATIN5	ISO 8859-9, ECMA 128	Turkish	Yes	1	ISO88599
LATIN6	ISO 8859-10, ECMA 144	Nordic	Yes	1	ISO885910
LATIN7	ISO 8859-13	Baltic	Yes	1	ISO885913
LATIN8	ISO 8859-14	Celtic	Yes	1	ISO885914
LATIN9	ISO 8859-15	LATIN1 with Euro and accents	Yes	1	ISO885915
LATIN10	ISO 8859-16, ASRO SR 14111	Romanian	Yes	1	ISO885916
MULE_INTERNAL	Mule internal code	Multilingual Emacs	Yes	1-4	
SJIS	Shift JIS	Japanese	No	1-2	Mskanji, ShiftJIS, WIN932, Windows932
SQL_ASCII	unspecified ²	any	Yes	1	
UHC	Unified Hangul Code	Korean	No	1-2	WIN949, Windows949
UTF8	Unicode, 8-bit	all	Yes	1-4	Unicode
WIN866	Windows CP866	Cyrillic	Yes	1	ALT
WIN874	Windows CP874	Thai	Yes	1	
WIN1250	Windows CP1250	Central European	Yes	1	
WIN1251	Windows CP1251	Cyrillic	Yes	1	WIN
WIN1252	Windows CP1252	Western European	Yes	1	
WIN1253	Windows CP1253	Greek	Yes	1	
WIN1254	Windows CP1254	Turkish	Yes	1	
WIN1255	Windows CP1255	Hebrew	Yes	1	
WIN1256	Windows CP1256	Arabic	Yes	1	
WIN1257	Windows CP1257	Baltic	Yes	1	
WIN1258	Windows CP1258	Vietnamese	Yes	1	ABC, TCVN, TCVN5712, VSCII

1. Not all APIs support all the listed character sets. For example, the JDBC driver does not support MULE_INTERNAL, LATIN6, LATIN8, and LATIN10.
2. The SQL_ASCII setting behaves considerably differently from the other settings. The server interprets byte values 0-127 according to the ASCII standard, while byte values 128-255 are taken as uninterpreted characters. No encoding conversion will be done when the setting is SQL_ASCII. Thus, this setting is not so much a declaration that a specific encoding is in use, as a declaration of ignorance about the encoding. If you are working with any non-ASCII data, it is unwise to use the SQL_ASCII setting, because Greenplum Database will be unable to convert or validate non-ASCII characters.

Setting the Character Set

`gpinit` defines the default character set for a Greenplum Database system by reading the setting of the `ENCODING` parameter in the `gp_init_config` file at initialization time. The default character set is `UNICODE`.

You can create a database with a different character set besides what is used as the system-wide default. For example:

```
=> CREATE DATABASE korean WITH ENCODING 'EUC_KR';
```

Important: Although you can specify any encoding you want for a database, it is unwise to choose an encoding that is not what is expected by the locale you have selected. The `LC_COLLATE` and `LC_CTYPE` settings imply a particular encoding, and locale-dependent operations (such as sorting) are likely to misinterpret data that is in an incompatible encoding.

Since these locale settings are frozen by `gpinit`, the apparent flexibility to use different encodings in different databases is more theoretical than real.

One way to use multiple encodings safely is to set the locale to `C` or `POSIX` during initialization time, thus disabling any real locale awareness.

Character Set Conversion Between Server and Client

Greenplum Database supports automatic character set conversion between server and client for certain character set combinations. The conversion information is stored in the master `pg_conversion` system catalog table. Greenplum Database comes with some predefined conversions or you can create a new conversion using the SQL command `CREATE CONVERSION`.

Table 7.2 Client/Server Character Set Conversions

Server Character Set	Available Client Character Sets
BIG5	not supported as a server encoding
EUC_CN	EUC_CN, MULE_INTERNAL, UTF8
EUC_JP	EUC_JP, MULE_INTERNAL, SJIS, UTF8
EUC_KR	EUC_KR, MULE_INTERNAL, UTF8
EUC_TW	EUC_TW, BIG5, MULE_INTERNAL, UTF8
GB18030	not supported as a server encoding
GBK	not supported as a server encoding
ISO_8859_5	ISO_8859_5, KOI8, MULE_INTERNAL, UTF8, WIN866, WIN1251

Table 7.2 Client/Server Character Set Conversions

Server Character Set	Available Client Character Sets
ISO_8859_6	ISO_8859_6, UTF8
ISO_8859_7	ISO_8859_7, UTF8
ISO_8859_8	ISO_8859_8, UTF8
JOHAB	JOHAB, UTF8
KOI8	KOI8, ISO_8859_5, MULE_INTERNAL, UTF8, WIN866, WIN1251
LATIN1	LATIN1, MULE_INTERNAL, UTF8
LATIN2	LATIN2, MULE_INTERNAL, UTF8, WIN1250
LATIN3	LATIN3, MULE_INTERNAL, UTF8
LATIN4	LATIN4, MULE_INTERNAL, UTF8
LATIN5	LATIN5, UTF8
LATIN6	LATIN6, UTF8
LATIN7	LATIN7, UTF8
LATIN8	LATIN8, UTF8
LATIN9	LATIN9, UTF8
LATIN10	LATIN10, UTF8
MULE_INTERNAL	MULE_INTERNAL, BIG5, EUC_CN, EUC_JP, EUC_KR, EUC_TW, ISO_8859_5, KOI8, LATIN1 to LATIN4, SJIS, WIN866, WIN1250, WIN1251
SJIS	not supported as a server encoding
SQL_ASCII	any (no conversion will be performed)
UHC	not supported as a server encoding
UTF8	all supported encodings
WIN866	WIN866
ISO_8859_5	KOI8, MULE_INTERNAL, UTF8, WIN1251
WIN874	WIN874, UTF8
WIN1250	WIN1250, LATIN2, MULE_INTERNAL, UTF8
WIN1251	WIN1251, ISO_8859_5, KOI8, MULE_INTERNAL, UTF8, WIN866
WIN1252	WIN1252, UTF8
WIN1253	WIN1253, UTF8
WIN1254	WIN1254, UTF8
WIN1255	WIN1255, UTF8
WIN1256	WIN1256, UTF8
WIN1257	WIN1257, UTF8
WIN1258	WIN1258, UTF8

To enable automatic character set conversion, you have to tell Greenplum Database the character set (encoding) you would like to use in the client. There are several ways to accomplish this:

- Using the `\encoding` command in `psql`, which allows you to change client encoding on the fly.
- Using `SET client_encoding TO`. Setting the client encoding can be done with this SQL command:

```
=> SET CLIENT_ENCODING TO 'value';
```

 To query the current client encoding:

```
=> SHOW client_encoding;
```

 To return to the default encoding:

```
=> RESET client_encoding;
```
- Using the `PGCLIENTENCODING` environment variable. When `PGCLIENTENCODING` is defined in the client's environment, that client encoding is automatically selected when a connection to the server is made. (This can subsequently be overridden using any of the other methods mentioned above.)
- Setting the configuration parameter `client_encoding`. If `client_encoding` is set in the master `postgresql.conf` file, that client encoding is automatically selected when a connection to Greenplum Database is made. (This can subsequently be overridden using any of the other methods mentioned above.)

If the conversion of a particular character is not possible — suppose you chose `EUC_JP` for the server and `LATIN1` for the client, then some Japanese characters do not have a representation in `LATIN1` — then an error is reported.

If the client character set is defined as `SQL_ASCII`, encoding conversion is disabled, regardless of the server's character set. Just as for the server, use of `SQL_ASCII` is unwise unless you are working with all-ASCII data.

8. Initializing a Greenplum Database System

This chapter describes how to initialize a Greenplum Database database system. The instructions in this chapter assume you have already installed the Greenplum Database software on all of the hosts in the system according to the instructions in [Chapter 6](#), “Installing the Greenplum Software”.

This chapter contains the following topics:

- [Overview](#)
- [Initializing Greenplum Database](#)
- [Next Steps](#)

Overview

Because a Greenplum Database database is distributed across many machines, the process for initializing the database is different than in PostgreSQL. With a regular PostgreSQL DBMS, you run a utility called `initdb` which creates the data storage directories, generates the shared catalog tables and configuration files, and creates the `template1` database, which is the template used to create other databases.

In a Greenplum Database DBMS, each database instance (the master and all segments) must be initialized across all of the hosts in the system in such a way that they can all work together as a unified DBMS. Greenplum provides its own version of `initdb` called `gpinitssystem`, which takes care of initializing the database on the master and on each segment instance, and starting each instance in the correct order.

After the Greenplum Database database system has been initialized and started, you can then create and manage databases as you would in a regular PostgreSQL DBMS by connecting to the Greenplum master.

Initializing Greenplum Database

These are the high-level tasks for initializing Greenplum Database:

1. Make sure you have completed all of the installation tasks described in [Chapter 6](#), “Installing the Greenplum Software”.
2. (optional) If you want a standby master host, make sure that host is installed and configured before you initialize.
3. Create a host file that contains the host name of each *segment* host. See “[Creating a Host List File](#)” on page 58.
4. Create your Greenplum Database system configuration file. See “[Creating the Greenplum Database Configuration File](#)” on page 59.

5. Run the Greenplum Database initialization utility on the master host. See [“Running the Initialization Utility”](#) on page 59.

Creating a Host List File

The host list file is used by the `gpinitssystem` system initialization utility to determine the hosts on which to create the Greenplum Database segment instances. This file has just the host names of the segment hosts. If using a multi-NIC configuration, this file should have *all* per-interface host names for each segment host — one host name per line.

To create the host list file

1. Create a file in any location you like. The examples in this documentation call this file `multi_seg_hosts_file`. For example:

```
$ vi multi_seg_hosts_file
```

2. In this file add the host name of each *segment* host, one name per line, no extra lines or spaces. If using multiple network interfaces per segment host, you must initialize the array using *all* configured host names of each segment host. For example:

```
sdw1-1
sdw1-2
sdw1-3
sdw1-4
sdw2-1
sdw2-2
sdw2-3
sdw2-4
sdw3-1
sdw3-2
sdw3-3
sdw3-4
```

3. Save and close the file.
4. If you created this file as `root`, make sure to change the ownership to the `gpadmin` user or group. For example:


```
# chown gpadmin multi_seg_hosts_file
# chgrp gpadmin multi_seg_hosts_file
```
5. Note the location where this file resides, as you will need to specify it for the `MACHINE_LIST_FILE` parameter in the next task, [“Creating the Greenplum Database Configuration File”](#) on page 59.

Creating the Greenplum Database Configuration File

Your Greenplum Database configuration file tells the `gpinitssystem` initialization utility how you want to configure your Greenplum Database system. An example configuration file can be found in

`$GPHOME/docs/cli_help/gp_init_config_example`. Also see “[Initialization Configuration File Reference](#)” on page 788 for a detailed description of each parameter.

To create a `gp_init_config` file

1. Make a copy of the `gp_init_config_example` document to use as a starting point. For example:

```
$ cp $GPHOME/docs/cli_help/gp_init_config_example
/home/gpadmin/gp_init_config
```

2. Open the file you just copied in a text editor. For example:

```
$ vi gp_init_config
```

3. Set all of the required parameters according to your environment. See “[Required Parameters](#)” on page 788 for more information. A Greenplum Database system must contain a master instance and *at least two* segment instances (even if setting up a demo system on a single host).

When creating the configuration file used by `gpinitssystem`, make sure you specify the correct number of segment instance data directories per host. Here is an example of the required parameters in the `gpdb_init_config` file:

```
ARRAY_NAME="Greenplum"
MACHINE_LIST_FILE=/home/gpadmin/multi_seg_hosts_file
SEG_PREFIX=gp
PORT_BASE=50000
declare -a DATA_DIRECTORY=(/data1/gpdb_p1 /data1/gpdb_p2
/data1/gpdb_p3 /data1/gpdb_p4)
MASTER_HOSTNAME=mdw1
MASTER_DIRECTORY=/data1/gpdb_p1
MASTER_PORT=5432
```

4. (optional) If you want to deploy mirror segments, set the mirroring parameters according to your environment. See “[Optional Parameters for Mirror Segments](#)” on page 790 for more information.

Note: You can also deploy mirrors later using the `gpaddmirrors` utility.

5. Save and close the file.

Running the Initialization Utility

The `gpinitssystem` utility will create a Greenplum Database system using the values defined in the configuration file.

To run the initialization utility

1. Run the following command referencing the path and file name of your initialization configuration file (`gp_init_config`). For example:

```
$ gpinitssystem -c /home/gpadmin/gp_init_config
```

If deploying an optional standby master, you would run:

```
$ gpinitssystem -c /home/gpadmin/gp_init_config -s
standby_master_hostname
```

2. The utility will verify your setup information and make sure it can connect to each host and access the data directories specified in your configuration. If all of the pre-checks are successful, the utility will prompt you to confirm your configuration. For example:

```
=> Continue with Greenplum creation? y
```

3. The utility will then begin setup and initialization of the master instance and each segment instance in the system. Each segment instance is set up in parallel. Depending on the number of segments, this process can take a while.

4. At the end of a successful setup, the utility will start your Greenplum Database system. You should see:

```
=> Greenplum Database instance successfully created.
```

Troubleshooting Initialization Problems

If the utility encounters any errors while setting up an instance, the entire process will fail, and could possibly leave you with a partially created system. Refer to the error messages and logs to determine the cause of the failure and where in the process the failure occurred. Log files are created in `~/gpAdminLogs`.

Depending on when the error occurred in the process, you may need to clean up and then try the `gpinitssystem` utility again. For example, if some segment instances were created and some failed, you may need to stop `postgres` processes and remove any utility-created data directories from your data storage area(s). A backout script is created to help with this cleanup if necessary.

Using the Backout Script

If the `gpinitssystem` utility fails, it will create the following backout script if it has left your system in a partially installed state:

```
~/gpAdminLogs/backout_gpinitssystem_<user>_<timestamp>
```

You can use this script to clean up a partially created Greenplum Database system. This backout script will remove any utility-created data directories, `postgres` processes, and log files. After correcting the error that caused `gpinitssystem` to fail and running the backout script, you should be ready to retry initializing your Greenplum Database array.

The following example shows how to run the backout script:

```
$ sh backout_gpinitssystem_gpadmin_20071031_121053
```

Setting the Master Data Directory Environment Variable

The Greenplum Database management utilities require that the `MASTER_DATA_DIRECTORY` environment variable be set. This should point to the directory created by the `gpinitssystem` utility in the master data directory location.

For example, you could add a line similar to the following to the `gpadmin` user's profile file (such as `.bashrc`):

```
MASTER_DATA_DIRECTORY=/gpdata/gp-1
export MASTER_DATA_DIRECTORY
```

After editing the chosen profile file, source it as the correct user to make the changes active. For example:

```
$ source ~/.bashrc
```

Next Steps

After your system is up and running, the next steps are:

- [Allowing Client Connections](#)
- [Creating Databases and Loading Data](#)

Allowing Client Connections

After a Greenplum Database is first initialized it will only allow local connections to the database from the `gpadmin` role (or whatever system user ran `gpinitssystem`). If you would like other users or client machines to be able to connect to Greenplum Database, you must give them access. See [Section III, “Access Control and Security”](#).

Creating Databases and Loading Data

After verifying your installation, you may want to begin creating databases and loading data. See [Section IV, “Database Administration”](#) for more information about creating databases, schemas, tables, and other database objects in Greenplum Database and loading your data.

9. Greenplum Database Demo Programs

This chapter provides information about the demo programs provided with your Greenplum Database distribution. The following demo programs are available:

- [Greenplum Database Demo \(gpdemo\)](#)
- [Running MIVP](#)

Greenplum Database Demo (gpdemo)

The Greenplum Database demo program will setup a virtual Greenplum Database system on a single host. It creates and starts a master instance and three segment instances within the same directory. Running the demo is a good way to test your initial installation before deploying the Greenplum Database software on a multi-host array. It is also a good way to become familiar with the system without investing the time involved in a multi-host setup.

Before You Begin

Before running the demo, you must have at least 110 MB of free disk space, and have ports 18501, 18507, 18508, and 18509 free.

Prerequisites for Solaris Users

If you are running the Greenplum Database demo programs on Solaris, make sure you have the following:

- Make sure that `make`, `gtar`, and a C compiler (`gcc` recommended) are in your `$PATH`.
- Verify that your `$USER` environment variable is set.
- You are using the bash shell to run the demo programs.

Running Greenplum Database Demo (gpdemo)

1. Login as the `gpadmin` user.

```
$ su - gpadmin
```
2. Go to the directory where you wish to unpack and run the demo program. This directory should have at least 110 MB of free space.
3. Unzip and untar `gpdemo.tar.gz`:

```
$ gtar -xvzf $GPHOME/demo/gpdemo.tar.gz
```
4. Go into the `gpdemo` directory you just untarred:

```
$ cd gpdemo
```
5. Run `make`:

```
$ make
```

6. If setup completes correctly, you will have a virtual Greenplum Database system running on the local host. You can connect to it using the `psql` client program as follows:

```
$ psql template1 -p 18501
```

7. To exit the `psql` client program:

```
=# \q
```

8. To explore the demo installation further, try “[Running MIVP](#)” on page 63 or see “[About TPCB](#)” on page 64.

Running MIVP

The Greenplum Multiple Instance Verification Program (MIVP) can be used to test an installed and running Greenplum Database system. MIVP is included inside the `gpdemo` program package and is configured by default to run with your demo system on default port 18501.

The MIVP program does the following tests:

- Connects to a Greenplum master database server instance running on the default port of 18501.
- Creates a sample database called `gptestdb`.
- Generates approximately 200 MB of sample data in your Greenplum Database system using a data generator program.
- Loads the data into the `gptestdb` database using `COPY`.
- Runs a sample query.

To run MIVP

1. Go to the following directory:

```
GPDEMO_HOME/gpdemo/MIVP
```

Note: you must untar the `gpdemo` package first in order to access MIVP. See “[Greenplum Database Demo \(gpdemo\)](#)” on page 62.

2. If running MIVP against a production Greenplum Database system (not the demo), you may need to edit the `Makefile` first. See “[Editing MIVP to Test a Production Installation](#)” on page 64.

3. Run `make`:

```
$ make
```

4. If you encounter any errors or problems, check the `MIVP.log` file.

5. If the program runs successfully, you should see results similar to the following:

```
Executing step "query-IVP"
The result from the next query should be 1000000
time psql -p 18501 gptestdb -c "select count(*) from
bigtable1"
count
```

```

-----
1000000
(1 row)

real          0m0.742s
user          0m0.002s
sys           0m0.003s
MIVP: load and query test completed successfully.

```

To remove MIVP

1. From the `MIVP` directory run `make clean`:

```
$ make clean
```

2. This will drop the `gpctestdb` database that the `MIVP` program created.

Editing MIVP to Test a Production Installation

Running `MIVP` against a multi-host installation of Greenplum Database is the same as running it against the demo installation. The only thing you may need to change before running `MIVP` against a production install is the port number. The demo uses the default port of 18501 for its master. If your production system is using a different master port, edit the `MIVP Makefile` and change all occurrences of 18501 to the correct port number.

Note that the demo also uses ports 18507, 18508, 18509 for its segment instances, so you may want to remove your demo installation first to free those ports if you are using any of those ports for your production Greenplum master instance. See “[Removing Greenplum Database Demo \(gpdemo\)](#)” on page 65.

About TPCH

The Transaction Processing Performance Council (TPC) is a third-party organization that provides database benchmark tools for the industry. TPC-H is their ad-hoc, decision support benchmark. This benchmark illustrates decision support systems that examine large volumes of data, execute queries with a high degree of complexity, and give answers to critical business questions. The TPC-H toolkit is used for Greenplum Database functional and performance testing.

The demo program includes the TPC-H version 1.0.1 specification in:

```
GPDEMO_HOME/gpdemo/TPCH/appendix
```

Refer to the TPC-H `README` files for instructions.

For more information about TPC-H, go to:

<http://www.tpc.org/tpch>

Removing Greenplum Database Demo (gpdemo)

To remove MIVP

1. From the `gpdemo` directory run make clean:

```
$ make clean
```
2. This will stop all Greenplum Database demo server processes (the master and the three segment instances), and remove all files and directories created by the demo program.

Section III: Access Control and Security

This section describes how to manage access to your Greenplum Database system. It contains the following chapters:

- [Managing Roles and Privileges](#) - Provides information on creating database roles (users and groups) and managing access privileges to database objects.
- [Configuring Client Authentication](#) - Provides information about the `pg_hba.conf` file, a configuration file used to control client access and authentication to Greenplum Database.
- [Accessing the Database](#) - Explains the various client tools you can use to connect to Greenplum Database, and how to establish a database session.
- [Managing Workload and Resources](#) - Describes the workload management feature of Greenplum Database, and explains the tasks involved in creating and managing resource queues.

10. Managing Roles and Privileges

Greenplum Database manages database access permissions using the concept of *roles*. The concept of roles subsumes the concepts of *users* and *groups*. A role can be a database user, a group, or both. Roles can own database objects (for example, tables) and can assign privileges on those objects to other roles to control access to the objects. Roles can be members of other roles, thus a member role can inherit the object privileges of its parent role.

Every Greenplum Database system contains a set of database roles (users and groups). Those roles are separate from the users and groups managed by the operating system on which the server runs. However, for convenience you may want to maintain a relationship between operating system user names and Greenplum Database role names, since many of the client applications use the current operating system user name as the default.

In Greenplum Database, users log in and connect through the master instance, which then verifies their role and access privileges. The master then issues out commands to the segment instances behind the scenes as the currently logged in role.

Roles are defined at the system level, meaning they are valid for all databases in the system.

In order to bootstrap the Greenplum Database system, a freshly initialized system always contains one predefined *superuser* role. This role will have the same name as the operating system user that initialized the Greenplum Database system. Customarily, this role is named `gpadmin`. In order to create more roles you first have to connect as this initial role.

Security Best Practices for Roles and Privileges

- Secure the `gpadmin` system user.** Greenplum requires a UNIX user id to install and initialize the Greenplum Database system. This system user is referred to as `gpadmin` in the Greenplum documentation. This `gpadmin` user is the default database superuser in Greenplum Database, as well as the file system owner of the Greenplum installation and its underlying data files. This default administrator account is fundamental to the design of Greenplum Database. The system cannot run without it, and there is no way to limit the access of this `gpadmin` user id. This `gpadmin` user can bypass all security features of Greenplum Database. Anyone who logs on to a Greenplum host as this user id can read, alter or delete any data, including system catalog data and database access rights. Therefore, it is very important to secure the `gpadmin` user id and only provide access to essential system administrators. Administrators should only log in to Greenplum as `gpadmin` when performing certain system maintenance tasks (such as upgrade or expansion). Database users should never log on as `gpadmin`, and ETL or production workloads should never run as `gpadmin`.

- **Assign a distinct role to each user that logs in.** For logging and auditing purposes, each user that is allowed to log in to Greenplum Database should be given their own database role. For applications or web services, consider creating a distinct role for each application or service. See “[Creating New Roles \(Users\)](#)” on page 68.
- **Use groups to manage access privileges.** See “[Creating Groups \(Role Membership\)](#)” on page 69.
- **Limit users who have the SUPERUSER role attribute.** Roles that are superusers bypass all access privilege checks in Greenplum Database, as well as resource queuing. Only system administrators should be given superuser rights. See “[Altering Role Attributes](#)” on page 68.

Creating New Roles (Users)

A user-level role is considered to be a database role that can log in to the database and initiate a database session. Therefore, when you create a new user-level role using the `CREATE ROLE` command, you must specify the `LOGIN` privilege. For example:

```
=# CREATE ROLE jsmith WITH LOGIN;
```

A database role may have a number of attributes that define what sort of tasks that role can perform in the database. You can set these attributes when you create the role, or later using the `ALTER ROLE` command. See [Table 10.1, “Role Attributes”](#) on page 68 for a description of the role attributes you can set.

Altering Role Attributes

A database role may have a number of attributes that define what sort of tasks that role can perform in the database.

Table 10.1 Role Attributes

Attributes	Description
SUPERUSER NOSUPERUSER	Determines if the role is a superuser. A superuser always bypasses all access permission checks within the database and has full access to everything. Superuser status is dangerous and should be used only when really needed. You must yourself be a superuser to create a new superuser. NOSUPERUSER is the default.
CREATEDB NOCREATEDB	Determines if the role is allowed to create databases. NOCREATEDB is the default.
CREATEROLE NOCREATEROLE	Determines if the role is allowed to create and manage other roles. NOCREATEROLE is the default.
INHERIT NOINHERIT	Determines whether a role inherits the privileges of roles it is a member of. A role with the INHERIT attribute can automatically use whatever database privileges have been granted to all roles it is directly or indirectly a member of. INHERIT is the default.
LOGIN NOLOGIN	Determines whether a role is allowed to log in. A role having the LOGIN attribute can be thought of as a user. Roles without this attribute are useful for managing database privileges (groups). NOLOGIN is the default.

Table 10.1 Role Attributes

Attributes	Description
CONNECTION LIMIT <i>connlimit</i>	If role can log in, this specifies how many concurrent connections the role can make. -1 (the default) means no limit.
PASSWORD ' <i>password</i> '	Sets the role's password. If you do not plan to use password authentication you can omit this option. If no password is specified, the password will be set to null and password authentication will always fail for that user. A null password can optionally be written explicitly as PASSWORD NULL.
ENCRYPTED UNENCRYPTED	Controls whether the password is stored encrypted in the system catalogs. The default behavior is determined by the configuration parameter <code>password_encryption</code> (currently set to MD5). If the presented password string is already in MD5-encrypted format, then it is stored encrypted as-is, regardless of whether ENCRYPTED or UNENCRYPTED is specified (since the system cannot decrypt the specified encrypted password string). This allows reloading of encrypted passwords during dump/restore.
VALID UNTIL ' <i>timestamp</i> '	Sets a date and time after which the role's password is no longer valid. If omitted the password will be valid for all time.
RESOURCE QUEUE <i>queue_name</i>	Assigns the role to the named resource queue for workload management. Any statement that role issues is then subject to the resource queue's limits. Note that the RESOURCE QUEUE attribute is not inherited; it must be set on each user-level (LOGIN) role.

You can set these attributes when you create the role, or later using the `ALTER ROLE` command. For example:

```

=# ALTER ROLE jsmith WITH PASSWORD 'passwd123';
=# ALTER ROLE admin VALID UNTIL 'infinity';
=# ALTER ROLE jsmith LOGIN;
=# ALTER ROLE jsmith RESOURCE QUEUE adhoc;

```

A role can also have role-specific defaults for many of the server configuration settings. For example, to set the default schema search path for a role:

```

=# ALTER ROLE admin SET search_path TO myschema, public;

```

Creating Groups (Role Membership)

It is frequently convenient to group users together to ease management of object privileges: that way, privileges can be granted to, or revoked from, a group as a whole. In Greenplum Database this is done by creating a role that represents the group, and then granting membership in the group role to individual user roles.

Use the `CREATE ROLE` SQL command to create a new group role. For example:

```

=# CREATE ROLE admin CREATEROLE CREATEDB;

```

Once the group role exists, you can add and remove members (user roles) using the `GRANT` and `REVOKE` commands. For example:

```

=# GRANT admin TO john, sally;
=# REVOKE admin FROM bob;

```

For managing object privileges, you would then grant the appropriate permissions to the group-level role only (see [Table 10.2, “Object Privileges”](#) on page 70). The member user roles then inherit the object privileges of the group role. For example:

```
=# GRANT ALL ON TABLE mytable TO admin;
=# GRANT ALL ON SCHEMA myschema TO admin;
=# GRANT ALL ON DATABASE mydb TO admin;
```

The role attributes `LOGIN`, `SUPERUSER`, `CREATEDB`, and `CREATEROLE` are never inherited as ordinary privileges on database objects are. User members must actually `SET ROLE` to a specific role having one of these attributes in order to make use of the attribute. In the above example, we gave `CREATEDB` and `CREATEROLE` to the `admin` role. If `sally` is a member of `admin`, she could issue the following command to assume the role attributes of the parent role:

```
=> SET ROLE admin;
```

Managing Object Privileges

When an object (table, view, sequence, database, function, language, schema, or tablespace) is created, it is assigned an owner. The owner is normally the role that executed the creation statement. For most kinds of objects, the initial state is that only the owner (or a superuser) can do anything with the object. To allow other roles to use it, privileges must be granted. Greenplum Database supports the following privileges for each object type:

Table 10.2 Object Privileges

Object Type	Privileges
Tables, Views, Sequences	SELECT INSERT UPDATE DELETE RULE ALL
External Tables	SELECT RULE ALL
Databases	CONNECT CREATE TEMPORARY TEMP ALL
Functions	EXECUTE
Procedural Languages	USAGE
Schemas	CREATE USAGE ALL



Note: Privileges must be granted for each object individually. For example, granting ALL on a database does not grant access to the tables in that database.

Use the `GRANT SQL` command to give a specified role privileges on an object. For example:

```
=# GRANT INSERT ON mytable TO jsmith;
```

To revoke privileges, use the `REVOKE` command. For example:

```
=# REVOKE ALL PRIVILEGES ON mytable FROM jsmith;
```

You can also use the `DROP OWNED` and `REASSIGN OWNED` commands for managing objects owned by deprecated roles (Note: only an object's owner or a superuser can drop an object or reassign ownership). For example:

```
=# REASSIGN OWNED BY sally TO bob;
=# DROP OWNED BY visitor;
```

Simulating Row and Column Level Access Control

Greenplum Database access control corresponds roughly to the Orange Book 'C2' level of security, not the 'B1' level. Greenplum Database currently supports access privileges at the object level. Row-level or column-level access is not supported, nor is labeled security.

Row-level and column-level access can be simulated using views to restrict the columns and/or rows that are selected. Row-level labels can be simulated by adding an extra column to the table to store sensitivity information, and then using views to control row-level access based on this column. Roles can then be granted access to the views rather than the base table. While these workarounds do not provide the same as "B1" level security, they may still be a viable alternative for many organizations that require more granular access control.

Encrypting Data

PostgreSQL provides an optional package of encryption/decryption functions called `pgcrypto`, which can also be installed and used in Greenplum Database. The `pgcrypto` package is not installed by default with Greenplum Database, however Greenplum can provide a platform-specific build of `pgcrypto` upon request. Contact Greenplum Customer Support to obtain a build of `pgcrypto` and its supporting documentation.

The `pgcrypto` functions allow database administrators to store certain columns of data in encrypted form. This adds an extra layer of protection for sensitive data, as data stored in Greenplum Database in encrypted form cannot be read by users who do not have the encryption key, nor be read directly from the disks.

It is important to note that the `pgcrypto` functions run inside database server. That means that all the data and passwords move between `pgcrypto` and the client application in clear-text. For optimal security, consider also using SSL connections between the client and the Greenplum master server.

11. Configuring Client Authentication

When a Greenplum Database system is first initialized, the system contains one predefined *superuser* role. This role will have the same name as the operating system user who initialized the Greenplum Database system. This role is referred to as `gpadmin`. By default, the system is configured to only allow local connections to the database from the `gpadmin` role. If you want to allow any other roles to connect, or if you want to allow connections from remote hosts, you have to configure Greenplum Database to allow such connections. This chapter explains how to configure client connections and authentication to Greenplum Database.

- [Allowing Connections to Greenplum Database](#)
- [Limiting Concurrent Connections](#)

Allowing Connections to Greenplum Database

Client access and authentication is controlled by a configuration file named `pg_hba.conf` (the standard PostgreSQL host-based authentication file). For detailed information about this file, see [The `pg_hba.conf` File](#) in the PostgreSQL documentation.

In Greenplum Database, the `pg_hba.conf` file of the master instance controls client access and authentication to your Greenplum system. The segments also have `pg_hba.conf` files, but these are already correctly configured to only allow client connections from the master host. The segments never accept outside client connections, so there is no need to alter the `pg_hba.conf` file on your segments.

The general format of the `pg_hba.conf` file is a set of records, one per line. Blank lines are ignored, as is any text after the `#` comment character. A record is made up of a number of fields which are separated by spaces and/or tabs. Fields can contain white space if the field value is quoted. Records cannot be continued across lines. Each remote client access record is in the format of:

```
host database role CIDR-address authentication-method
```

Each Unix-domain socket access record is in the format of:

```
local database role authentication-method
```

The meaning of the fields is as follows:

Table 11.1 `pg_hba.conf` Fields

Field	Description
local	Matches connection attempts using Unix-domain sockets. Without a record of this type, Unix-domain socket connections are disallowed.
host	Matches connection attempts made using TCP/IP. Remote TCP/IP connections will not be possible unless the server is started with an appropriate value for the <code>listen_addresses</code> server configuration parameter.

Table 11.1 pg_hba.conf Fields

Field	Description
database	Specifies which database names this record matches. The value <code>all</code> specifies that it matches all databases. Multiple database names can be supplied by separating them with commas. A separate file containing database names can be specified by preceding the file name with <code>@</code> .
role	Specifies which database role names this record matches. The value <code>all</code> specifies that it matches all roles. If the specified role is a group and you want all members of that group to be included, precede the role name with a <code>+</code> . Multiple role names can be supplied by separating them with commas. A separate file containing role names can be specified by preceding the file name with <code>@</code> .
CIDR-address	Specifies the client machine IP address range that this record matches. It contains an IP address in standard dotted decimal notation and a CIDR mask length. IP addresses can only be specified numerically, not as domain or host names. The mask length indicates the number of high-order bits of the client IP address that must match. Bits to the right of this must be zero in the given IP address. There must not be any white space between the IP address, the <code>/</code> , and the CIDR mask length. Typical examples of a CIDR-address are <code>172.20.143.89/32</code> for a single host, or <code>172.20.143.0/24</code> for a small network, or <code>10.6.0.0/16</code> for a larger one. To specify a single host, use a CIDR mask of 32 for IPv4 or 128 for IPv6. In a network address, do not omit trailing zeroes.
authentication-method	Specifies the authentication method to use when connecting. See Authentication Methods in the PostgreSQL 8.3 documentation for details.

Editing the pg_hba.conf File

This example shows how to edit the `pg_hba.conf` file of the master to allow remote client access to all databases from all roles using md5-encrypted password authentication.



Note: For a more secure system, consider removing all connections that use `trust` authentication from your master `pg_hba.conf`. Trust authentication means the role is granted access without any authentication, therefore bypassing all security. Replace `trust` entries with `ident` authentication if your system has an ident service available.

Editing pg_hba.conf

1. Open the file `$MASTER_DATA_DIRECTORY/pg_hba.conf` in a text editor.
2. Add a line to the file for each type of connection you want to allow. Records are read sequentially, so the order of the records is significant. Typically, earlier records will have tight connection match parameters and weaker authentication methods, while later records will have looser match parameters and stronger authentication methods. For example:

```
# allow the gpadmin user local access to all databases
# using ident authentication
local all gpadmin ident sameuser
host all gpadmin 127.0.0.1/32 ident
host all gpadmin ::1/128 ident
# Now allow any role remote access to any database from any
# host with IP address 192.168.x.x and use md5 encrypted
# passwords to authenticate the user
host all all 192.168.0.0/32 md5
```

3. Save and close the file.
4. Reload the `pg_hba.conf` configuration file for your changes to take effect:


```
$ gpstop -u
```



Note: Note that you can also control database access by setting object privileges as described in “[Managing Object Privileges](#)” on page 70. The `pg_hba.conf` file just controls who can initiate a database session and how those connections are authenticated.

Limiting Concurrent Connections

To limit the number of active concurrent sessions to your Greenplum Database system, you can configure the `max_connections` server configuration parameter. This is a *local* parameter, meaning that you must set it in the `postgresql.conf` file of the master, the standby master, and each segment instance (primary and mirror). The value of `max_connections` on segments must be 5-10 times the value on the master.

When you set `max_connections`, you must also set the dependent parameter `max_prepared_transactions`. This value must be at least as large as the value of `max_connections` on the master, and segment instances should be set to the same value as the master.

For example:

In `$MASTER_DATA_DIRECTORY/postgresql.conf` (including standby master):

```
max_connections=100
max_prepared_transactions=100
```

In `SEGMENT_DATA_DIRECTORY/postgresql.conf` for all segment instances:

```
max_connections=500
max_prepared_transactions=100
```

To change the number of allowed connections

1. Stop your Greenplum Database system:


```
$ gpstop
```

2. On your master host, edit `$MASTER_DATA_DIRECTORY/postgresql.conf` and change the following two parameters:
 - `max_connections` (the number of active user sessions you want to allow plus the number of `superuser_reserved_connections`)
 - `max_prepared_transactions` (must be greater than or equal to `max_connections`)
3. On each segment instance, edit `SEGMENT_DATA_DIRECTORY/postgresql.conf` and change the following two parameters:
 - `max_connections` (must be 5-10 times the value on the master)
 - `max_prepared_transactions` (must be equal to the value on the master)
4. Restart your Greenplum Database system:


```
$ gpstart
```



Note: Raising the values of these parameters may cause Greenplum Database to request more shared memory. To mitigate this effect, consider decreasing other memory-related parameters such as `gp_cached_segworkers_threshold`.

Encrypting Client/Server Connections

Greenplum Database has native support for SSL connections between the client and the master server. SSL connections prevent third parties from snooping on the packets, and also prevent man-in-the-middle attacks. SSL should be used whenever the client connection goes through an insecure link, and must be used whenever client certificate authentication is used.

To enable SSL requires that OpenSSL be installed on both the client and the master server systems. Greenplum can be started with SSL enabled by setting the server configuration parameter `ssl=on` in the master `postgresql.conf`. When starting in SSL mode, the server will look for the files `server.key` (server private key) and `server.crt` (server certificate) in the master data directory. These files must be set up correctly before an SSL-enabled Greenplum system can start. If the private key is protected with a passphrase, the server will prompt for the passphrase and will not start until it has been entered.

For details on how to create your server private key and certificate, refer to the OpenSSL documentation. A self-signed certificate can be used for testing, but a certificate signed by a certificate authority (CA) (either one of the global CAs or a local one) should be used in production so the client can verify the server's identity.

12. Accessing the Database

This chapter explains the various client tools you can use to connect to Greenplum Database, and how to establish a database session. It contains the following topics:

- [Establishing a Database Session](#)
- [Supported Client Applications](#)
- [Troubleshooting Connection Problems](#)

Establishing a Database Session

Users can connect to Greenplum Database using a PostgreSQL-compatible client program, such as `psql`. Users and administrators *always* connect to Greenplum Database through the *master* - the segments cannot accept client connections.

In order to establish a connection to the Greenplum Database master, you will need to know the following connection information and configure your client program accordingly.

Table 12.1 Connection Parameters

Connection Parameter	Description	Environment Variable
Database name	The name of the database to which you want to connect. For a newly initialized system, use the <code>template1</code> database to connect for the first time.	<code>\$PGDATABASE</code>
Host name	The host name of the Greenplum Database master. The default host is the local host.	<code>\$PGHOST</code>
Port	The port number that the Greenplum Database master instance is running on. The default is 5432.	<code>\$PGPORT</code>
User name	The database user (role) name to connect as. This is not necessarily the same as your OS user name. Check with your Greenplum administrator if you are not sure what your database user name is. Note that every Greenplum Database system has one superuser account that is created automatically at initialization time. This account has the same name as the OS name of the user who initialized the Greenplum system (typically <code>gadmin</code>).	<code>\$PGUSER</code>

Supported Client Applications

Users can connect to Greenplum Database using various client applications:

- A number of [Greenplum Database Client Applications](#) are provided with your Greenplum installation. The `psql` client application provides an interactive command-line interface to Greenplum Database.
- [pgAdmin III for Greenplum Database](#) is an enhanced version of the popular management tool pgAdmin III. Since version 1.10.0, the pgAdmin III client available from PostgreSQL Tools includes support for Greenplum-specific features. Installation packages are available for download from [Greenplum Network](#) and from the [pgAdmin download site](#).
- Using standard [Graphical Query Plan in pgAdmin III](#), such as ODBC and JDBC, users can create their own client applications that interface to Greenplum Database. Because Greenplum Database is based on PostgreSQL, it uses the standard PostgreSQL database drivers.
- Most [Third-Party Client Tools](#) that use standard database interfaces, such as ODBC and JDBC, can be configured to connect to Greenplum Database.

Greenplum Database Client Applications

Greenplum Database comes installed with a number of client applications located in `$GPHOME/bin` of your Greenplum Database master host installation. The following are the most commonly used client applications:

Table 12.2 Commonly used client applications

Name	Usage
<code>createdb</code>	create a new database
<code>createlang</code>	define a new procedural language
<code>createuser</code>	define a new database role
<code>dropdb</code>	remove a database
<code>droplang</code>	remove a procedural language
<code>dropuser</code>	remove a role
<code>psql</code>	PostgreSQL interactive terminal
<code>reindexdb</code>	reindex a database
<code>vacuumdb</code>	garbage-collect and analyze a database

When using these client applications, you must connect to a database through the Greenplum master instance. You will need to know the name of your target database, the host name and port number of the master, and what database user name to connect as. This information can be provided on the command-line using the options `-d`, `-h`, `-p`, and `-U` respectively. If an argument is found that does not belong to any option, it will be interpreted as the database name first.

All of these options have default values which will be used if the option is not specified. The default host is the local host. The default port number is 5432. The default user name is your OS system user name, as is the default database name. Note that OS user names and Greenplum Database user names are not necessarily the same.

If the default values are not correct, you can save yourself some typing by setting the environment variables `PGDATABASE`, `PGHOST`, `PGPORT`, and `PGUSER` to the appropriate values. See “[Greenplum Environment Variables](#)” on page 812 for more information. It is also convenient to have a `~/.pgpass` file to avoid regularly having to type in passwords. See “[psql](#)” on page 730 for more information.

Connecting with psql

Depending on the default values used or the environment variables you have set, the following examples show how to access a database via `psql`:

```
$ psql -d gpdatabase -h master_host -p 5432 -U gpadmin
$ psql gpdatabase
$ psql
```

If a user-defined database has not yet been created, you can access the system by connecting to the `template1` database. For example:

```
$ psql template1
```

After connecting to a database, `psql` provides a prompt with the name of the database to which `psql` is currently connected, followed by the string `=>` (or `=#` if you are the database super user). For example:

```
gpdatabase=>
```

At the prompt, you may type in SQL commands. A SQL command must end with a `;` (semicolon) in order to be sent to the server and executed. For example:

```
=> SELECT * FROM mytable;
```

For more information on using the `psql` client application, see “[psql](#)” on page 730. For more information on SQL commands and syntax, see “[SQL Command Reference](#)” on page 259.

pgAdmin III for Greenplum Database

If you prefer a graphic interface, use pgAdmin III for Greenplum Database. This GUI client supports PostgreSQL databases with all standard pgAdmin III features, while adding support for Greenplum-specific features.

pgAdmin III for Greenplum Database supports the following Greenplum-specific features:

- External tables
- Append-only tables, including compressed append-only tables
- Table partitioning
- Resource queues
- Graphical EXPLAIN ANALYZE

- Greenplum server configuration parameters

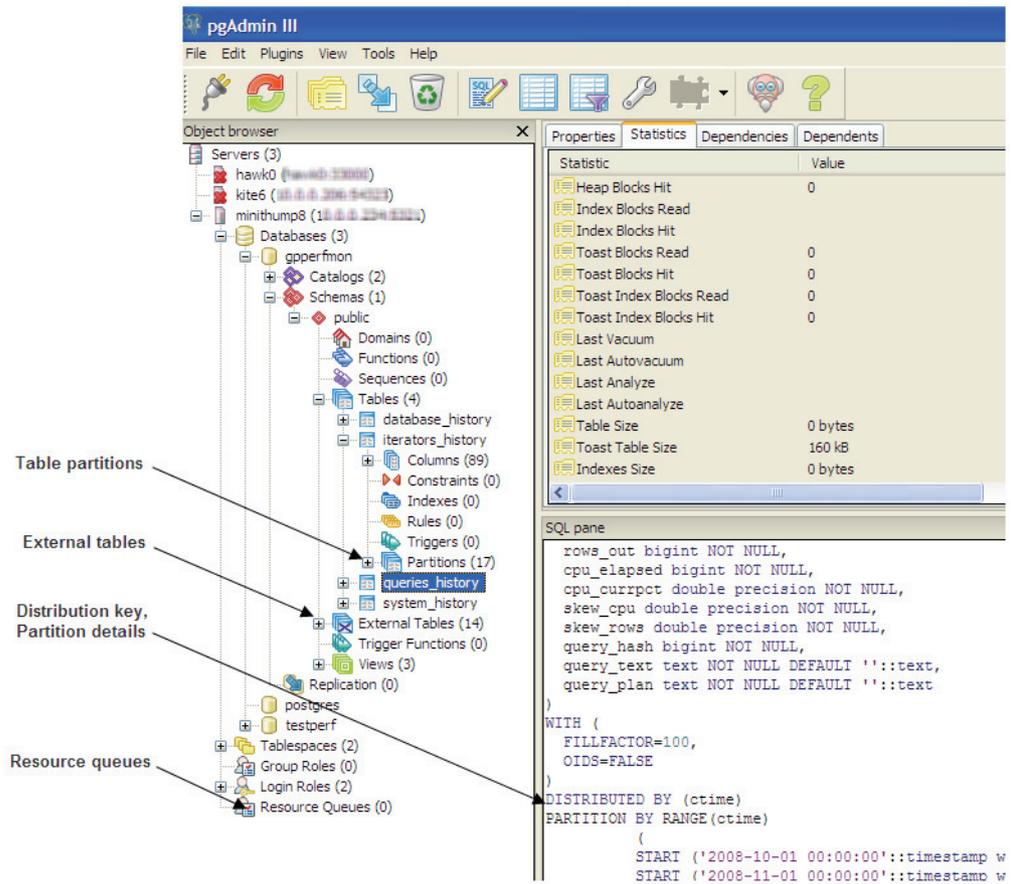


Figure 12.1 Greenplum Options in pgAdmin III

Installing pgAdmin III for Greenplum Database

The installation package for pgAdmin III for Greenplum Database is available for download from the official pgAdmin III download site (<http://www.pgadmin.org>). Installation instructions are included in the installation package.

Documentation for pgAdmin III for Greenplum Database

For general help on the features of the graphical interface, select **Help contents** from the **Help** menu.

For help with Greenplum-specific SQL support, select **Greenplum Database Help** from the **Help** menu. If you have an active internet connection, you will be directed to online Greenplum SQL reference documentation. Alternately, you can install the Greenplum Client Tools package. This package contains SQL reference documentation that is accessible to the help links in pgAdmin III.

Performing Administrative Tasks with pgAdmin III

This section highlights two of the many Greenplum Database administrative tasks you can perform with pgAdmin III: editing the server configuration, and viewing a graphical representation of a query plan.

Editing Server Configuration

The pgAdmin III interface provides two ways to update the server configuration in `postgresql.conf`: locally, through the **File** menu, and remotely on the server through the **Tools** menu. Editing the server configuration remotely may be more convenient in many cases, because it does not require you to upload or copy `postgresql.conf`.

To edit server configuration remotely

1. Connect to the server whose configuration you want to edit. If you are connected to multiple servers, make sure that the correct server is highlighted in the object browser in the left pane.
2. Select **Tools > Server Configuration > postgresql.conf**. The Backend Configuration Editor opens, displaying the list of available and enabled server configuration parameters.
3. Locate the parameter you want to edit, and double click on the entry to open the Configuration settings dialog.
4. Enter the new value for the parameter, or select/deselect **Enabled** as desired and click **OK**.
5. If the parameter can be enabled by reloading server configuration, click the green reload icon, or select **File > Reload server**. Many parameters require a full restart of the server.

Viewing a Graphical Query Plan

Using the pgAdmin III query tool, you can run a query with EXPLAIN to view the details of the query plan. The output includes details about operations unique to Greenplum distributed query processing such as plan slices and motions between segments. You can view a graphical depiction of the plan as well as the text-based data output.

To view a graphical query plan

1. With the correct database highlighted in the object browser in the left pane, select **Tools > Query** tool.
2. Enter the query by typing in the SQL Editor, dragging objects into the Graphical Query Builder, or opening a file.
3. Select **Query > Explain** options and verify the following options:
 - **Verbose** — this must be deselected if you want to view a graphical depiction of the query plan
 - **Analyze** — select this option if you want to run the query in addition to viewing the plan
4. Trigger the operation by clicking the Explain query option at the top of the pane, or by selecting **Query > Explain**.

The query plan displays in the Output pane at the bottom of the screen. Select the Explain tab to view the graphical output. For example:

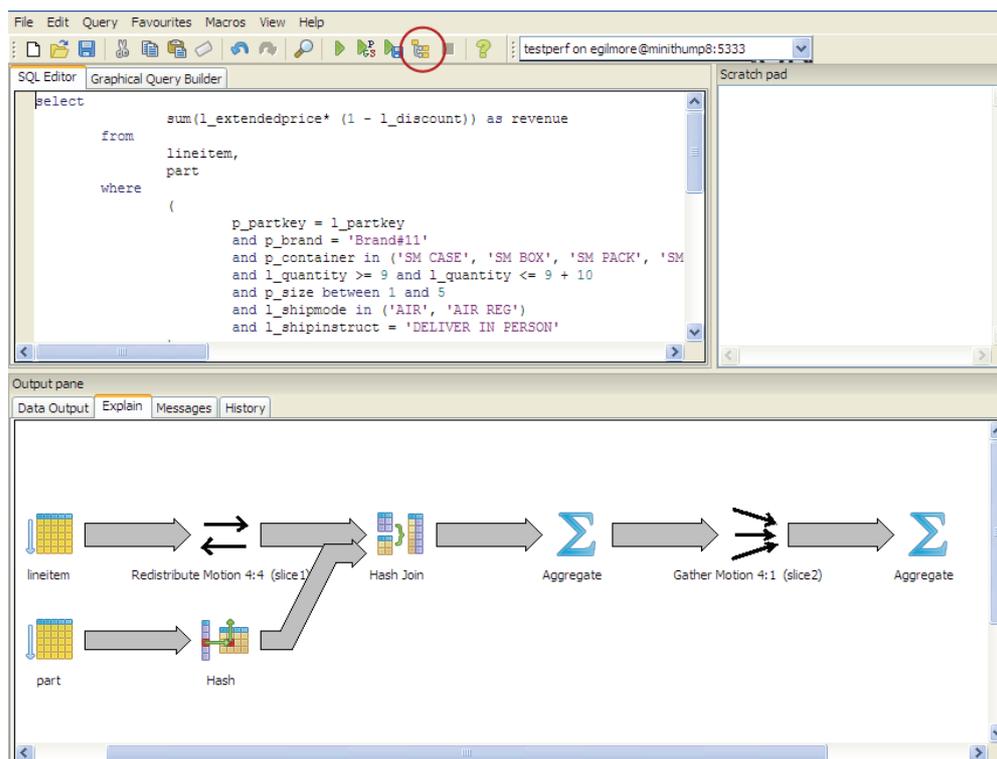


Figure 12.2 Graphical Query Plan in pgAdmin III

Database Application Interfaces

You may want to develop your own client applications that interface to Greenplum Database. PostgreSQL provides a number of database drivers for the most commonly used database application programming interfaces (APIs), which can also be used with Greenplum Database. These drivers are not packaged with the Greenplum Database base distribution. Each driver is an independent PostgreSQL development project and must be downloaded, installed and configured to connect to Greenplum Database. The following drivers are available:

Table 12.3 Greenplum Database Interfaces

API	PostgreSQL Driver	Download Link
ODBC	pgodbc	Available in the <i>Greenplum Database Connectivity</i> package, which can be downloaded from Greenplum Network .
JDBC	pgjdbc	Available in the <i>Greenplum Database Connectivity</i> package, which can be downloaded from Greenplum Network .
Perl DBI	pgperl	http://gborg.postgresql.org/project/pgperl
Python DBI	pygresql	http://www.pygresql.org

General instructions for accessing a Greenplum Database with an API are:

1. Download your programming language platform and respective API from the appropriate source. For example, you can get the Java development kit (JDK) and JDBC API from Sun.
2. Write your client application according to the API specifications. When programming your application, be aware of the SQL support in Greenplum Database so you do not include any unsupported SQL syntax. See “[SQL Command Reference](#)” on page 259.
3. Download the appropriate PostgreSQL driver and configure connectivity to your Greenplum Database master instance. Greenplum provides a client tools package that contains the supported database drivers for Greenplum Database. The client tools package (and associated documentation) is available for download from [Greenplum Network](#).

Third-Party Client Tools

Most third-party extract-transform-load (ETL) and business intelligence (BI) tools use standard database interfaces, such as ODBC and JDBC, and can be configured to connect to Greenplum Database. Greenplum has worked with the following tools on previous customer engagements and is in the process of becoming officially certified:

- Business Objects
- Microstrategy
- Informatica Power Center
- Microsoft SQL Server Integration Services (SSIS) and Reporting Services (SSRS)
- Ascential Datastage
- SAS
- Cognos

Greenplum Professional Services can assist users in configuring their chosen third-party tool for use with Greenplum Database.

Troubleshooting Connection Problems

A number of things can prevent a client application from successfully connecting to Greenplum Database. This section explains some of the common causes of connection problems and how to correct them.

Table 12.4 Common connection problems

Problem	Solution
No <code>pg_hba.conf</code> entry for host or user	In order for Greenplum Database to be able to accept remote client connections, you must configure your Greenplum Database master instance so that connections are allowed from the client hosts and database users that will be connecting to Greenplum Database. This is done by adding the appropriate entries to the <code>pg_hba.conf</code> configuration file (located in the master instance's data directory). For more detailed information, see “Allowing Connections to Greenplum Database” on page 73.
Greenplum Database is not running	If the Greenplum Database master instance is down, users will not be able to connect. You can verify that the Greenplum Database system is up by running the <code>gpstate</code> utility on the Greenplum master host.
Network problems Interconnect timeouts	If users are connecting to the Greenplum master host from a remote client, network problems may be preventing a connection (for example, DNS host name resolution problems, the host system is down, etc.). To ensure that network problems are not the cause, try connecting to the Greenplum master host from the remote client host. For example: <code>ping hostname</code> If the system cannot resolve the host names and IP addresses of the hosts involved in Greenplum Database, queries and connections will fail. Keep in mind that for some operations, connections to the Greenplum Database master use <code>localhost</code> while others use the actual host name, so you must be able to resolve both. If you encounter this error, first make sure you can connect to each host in your Greenplum Database array from the master host over the network. In the <code>/etc/hosts</code> file of the master and all segments, make sure you have the correct host names and IP addresses for all hosts involved in the Greenplum Database array. The <code>127.0.0.1</code> IP should only resolve to <code>localhost</code> .
Too many clients already	By default, Greenplum Database is configured to allow a maximum of 25 concurrent user connections. A connection attempt that causes that limit to be exceeded will be refused. This limit is controlled by the <code>max_connections</code> parameter in the <code>postgresql.conf</code> configuration file of the Greenplum Database master. If you change this setting for the master, you must also make appropriate changes at the segments.

13. Managing Workload and Resources

This chapter describes the workload management feature of Greenplum Database, and explains the tasks involved in creating and managing resource queues. The following topics are covered in this chapter:

- [Overview of Greenplum Workload Management](#)
- [Configuring Workload Management](#)
- [Creating Resource Queues](#)
- [Assigning Roles \(Users\) to a Resource Queue](#)
- [Modifying Resource Queues](#)
- [Checking Resource Queue Status](#)

Overview of Greenplum Workload Management

The purpose of Greenplum Database workload management is to limit the number of active queries in the system at any given time in order to avoid exhausting system resources such as memory, CPU, and disk I/O. This is accomplished by creating role-based *resource queues*. A resource queue has attributes that limit the size and/or total number of queries that can be executed by the users (or roles) in that queue. By assigning all of your database roles to the appropriate resource queue, administrators can control concurrent user queries and prevent the system from being overloaded.

How Resource Queues Work in Greenplum Database

Administrators create resource queues for the various types of workloads in their organization. For example, you may have a resource queue for power users, web users, and management reports. The administrator would then set limits on the resource queue based on his estimate of how resource-intensive the queries associated with that workload are likely to be. Currently, the only configurable limits on a queue are *active statement cost* and/or *active statement count*, however more queue attributes will be added in future Greenplum Database releases. Database roles (users and groups) are then assigned to the appropriate resource queue. A resource queue can have multiple roles, but a role can have only one assigned resource queue.

At runtime, when the user submits a query for execution, that query is evaluated against the resource queue's limits. If the query does not cause the queue to exceed its resource limits, then that query will run immediately. If the query causes the queue to exceed its limits (for example, if the maximum number of active statement slots are

currently in use), then the query must wait until queue resources are free before it can run. Queries submitted through a queue are evaluated and executed on a first in, first out basis.

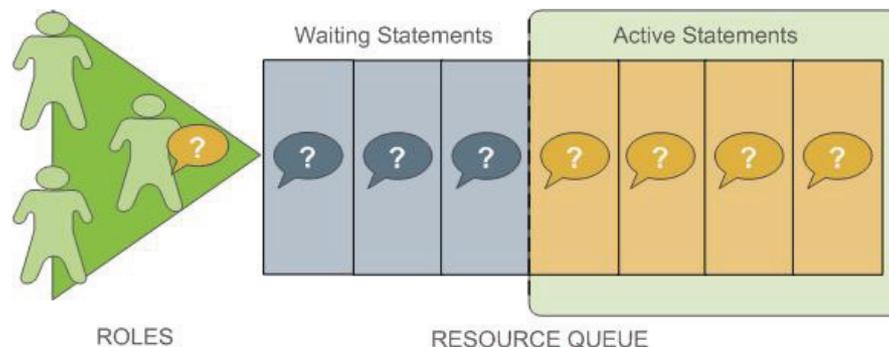


Figure 13.1 Resource Queue Example

Exempt Users and Superusers

You may decide that some roles (users) should be exempt from resource scheduling. This means that any query issued by that role is automatically let in to the system without any limits on query cost or number of active statements. Queries from exempt roles always execute immediately. To make a role exempt, you would simply not assign that role to a resource queue.

Roles with the SUPERUSER attribute are *always* exempt from resource scheduling, regardless of whether they are assigned to a resource queue or not.

Types of Queries Evaluated for Resource Queues

Not all SQL statements submitted through a resource queue are evaluated against the queue limits. By default only `SELECT`, `SELECT INTO`, `CREATE TABLE AS SELECT`, and `DECLARE CURSOR` statements are evaluated. If the server configuration parameter `resource_select_only` is set to *off*, then `INSERT`, `UPDATE`, and `DELETE` statements will be evaluated as well.

Steps to Enable Workload Management

Enabling and using workload management in Greenplum Database involves the following high-level tasks:

1. Setting the server configuration parameters for workload management. See [“Configuring Workload Management”](#) on page 87.
2. Creating the resource queues and setting limits on them. See [“Creating Resource Queues”](#) on page 87.
3. Assigning a queue to one or more user roles. See [“Assigning Roles \(Users\) to a Resource Queue”](#) on page 89.

- Using the workload management system views to monitor and manage the resource queues. See “Checking Resource Queue Status” on page 90.

Configuring Workload Management

Resource scheduling is enabled by default when you install Greenplum Database. If you decide not to use resource queues for workload management, you can disable the feature. This is done by setting a server configuration parameter in the `postgresql.conf` file of your master instance.

To configure workload management

- The resource scheduling feature is enabled by default. To confirm that it is enabled, make sure the `resource_scheduler` parameter is set to `on`:


```
$ psql template1 -c 'SHOW resource_scheduler;'
```
- (optional) You can also configure the following additional workload management parameters:
 - `max_resource_queues` - Sets the maximum number of resource queues.
 - `max_resource_portals_per_transaction` - Sets the maximum number of simultaneously open cursors allowed per transaction. Note that an open cursor will hold an active query slot in a resource queue.
 - `resource_select_only` - If set to `on`, then `SELECT`, `SELECT INTO`, `CREATE TABLE AS SELECT`, and `DECLARE CURSOR` commands are evaluated. If set to `off` `INSERT`, `UPDATE`, and `DELETE` commands will be evaluated as well.
 - `resource_cleanup_gangs_on_wait` - Cleans up idle processes before taking a slot in the resource queue.
 - `stats_queue_level` - Enables statistics collection on resource queue usage, which can then be viewed by querying the `pg_stat_resqueues` system view.
- If you wish to change any of the default parameter values, edit the `postgresql.conf` configuration file of your master instance. For example:


```
$ vi $MASTER_DATA_DIRECTORY/postgresql.conf
```
- Save and close the `postgresql.conf` file after you have made your changes.
- Stop and restart Greenplum Database to reload the configuration file changes.


```
$ gpstop
$ gpstart
```

Creating Resource Queues

Creating a resource queue involves giving it a name and setting either a query cost threshold or an active query threshold (or both) on the resource queue. Use the `CREATE RESOURCE QUEUE` command to create new resource queues.

Creating Queues with an Active Threshold

Resource queues with an `ACTIVE THRESHOLD` limit the number of queries that can be executed by roles assigned to that queue. For example, to create a resource queue named *adhoc* with an active query limit of three:

```
=# CREATE RESOURCE QUEUE adhoc ACTIVE THRESHOLD 3;
```

This means that for all roles assigned to the *adhoc* resource queue, only three active queries can be running on the system at any given time. If this queue has three queries running, and a fourth query is submitted by a role in that queue, that query must wait until a slot is free before it can run.

Creating Queues with a Cost Threshold

Resource queues with a `COST THRESHOLD` limit the total cost of queries that can be executed by roles assigned to that queue. Cost is specified as a floating point number (for example 100.0) or can also be specified as an exponent (for example 1e+2).

Cost is measured in the *estimated total cost* for the query as determined by the Greenplum query planner (as shown in the `EXPLAIN` output for a query). Therefore, an administrator must be familiar with the queries typically executed on the system in order to set an appropriate cost threshold for a queue. Cost is measured in units of disk page fetches; 1.0 equals one sequential disk page read.

For example, to create a resource queue named *webuser* with a query cost limit of 100000.0 (1e+5):

```
=# CREATE RESOURCE QUEUE webuser COST THRESHOLD 100000.0;
```

or

```
=# CREATE RESOURCE QUEUE webuser COST THRESHOLD 1e+5;
```

This means that for all roles assigned to the *webuser* resource queue, it will only allow queries into the system until the cost limit of 100000.0 is reached. So for example, if this queue has 200 queries with a 500.0 cost all running at the same time, and query 201 with a 1000.0 cost is submitted by a role in that queue, that query must wait until space is free before it can run.

Cost Threshold Overcommit

If a resource queue is limited based on a cost threshold, then the administrator can allow `OVERCOMMIT` (the default). Resource queues with a cost threshold and overcommit enabled will allow a query that exceeds the cost threshold to run, provided that there are no other queries in the queue at the time the query is submitted. The cost threshold will only be enforced if there are multiple sessions submitting queries through the queue at once.

If `NOOVERCOMMIT` is specified, then queries that exceed the cost limit will always be rejected and never allowed to run.

Assigning Roles (Users) to a Resource Queue

Once a resource queue is created, you must assign roles (users) to their appropriate resource queue. Use the `ALTER ROLE` or `CREATE ROLE` commands to assign a role to a resource queue. For example:

```
=# ALTER ROLE name RESOURCE QUEUE queue_name;  
=# CREATE ROLE name WITH LOGIN RESOURCE QUEUE queue_name;
```

A role can only be assigned to one resource queue at any given time, so you can use the `ALTER ROLE` command to initially assign or change a role's resource queue.

Resource queues must be assigned on a user-by-user basis. If you have a role hierarchy (for example, a group-level role) then assigning a resource queue to the group does not propagate down to the users in that group.

Superusers are always exempt from resource queue limits. Even if you assign a superuser role to a resource queue, their queries will always run regardless just as though they are not in a queue at all.

Roles that are not assigned to a resource queue are considered exempt from resource scheduling, which means that any query issued by that role is automatically let in to the system without any limits on query cost or number of active statements.

Removing a Role from a Resource Queue

If you wish to remove a role from a resource queue, change the role's queue assignment to `none`. For example:

```
=# ALTER ROLE role_name RESOURCE QUEUE none;
```

Modifying Resource Queues

After a resource queue has been created, you can change or reset the active query or query cost limits using the `ALTER RESOURCE QUEUE` command. You can remove a resource queue using the `DROP RESOURCE QUEUE` command. To change the roles (users or groups) assigned to a resource queue, see [“Assigning Roles \(Users\) to a Resource Queue”](#) on page 89.

Altering a Resource Queue

The `ALTER RESOURCE QUEUE` command changes the limits of a resource queue. A resource queue must have either an `ACTIVE THRESHOLD` or a `COST THRESHOLD` value (or it can have both). To change the limit of a resource queue, specify the new threshold value you want for the queue. For example:

```
=# ALTER RESOURCE QUEUE queue_name ACTIVE THRESHOLD integer;  
=# ALTER RESOURCE QUEUE queue_name COST THRESHOLD float;
```

To reset an `ACTIVE THRESHOLD` to have no limit, enter a value of `-1`. To reset a `COST THRESHOLD` to have no limit, enter a value of `-1.0`. For example:

```
=# ALTER RESOURCE QUEUE queue_name COST THRESHOLD -1.0;
```

Dropping a Resource Queue

The `DROP RESOURCE QUEUE` command drops a resource queue. To drop a resource queue, the queue cannot have any roles assigned to it, nor can it have any statements waiting in the queue. See “[Removing a Role from a Resource Queue](#)” on page 89 and “[Clearing a Waiting Statement From a Resource Queue](#)” on page 92 for instructions on emptying a resource queue. To drop a resource queue:

```
=# DROP RESOURCE QUEUE name;
```

Checking Resource Queue Status

Checking resource queue status involves the following tasks:

- [Viewing Resource Queue Status](#)
- [Viewing Resource Queue Statistics](#)
- [Viewing the Roles Assigned to a Resource Queue](#)
- [Viewing the Waiting Queries for a Resource Queue](#)
- [Clearing a Waiting Statement From a Resource Queue](#)

Viewing Resource Queue Status

The `pg_resqueue_status` system view allows administrators to see status and activity for a workload management resource queue. It shows how many queries are waiting to run and how many queries are currently active in the system from a particular resource queue. To see the resource queues created in the system, their limit attributes, and their current status:

```
=# SELECT * FROM pg_resqueue_status;
```

Viewing Resource Queue Statistics

If you want to track statistics and performance of resource queues over time, you can enable statistics collecting for resource queues. This is done by setting the following server configuration parameter in your master `postgresql.conf` file:

```
stats_queue_level = on
```

Once this is enabled, you can use the `pg_stat_resqueues` system view to see the statistics collected on resource queue usage. Note that enabling this feature does incur slight performance overhead, as each query submitted through a resource queue must be tracked. It may be useful to enable statistics collecting on resource queues for initial diagnostics and administrative planning, and then disable the feature for continued use.

See the section on the [Statistics Collector](#) in the PostgreSQL documentation for more information about collecting statistics in Greenplum Database.

Viewing the Roles Assigned to a Resource Queue

To see the roles assigned to a resource queue, perform the following query of the `pg_roles` and `pg_resqueue` system catalog tables:

```
=# SELECT rolname, rsqname FROM pg_roles, pg_resqueue
   WHERE pg_roles.rolresqueue=pg_resqueue.oid;
```

You may want to create a view of this query to simplify future inquiries. For example:

```
=# CREATE VIEW role2queue AS
   SELECT rolname, rsqname FROM pg_roles, pg_resqueue
   WHERE pg_roles.rolresqueue=pg_resqueue.oid;
```

Then you can just query the view:

```
=# SELECT * FROM role2queue;
```

Viewing the Waiting Queries for a Resource Queue

When a slot is in use for a resource queue, it is recorded in the `pg_locks` system catalog table. This is where you can see all of the currently active and waiting queries for all resource queues.

To see all the currently active queries for all resource queues, perform the following query of the `pg_locks` table joined with the `pg_roles` and `pg_resqueue` tables:

```
=# SELECT rolname, rsqname, locktype, objid, transaction,
   pid, mode, granted FROM pg_roles, pg_resqueue, pg_locks
   WHERE pg_roles.rolresqueue=pg_locks.objid
   AND pg_locks.objid=pg_resqueue.oid;
```

You may want to create a view of this query to simplify future inquiries. For example:

```
=# CREATE VIEW queued_statements AS
   SELECT rolname, rsqname, locktype, objid, transaction,
   pid, mode, granted FROM pg_roles, pg_resqueue, pg_locks
   WHERE pg_roles.rolresqueue=pg_locks.objid
   AND pg_locks.objid=pg_resqueue.oid;
```

Then you can just query the view:

```
=# SELECT * FROM queued_statements;
```

If this query returns no results, then that means there are currently no statements in a resource queue. A sample of a resource queue with two statements in it looks something like this:

```
rolname | rsqname | locktype | objid | transaction | pid | mode | granted
-----|-----|-----|-----|-----|-----|-----|-----
sammy | webuser | resource queue | 16510 | 1054 | 31905 | ExclusiveLock | f
daria | webuser | resource queue | 16510 | 1051 | 31861 | ExclusiveLock | t
(2 rows)
```

The `objid` is the object ID of the resource queue. Statements with a `granted` status of `t` are currently active. Statements with a `granted` status of `f` are waiting in the queue.

Clearing a Waiting Statement From a Resource Queue

In some cases, you may want to clear a waiting statement from a resource queue. For example, you may want to remove a query that is waiting in the queue but has not been executed yet. You may also want to stop a query that has been started if it is taking too long to execute, or if it is sitting idle in a transaction and taking up resource queue slots that are needed by other users. To do this, you must first identify the statement you want to clear, determine its process id (pid), and then kill that process id.

For example, to see process information about all statements currently active or waiting in all resource queues, run the following query:

```
=# SELECT rolname, rsqname, pid, granted,
       current_query, datname
   FROM pg_roles, pg_resqueue, pg_locks, pg_stat_activity
  WHERE pg_roles.rolresqueue=pg_locks.objid
 AND pg_locks.objid=pg_resqueue.oid
 AND pg_stat_activity.procpid=pg_locks.pid;
```

You may want to create a view of this query to simplify future inquiries. For example:

```
=# CREATE VIEW resqueue_procs AS
   SELECT rolname, rsqname, pid, granted,
          current_query, datname
   FROM pg_roles, pg_resqueue, pg_locks, pg_stat_activity
  WHERE pg_roles.rolresqueue=pg_locks.objid
 AND pg_locks.objid=pg_resqueue.oid
 AND pg_stat_activity.procpid=pg_locks.pid;
```

Then you can just query the view:

```
=# SELECT * FROM resqueue_procs;
```

If this query returns no results, then that means there are currently no statements in a resource queue. A sample of a resource queue with two statements in it looks something like this:

rolname	rsqname	pid	granted	current_query	datname
sammy	webuser	31861	t	<IDLE> in transaction	namesdb
daria	webuser	31905	f	SELECT * FROM topten;	namesdb

Use this output to identify the process id (pid) of the statement you want to clear from the resource queue. To clear the statement, you would then open a terminal window (as the database superuser or as root) on the master host and kill the corresponding process. For example:

```
=# kill 31905
```

Section IV: Database Administration

This section describes how create, manage and access databases and database objects using SQL (structured query language).

- [Defining Database Objects](#) - This chapter covers data definition language (DDL) in Greenplum Database and how to create and manage database objects.
- [Managing Data](#) - This chapter covers data manipulation language (DML) in Greenplum Database and how transaction concurrency is handled.
- [Querying Data](#) - This chapter describes the use of the SQL language in Greenplum Database.
- [Parallel Data Loading](#) - This chapter describes the various ways to load data into Greenplum Database.

14. Defining Database Objects

This chapter covers data definition language (DDL) in Greenplum Database and how to create and manage database objects.

- [Creating and Managing Databases](#)
- [Creating and Managing Schemas](#)
- [Creating and Managing Tables](#)
- [Partitioning Large Tables](#)
- [Creating and Using Sequences](#)
- [Using Indexes in Greenplum Database](#)
- [Creating and Managing Views](#)

Creating and Managing Databases

A Greenplum Database system can have one or more databases. This is different from some other database management systems (such as Oracle) where the database instance *is* the database. Although you can create many databases in a Greenplum system, client programs can connect and access one database at a time — you cannot cross-query between databases.

About Template Databases

Every new database you create is based on a *template*. A default database called `template1` exists in every newly initialized Greenplum Database system. You can use this database to connect to Greenplum Database for the first time. This is the template used to create other databases by default if you do not explicitly declare a template when creating a new database. You do not want to create any objects in this database unless you want those objects to also be in every other database you create afterwards.

In addition to `template1`, every newly created Greenplum system has two other database templates, `template0` and `postgres`, which are used internally by the system and should not be dropped or modified. The `template0` database template can be used to create a completely clean database containing only the standard objects predefined by Greenplum Database at initialization. This is useful if you wish to avoid copying any objects that may have been added to `template1`.

Creating a Database

The `CREATE DATABASE` command creates a new database. For example:

```
=> CREATE DATABASE new_dbname;
```

In order to create a database, you must have privileges to create a database or be a Greenplum superuser. If you do not have the correct privileges, then you will not be able to create a database. Contact your Greenplum administrator to either give you the necessary privilege or to create a database for you.

There is also a client program called `createdb` that you can use to create a database. For example, running the following command in a command line terminal will make a connection to Greenplum Database using the provided host name and port and create the database named *mydatabase*:

```
$ createdb -h masterhost -p 5432 mydatabase
```

Cloning a Database

By default, a new database is created by cloning the standard system database `template1`. Any database can be used as a template when creating a new database, thereby providing the capability to ‘clone’ or copy an existing database and all of the objects and data within that database. For example:

```
=> CREATE DATABASE new_dbname TEMPLATE old_dbname;
```

Viewing the List of Databases

If you are working in the `psql` client program, you can use the `\l` meta-command to show the list of databases and templates in your Greenplum Database system. If using another client program, you can query the list of databases from the `pg_database` system catalog table (you must be a superuser). For example:

```
=> SELECT datname from pg_database;
```

Altering a Database

The `ALTER DATABASE` command is used to change certain attributes about a database, such its owner, name or default configuration attributes. You must be either the owner of the database or a superuser to alter it. Here is an example of altering a database to set its default schema search path (the `search_path` configuration parameter):

```
=> ALTER DATABASE mydatabase SET search_path TO myschema,
public, pg_catalog;
```

Dropping a Database

The `DROP DATABASE` command can be used to drop (or delete) a database. It removes the system catalog entries for the database and also deletes the database directory on disk containing the data. You must be the database owner or a superuser to drop a database, and you cannot drop a database while you or anyone else is connected to it. Connect into `template1` (or another database) before dropping a database. For example:

```
=> \c template1
=> DROP DATABASE mydatabase;
```

There is also a client program called `dropdb` that you can use to drop a database. For example, running the following command in a command line terminal will make a connection to Greenplum Database using the provided host name and port and drop the database named *mydatabase*:

```
$ dropdb -h masterhost -p 5432 mydatabase
```



Warning: Dropping a database cannot be undone, so use this command with care!

Creating and Managing Schemas

Schemas are a way to logically organize objects and data in a database. Schemas allow you to have more than one object (such as tables) with the same name in the database without conflict, as long as they are in different schemas.

The Default 'Public' Schema

Every newly created database has a default schema named *public*. If you do not create any schemas of your own, objects will be created in the *public* schema by default. All database roles (users) have `CREATE` and `USAGE` privileges in the *public* schema by default. Any other schemas you create, you will have to grant the appropriate privileges so that users can access the schema.

Creating a Schema

Use the `CREATE SCHEMA` command to create a new schema. For example:

```
=> CREATE SCHEMA myschema;
```

To create or access objects in a schema, write a qualified name consisting of the schema name and table name separated by a dot. For example:

```
schema.table
```

See “[Schema Search Paths](#)” on page 96 for more information about accessing a schema.

You may want to create a schema owned by someone else (since this is one of the ways to restrict the activities of your users to well-defined namespaces). The syntax for that is:

```
=> CREATE SCHEMA schemaname AUTHORIZATION username;
```

Schema Search Paths

For the database to know in which schema it should look for an object, you can always use the schema-qualified name. For example:

```
=> SELECT * FROM myschema.mytable;
```

If you do not want to type the schema-qualified name all the time, you can set the `search_path` configuration parameter. This tells the database in which order it should search the available schemas for objects. The schema listed first in the search path becomes the *default* schema. The default schema is where new objects will be created if a schema name is not explicitly declared.

Setting the Schema Search Path

The `search_path` configuration parameter is used to set the order in which schemas are searched. You can set `search_path` for a database using the `ALTER DATABASE` command. For example:

```
=> ALTER DATABASE mydatabase SET search_path TO myschema,
public, pg_catalog;
```

You can also set `search_path` for a particular role (user) using the `ALTER ROLE` command. For example:

```
=> ALTER ROLE sally SET search_path TO myschema, public,
pg_catalog;
```

Viewing the Current Schema

There may be times when you are not sure what schema you are currently in or what your `search_path` setting is. To find out this information, you can use the `current_schema()` function or the `SHOW` command. For example:

```
=> SELECT current_schema();
=> SHOW search_path;
```

Dropping a Schema

Use the `DROP SCHEMA` command to drop (delete) a schema. For example:

```
=> DROP SCHEMA myschema;
```

By default, the schema must be empty before you can drop it. If you want to drop a schema and all of the objects in that schema (tables, data, functions, etc.) use:

```
=> DROP SCHEMA myschema CASCADE;
```

System Schemas

The following system-level schemas also exist in every database:

- **pg_catalog** is the schema that has the system catalog tables, built-in data types, functions, and operators. It is always part of the schema search path, even if it is not explicitly named in the search path. For information on the system catalog tables in the `pg_catalog` schema, see “[System Catalog Reference](#)” on page 817.
- **information_schema** consists of a standardized set of views that contain information about the objects in the database. These views are used to get system information from the system catalog tables in a standardized way.
- **pg_toast** is a system schema where large objects are stored (records that exceed the page size). This schema is used internally by the Greenplum Database system and is not typically accessed by database administrators or users.

- `pg_bitmapindex` is the system schema where bitmap index objects are stored (list of values, etc.). This schema is used internally by the Greenplum Database system and is not typically accessed by database administrators or users.
- `pg_aoseg` is the system schema where append-only table objects are stored. This schema is used internally by the Greenplum Database system and is not typically accessed by database administrators or users.
- `gp_jetpack` is an administrative schema that can be optionally installed by a Greenplum Database superuser. It contains a number of views and functions for checking log files and other system status metrics. See [“Using the Jetpack Administrative Interface”](#) on page 226.

Creating and Managing Tables

A table in Greenplum Database is much like a table in any other relational database, except that the records in the table are distributed across the different segments in the system. When you create a table, there is additional SQL syntax to declare the table’s distribution policy.

Creating a Table

The `CREATE TABLE` command is used to create a new table and define its structure. When creating a table, you will typically define the following aspects of the table:

- The columns of the table and their associated data types. See [“Choosing Column Data Types”](#) on page 98.
- Any table or column constraints to limit the data that a column or table can contain. See [“Setting Table and Column Constraints”](#) on page 99.
- The distribution policy of the table, which determines how the data is divided across the Greenplum Database segments. See [“Choosing the Table Distribution Policy”](#) on page 100.
- The way the table is stored on disk. See [“Choosing the Table Storage Model”](#) on page 101.
- The table partitioning strategy for large tables. See [“Partitioning Large Tables”](#) on page 107.

Choosing Column Data Types

The data type of a column determines the types of data values that column can contain. As a general rule, you want to choose the data type that uses the least possible space, yet can still accommodate your data. You should also select the data type that best constrains the data in that column. For example, use character data types for strings, date or timestamp data types for dates, and numeric data types for numbers.

For character column data types, there are no performance differences between the use of `CHAR`, `VARCHAR`, and `TEXT` data types, apart from the increased storage size when using the blank-padded type. While `CHAR` has performance advantages in some other database systems, it has no such advantages in Greenplum Database. In most situations, `TEXT` or `VARCHAR` should be used instead.

For numeric column data types, use the smallest data type in which the data will fit. A lot of space is wasted if, for example, the `BIGINT` data type is used when the data would always fit in `INT` or `SMALLINT`.

Also consider using identical data types for the columns you plan to use in cross-table joins. Joins work much more efficiently if the data types of the columns used in the join predicate (usually the primary key in one table and a foreign key in the other table) have identical data types. When the data types are different, the database has to convert one of them so that the data values can be compared correctly, and such conversion amounts to unnecessary overhead.

Greenplum Database has a rich set of native data types available to users. See [“Greenplum Database Data Types”](#) on page 814 for a list of the built-in data types.

Setting Table and Column Constraints

Data types are a way to limit the kind of data that can be stored in a table. For many applications, however, the constraint they provide is too coarse. SQL allows you to define constraints on columns and tables. Constraints give you more control over the data in your tables. If a user attempts to store data in a column that would violate a constraint, an error is raised.

There are some limitations and conditions when using constraints in Greenplum Database, most notably with regards to foreign key, primary key, and unique constraints. Otherwise constraints are supported as in PostgreSQL.

Check Constraints

A check constraint is the most generic constraint type. It allows you to specify that the value in a certain column must satisfy a Boolean (truth-value) expression. For instance, to require positive product prices, you could use:

```
=> CREATE TABLE products
      ( product_no integer,
        name text,
        price numeric CHECK (price > 0) );
```

Not-Null Constraints

A not-null constraint simply specifies that a column must not assume the null value. A not-null constraint is always written as a column constraint. For example:

```
=> CREATE TABLE products
      ( product_no integer NOT NULL,
        name text NOT NULL,
        price numeric );
```

Unique Constraints

Unique constraints ensure that the data contained in a column or a group of columns is unique with respect to all the rows in the table. For a unique constraint to be enforced in Greenplum Database, the table must be hash-distributed (not `DISTRIBUTED RANDOMLY`), and the constraint columns must be the same as (or a superset of) the table's distribution key columns.

```
=> CREATE TABLE products
      ( product_no integer UNIQUE,
        name text,
```

```

        price numeric)
    DISTRIBUTED BY (product_no);

```

Primary Keys

A primary key constraint is simply a combination of a `UNIQUE` constraint and a `NOT NULL` constraint. For a primary key constraint to be enforced in Greenplum Database, the table must be hash-distributed (not `DISTRIBUTED RANDOMLY`), and the primary key columns must be the same as (or a superset of) the table's distribution key columns. If a table has a primary key, this column (or group of columns) is chosen as the distribution key for the table by default. For example:

```

=> CREATE TABLE products
    ( product_no integer PRIMARY KEY,
      name text,
      price numeric)
    DISTRIBUTED BY (product_no);

```

Foreign Keys

Foreign keys are not supported in this release of Greenplum Database. It is possible to declare them, however they will not be enforced.

A foreign key constraint specifies that the values in a column (or a group of columns) must match the values appearing in some row of another table. This maintains the referential integrity between two related tables. In this release of Greenplum Database, referential integrity checks cannot be enforced between the distributed table segments of a Greenplum database.

Choosing the Table Distribution Policy

All Greenplum Database tables are distributed. When you create or alter a table, there is an optional `DISTRIBUTED BY` (hash distribution) or `DISTRIBUTED RANDOMLY` (round-robin distribution) clause to declare how the rows of the table should be distributed. See [“Understanding Greenplum Distribution Policies”](#) on page 13.

The following considerations should be taken into account when declaring a distribution policy for a table (in order of importance):

- **Even Data Distribution** — For the best possible performance, all of the segments should contain equal portions of data. If the data is unbalanced or skewed, then the segments with more data will have to work harder to perform their portion of the query processing. To ensure an even distribution of data, you want to choose a distribution key that is unique for each record, such as the primary key.
- **Local and Distributed Operations** — During query processing, it is faster if the work associated with join, sort or aggregation operations can be done locally at the segment-level rather than at the system-level (distributing tuples across the segments). When tables share a common distribution key in Greenplum Database, joining or sorting on their shared distribution key columns will result in the most efficient query processing, as the majority of the work is done locally at the segment-level. Local operations are approximately 5 times faster than distributed operations. With a random distribution policy, local operations are not an option.

- **Even Query Processing** — When a query is being processed, you want all of the segments to handle an equal amount of the query workload to get the best possible performance. In some cases, query processing workload can be skewed if the table's data distribution policy and the query predicates are not well matched. For example, suppose you have a table of sales transactions. The table is distributed based on a column that contains corporate names as values. The hashing algorithm distributes the data based on the values of the distribution key, so if a predicate in a query references a single value from the distribution key, the work in the query will run on only one segment. This may be a viable distribution policy if your query predicates tend to select data on a criteria other than corporation name. However, for queries that do use corporation name in their predicates, you can potentially have just one segment instance handling all of the query workload.

Declaring Distribution Keys

When creating a table, there is an additional clause to declare the Greenplum Database distribution policy. If a `DISTRIBUTED BY` or `DISTRIBUTED RANDOMLY` clause is not supplied, then Greenplum assigns a hash distribution policy to the table using either the `PRIMARY KEY` (if the table has one) or the first column of the table as the distribution key. Columns of geometric or user-defined data types are not eligible as Greenplum distribution key columns. If a table does not have a column of an eligible data type, the rows are distributed based on a round-robin or random distribution.

To ensure an even distribution of data, you want to choose a distribution key that is unique for each record, or if that is not possible, then choose `DISTRIBUTED RANDOMLY`. For example:

```
=> CREATE TABLE products
        (name varchar(40),
         prod_id integer,
         supplier_id integer)
        DISTRIBUTED BY (prod_id);
=> CREATE TABLE random_stuff
        (things text,
         doodads text,
         etc text)
        DISTRIBUTED RANDOMLY;
```

Choosing the Table Storage Model

Greenplum Database provides an agile and flexible processing engine capable of supporting several storage models (or a hybrid of storage models). When you create a new table, you have several options as to how its data is stored on disk. This section explains the various options for table storage and how to decide on the best storage model for your workload.

- [Choosing Heap or Append-Only Storage](#)
- [Choosing Row or Column-Oriented Storage](#)
- [Using Compression \(Append-Only Tables Only\)](#)
- [Checking the Compression and Distribution of an Append-Only Table](#)

Choosing Heap or Append-Only Storage

By default, Greenplum Database uses the same heap storage model as PostgreSQL. Heap table storage favors OLTP-type workloads where the data is often modified after it is initially loaded. `UPDATE` and `DELETE` operations require table-level locking and row-level versioning information to be stored in order to ensure that database transactions are processed reliably. Heap tables are best suited for smaller tables, such as dimension tables, that are often updated after they are initially loaded.

Greenplum Database also offers an append-only table storage model. Append-only table storage favors denormalized fact tables in a data warehouse environment, which are typically the largest tables in the system. Fact tables are typically loaded in batches, and then accessed by read-only queries. Data is not updated after it is loaded. Moving large fact tables to an append-only storage model eliminates the storage overhead of the per-row update visibility information (about 20 bytes per row is saved). This allows for a leaner and easier-to-optimize page structure. Append-only tables do not allow `UPDATE` and `DELETE` operations, or the creation of indexes. The storage model of append-only tables is optimized for bulk data loading. Single row `INSERT` statements are not recommended.

To create a heap table

Row-oriented heap tables are the default storage type, so no extra `CREATE TABLE` command syntax is required to create a heap table. For example:

```
=> CREATE TABLE foo (a int, b text) DISTRIBUTED BY (a);
```

To create an append-only table

The `WITH` clause of the `CREATE TABLE` command is used to declare the storage options of the table. If not declared, the table will be created as a regular row-oriented heap-storage table. For example, to create an append-only table with no compression:

```
=> CREATE TABLE bar (a int, b text)
    WITH (appendonly=true)
    DISTRIBUTED BY (a);
```

Choosing Row or Column-Oriented Storage

Greenplum provides a choice of storage orientation models: row or column (or a hybrid of both). This section provides some general guidelines for choosing the correct storage orientation for a table; however you are encouraged to evaluate performance using your own data and query workloads.

For most general purpose or mixed workloads, row-oriented storage offers the best combination of flexibility and performance. However, there are certain specific use cases where a column-oriented storage model provides more efficient I/O and storage. Consider the following requirements when deciding on the storage orientation model of a table:

- **Updates of table data.** If table data must be updated after it is loaded, choose a row-oriented heap table. Column-oriented table storage is only available on append-only tables. See “[Choosing Heap or Append-Only Storage](#)” on page 102 for more information.

- **Frequent INSERTs.** If new rows are frequently inserted into the table, consider a row-oriented model. Column-oriented tables are not optimized for write operations, as column values for a row must be written to different places on disk.
- **Number of columns requested in queries.** If you typically request all or the majority of columns in the `SELECT` list or `WHERE` clause of your queries, consider a row-oriented model. Column-oriented tables are best suited to queries that aggregate many values of a single column where the `WHERE` or `HAVING` predicate is also on the aggregate column,


```
SELECT SUM(salary) ...
SELECT AVG(salary) ... WHERE salary > 10000
```

 or where the `WHERE` predicate is on a single column and is highly selective (returns a relatively small number of rows).


```
SELECT salary, dept ... WHERE state='CA'
```
- **Number of columns in the table.** Row-oriented storage is more efficient when many columns are required at the same time, or when the row-size of a table is relatively small. Column-oriented tables can offer better query performance on wide tables (lots of columns) where you typically only access a small subset of columns in your queries.
- **Compression.** Since column data is of the same data type, there are some storage size optimizations available in column-oriented data that are not available in row-oriented data. For example, many compression schemes make use of the similarity of adjacent data to compress. However, the greater adjacent compression achieved, the more difficult random access may become, as data might need to be uncompressed to be read.

To create a column-oriented table

The `WITH` clause of the `CREATE TABLE` command is used to declare the storage options of the table. If not declared, the table will be created as a row-oriented heap table. Tables that use column-oriented storage must also be append-only tables. For example, to create a column-oriented table:

```
=> CREATE TABLE bar (a int, b text)
    WITH (appendonly=true, orientation=column)
    DISTRIBUTED BY (a);
```

Using Compression (Append-Only Tables Only)

Tables that utilize the append-only storage model also have the option of using in-database compression (with `zlib` or `QuickLZ`) to save disk space. Using in-database compression requires that your segment systems have the available CPU power to compress and uncompress the data. Compressed append-only tables should not be used on file systems that also are using compression. If the file system where your segment data directory resides is a compressed file system, your append-only table should *not* use compression.

When choosing a compression type and level for append-only tables, consider these factors:

- CPU usage
- Compression ratio/disk size

- Speed of compression
- Speed of decompression/scan rate

Though minimizing disk size may be the main goal in compressing tables, the time and CPU capacity required to compress and scan data is also important to consider. Every system has an optimal range of settings where compression most efficiently reduces data size without causing excessively long compression times or slow scan rates.

QuickLZ compression generally uses less CPU capacity and compresses data faster at a lower compression ratio than zlib. Conversely, zlib provides higher compression ratios at lower speeds. At compression level 1 (`compresslevel=1`), QuickLZ and zlib may yield comparable compression ratios (though at different speeds). However, using zlib compression at a higher level of 6 might dramatically increase its advantage over QuickLZ in compression ratio (though consequently lowering the speed of compression).

Performance with compressed append-only tables depends on hardware, query tuning settings, and other factors. Greenplum recommends performing comparison testing to determine the actual performance in your environment.

To create a compressed table

The `WITH` clause of the `CREATE TABLE` command is used to declare the storage options of the table. Tables that use compression must also be append-only tables. For example, to create an append-only table with zlib compression at a compression level of 5:

```
=> CREATE TABLE foo (a int, b text)
    WITH (appendonly=true, compresstype=zlib,
         compresslevel=5);
```



Note: QuickLZ compression level can only be set to level 1; no other options are available. Compression level with zlib can be set at any value from 1 - 9.

Checking the Compression and Distribution of an Append-Only Table

Greenplum provides built-in functions to check the compression ratio and the distribution of an append-only table. Both functions take either the object ID or name of a table. The table name may be qualified with a schema name.

Table 14.1 Functions for compressed append-only table metadata

Function	Return Type	Description
<code>get_ao_distribution(oid,name)</code>	Set of (dbid, tuplecount) rows	Shows the distribution of rows of an append-only table across the array. Returns a set of rows, each of which includes a segment <i>dbid</i> and the number of tuples stored on the segment.
<code>get_ao_compression_ratio(oid,name)</code>	float8	Calculates the compression ratio for a compressed append-only table. If information is not available, this function returns a value of -1.

The compression ratio is returned as a common ratio. For example, a returned value of 3.19, or 3.19:1, means that the uncompressed table is slightly larger than three times the size of the compressed table.

The distribution of the table is returned as a set of rows indicating how many tuples are stored on each segment. For example, in a system with four primary segments with *dbid* values ranging from 0 - 3, the function returns four rows similar to the following:

```
=# SELECT get_ao_distribution('lineitem_comp');
   get_ao_distribution
-----
(0,7500721)
(1,7501365)
(2,7499978)
(3,7497731)
(4 rows)
```

Altering a Table

The `ALTER TABLE` command is used to change the definition of an existing table. With `ALTER TABLE`, you can change various table attributes such as column definitions, distribution policy, storage model, and partition structure (see also “[Maintaining Partitioned Tables](#)” on page 114). For example, to add a not-null constraint to a table column:

```
=> ALTER TABLE address ALTER COLUMN street SET NOT NULL;
```

Altering Table Distribution

`ALTER TABLE` provides options to change the distribution policy of a table. When the distribution options of a table change, the table data is redistributed on disk, which can be resource intensive. There is also an option to redistribute table data using the existing distribution policy.

Changing the Distribution Policy

You can use the `ALTER TABLE` command to change the distribution policy for a table. For partitioned tables, changes to the distribution policy recursively apply to the child partitions. This operation preserves the ownership and all other attributes of the table. For example, the following command redistributes the table `sales` across all segments using the `customer_id` column as the distribution key:

```
ALTER TABLE sales SET DISTRIBUTED BY (customer_id);
```

When you change the hash distribution of a table, table data is automatically redistributed. However, changing the distribution policy to a random distribution will not cause the data to be redistributed. For example:

```
ALTER TABLE sales SET DISTRIBUTED RANDOMLY;
```

Redistributing Table Data

To redistribute table data for tables with a random distribution policy (or when the hash distribution policy has not changed) use `REORGANIZE=TRUE`. This sometimes may be necessary to correct a data skew problem, or when new segment resources have been added to the system. For example:

```
ALTER TABLE sales SET WITH (REORGANIZE=TRUE);
```

This command rebalances a table evenly across all segments using the current distribution policy (including random distribution).

Altering the Table Storage Model

It is not possible to alter the storage model of a table that has already been created. Storage, compression, and orientation can only be declared at `CREATE TABLE` time. If you have an existing table for which you want to change the storage model, you must recreate the table with the correct storage options, reload the data into the newly created table, drop the original table and rename the new table to the original name. You must also re-grant any table permissions. For example:

```
CREATE TABLE sales2 (LIKE sales)
WITH (appendonly=true, compressstype=quicklz,
compresslevel=1, orientation=column);

INSERT INTO sales2 SELECT * FROM sales;
DROP TABLE sales;
ALTER TABLE sales2 RENAME TO sales;
GRANT ALL PRIVILEGES ON sales TO admin;
GRANT SELECT ON sales TO guest;
```

See also, “[Exchanging a Partition](#)” on page 116 for instructions on changing the storage model of a partitioned table.

Dropping a Table

The `DROP TABLE` command removes tables from the database. For example:

```
DROP TABLE mytable;
```

To empty a table of rows without removing the table definition, use `DELETE` or `TRUNCATE`. For example:

```
DELETE FROM mytable;
TRUNCATE mytable;
```

`DROP TABLE` always removes any indexes, rules, triggers, and constraints that exist for the target table. However, to drop a table that is referenced by a view, `CASCADE` must be specified. `CASCADE` will remove a dependent view entirely.

Partitioning Large Tables

Table partitioning addresses the problem of supporting very large tables, such as fact tables, by allowing you to divide them into smaller and more manageable pieces. Partitioned tables can improve query performance by allowing the Greenplum Database query planner to scan only the relevant data needed to satisfy a given query rather than scanning the entire contents of a large table. Partitioned tables can also be used to facilitate database maintenance tasks, such as rolling old data out of the data warehouse.

Understanding Table Partitioning in Greenplum Database

Tables are partitioned at `CREATE TABLE` time using the `PARTITION BY` (and optionally the `SUBPARTITION BY`) clause. When you partition a table in Greenplum Database, you are actually creating a top-level (or parent) table with one or more levels of sub-tables (or child tables). Internally, Greenplum Database creates an inheritance relationship between the top-level table and its underlying partitions (similar to the functionality of the `INHERITS` clause of PostgreSQL).

Using the partition criteria defined during table creation, each partition is created with a distinct `CHECK` constraint, which limits the data that table can contain. The `CHECK` constraints are also used by the query planner to determine which table partitions to scan in order to satisfy a given query predicate.

Partition hierarchy information is stored in the Greenplum system catalog so that rows inserted into the top-level parent table appropriately propagate to the child table partitions. Any changes to the partition design or table structure must be done through the parent table using the `PARTITION` clauses of the `ALTER TABLE` command.

Greenplum Database supports both *range partitioning* (division of data based on a numerical range, such as date or price) or *list partitioning* (division of data based on a list of values, such as sales territory or product line), or a combination of both types.

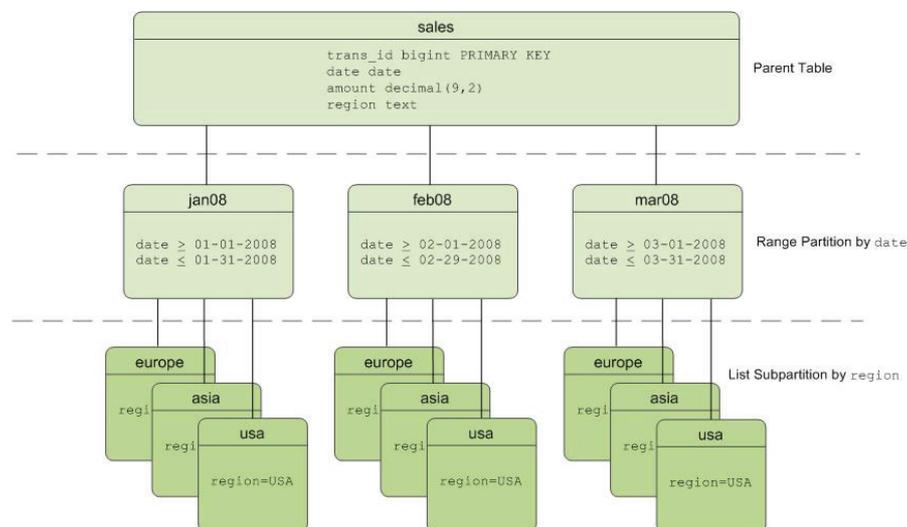


Figure 14.1 Example Multi-level Partition Design

Partitioned tables are also *distributed* across Greenplum Database segments as is any non-partitioned table. Table *distribution* in Greenplum Database physically divides a table across the Greenplum segments to enable parallel query processing. Table *partitioning* is a tool to logically divide big tables to improve query performance and facilitate data warehouse maintenance tasks. Partitioning does not change the physical distribution of the table data across the segments.

Deciding on a Table Partitioning Strategy

Not all tables are good candidates for partitioning. If the answer is *yes* to all or most of the following questions, then table partitioning is a viable database design strategy for improving query performance. If the answer is *no* to most of the following questions, then table partitioning is not the right solution for that table:

- **Is the table large enough?** Large fact tables are good candidates for table partitioning. If you have millions or billions of records in a table, you will see performance benefits from logically breaking that data up into smaller chunks. For smaller tables with only a few thousand rows or less, the administrative overhead of maintaining the partitions will outweigh any performance benefits you might see.
- **Are you experiencing unsatisfactory performance?** As with any performance tuning initiative, a table should be partitioned only if queries against that table are producing slower response times than desired.

- **Do your query predicates have identifiable access patterns?** Examine the `WHERE` clauses of your query workload and look for table columns that are consistently used to access data. For example, if most of your queries tend to look up records by date, then a monthly or weekly date-partitioning design might be beneficial. Or if you tend to access records by region, consider a list-partitioning design to divide the table by region.
- **Does your data warehouse maintain a window of historical data?** Another consideration for partition design is your organization's business requirements for maintaining historical data. For example, your data warehouse may only require you to keep the past twelve months worth of data. If the data is partitioned by month, you can easily drop the oldest monthly partition from the warehouse, and load current data into the most recent monthly partition.
- **Can the data be divided into somewhat equal parts based on some defining criteria?** You should choose partitioning criteria that will divide your data as evenly as possible. If the partitions contain a relatively equal number of records, query performance improves based on the number of partitions created. For example, by dividing a large table into 10 partitions, a query will execute 10 times faster than it would against the unpartitioned table (provided that the partitions are designed to support the query's criteria).

Creating Partitioned Tables

A table can only be partitioned at creation time using the `CREATE TABLE` command.

The first step in partitioning a table is to decide on the partition design (date range, numeric range, or list of values) and choose the column(s) on which to partition the table. Decide how many levels of partitions you want. For example, you may want to date range partition a table by month and then further subpartition the monthly partitions by sales region. This section shows examples of SQL syntax for creating a table with various partition designs.

- [Defining Date Range Table Partitions](#)
- [Defining Numeric Range Table Partitions](#)
- [Defining List Table Partitions](#)
- [Defining Multi-level Partitions](#)
- [Partitioning an Existing Table](#)

Defining Date Range Table Partitions

A date range partitioned table uses a single `date` or `timestamp` column as the partition key column. You can use the same partition key column to further subpartition a table if necessary (for example, to partition by month and then subpartition by day). When date partitioning a table, consider partitioning by the most granular level you are interested in. For example, partition by day and have 365 daily partitions, rather than partition by year then subpartition by month then subpartition by day. A multi-level design can reduce query planning time, but a flat partition design will execute faster at query run time.

You can have Greenplum Database automatically generate partitions by giving a `START` value, an `END` value, and an `EVERY` clause that defines the partition increment value. By default, `START` values are always inclusive and `END` values are always exclusive. For example:

```
CREATE TABLE sales (id int, date date, amt decimal(10,2))
DISTRIBUTED BY (id)
PARTITION BY RANGE (date)
( START (date '2008-01-01') INCLUSIVE
  END (date '2009-01-01') EXCLUSIVE
  EVERY (INTERVAL '1 day') );
```

You can also declare and name each partition individually. For example:

```
CREATE TABLE sales (id int, date date, amt decimal(10,2))
DISTRIBUTED BY (id)
PARTITION BY RANGE (date)
( PARTITION Jan08 START (date '2008-01-01') INCLUSIVE ,
  PARTITION Feb08 START (date '2008-02-01') INCLUSIVE ,
  PARTITION Mar08 START (date '2008-03-01') INCLUSIVE ,
  PARTITION Apr08 START (date '2008-04-01') INCLUSIVE ,
  PARTITION May08 START (date '2008-05-01') INCLUSIVE ,
  PARTITION Jun08 START (date '2008-06-01') INCLUSIVE ,
  PARTITION Jul08 START (date '2008-07-01') INCLUSIVE ,
  PARTITION Aug08 START (date '2008-08-01') INCLUSIVE ,
  PARTITION Sep08 START (date '2008-09-01') INCLUSIVE ,
  PARTITION Oct08 START (date '2008-10-01') INCLUSIVE ,
  PARTITION Nov08 START (date '2008-11-01') INCLUSIVE ,
  PARTITION Dec08 START (date '2008-12-01') INCLUSIVE
  END (date '2009-01-01') EXCLUSIVE );
```

Note that you do not need to declare an `END` value for each partition, only the last one. In this example, `Jan08` would end where `Feb08` starts.

Defining Numeric Range Table Partitions

A numeric range partitioned table uses a single numeric data type column as the partition key column. For example:

```
CREATE TABLE rank (id int, rank int, year int, gender
char(1), count int)
DISTRIBUTED BY (id)
PARTITION BY RANGE (year)
( START (2001) END (2008) EVERY (1),
  DEFAULT PARTITION extra );
```

For more information on default partitions, see [“Adding a Default Partition”](#) on page 116.

Defining List Table Partitions

A list partitioned table can use any data type column that allows equality comparisons as its partition key column. A list partition can also have a multi-column (composite) partition key, whereas a range partition only allows a single column as the partition key. For list partitions, you must declare a partition specification for every partition (list value) you want to create. For example:

```
CREATE TABLE rank (id int, rank int, year int, gender
char(1), count int )
DISTRIBUTED BY (id)
PARTITION BY LIST (gender)
( PARTITION girls VALUES ('F'),
  PARTITION boys VALUES ('M'),
  DEFAULT PARTITION other );
```

For more information on default partitions, see [“Adding a Default Partition”](#) on page 116.

Defining Multi-level Partitions

It is possible to create a multi-level partition design where you have subpartitions of partitions. Using a *subpartition template* ensures that every partition has the same subpartition design, even partitions that are added later. For example, to create the two-level partition design illustrated in [Figure 14.1, “Example Multi-level Partition Design”](#) on page 108:

```
CREATE TABLE sales (trans_id int, date date, amount
decimal(9,2), region text)
DISTRIBUTED BY (trans_id)
PARTITION BY RANGE (date)
SUBPARTITION BY LIST (region)
SUBPARTITION TEMPLATE
( SUBPARTITION usa VALUES ('usa'),
  SUBPARTITION asia VALUES ('asia'),
  SUBPARTITION europe VALUES ('europe'),
  DEFAULT SUBPARTITION other_regions)
( START (date '2008-01-01') INCLUSIVE
  END (date '2009-01-01') EXCLUSIVE
  EVERY (INTERVAL '1 month') ),
  DEFAULT PARTITION outlying_dates );
```

Below is a similar example illustrating a three-level partition design where the sales table is partitioned by year, then month, then region. The SUBPARTITION TEMPLATE clauses ensure that each yearly partition has the same subpartition structure. Also note that a DEFAULT partition is declared at each level of the hierarchy:

```
CREATE TABLE sales (id int, year int, month int, day int,
region text)
DISTRIBUTED BY (id)
PARTITION BY RANGE (year)
  SUBPARTITION BY RANGE (month)
```

```

SUBPARTITION TEMPLATE (
  START (1) END (13) EVERY (1),
  DEFAULT SUBPARTITION other_months )
SUBPARTITION BY LIST (region)
  SUBPARTITION TEMPLATE (
    SUBPARTITION usa VALUES ('usa'),
    SUBPARTITION europe VALUES ('europe'),
    SUBPARTITION asia VALUES ('asia'),
    DEFAULT SUBPARTITION other_regions )
( START (2002) END (2010) EVERY (1),
  DEFAULT PARTITION outlying_years );

```

Partitioning an Existing Table

It is not possible to partition a table that has already been created. Tables can only be partitioned at `CREATE TABLE` time. If you have an existing table that you want to partition, you must recreate the table as a partitioned table, reload the data into the newly partitioned table, drop the original table and rename the partitioned table to the original name. You must also regrant any table permissions. For example:

```

CREATE TABLE sales2 (LIKE sales)
PARTITION BY RANGE (date)
( START (date '2008-01-01') INCLUSIVE
  END (date '2009-01-01') EXCLUSIVE
  EVERY (INTERVAL '1 month') );

INSERT INTO sales2 SELECT * FROM sales;
DROP TABLE sales;
ALTER TABLE sales2 RENAME TO sales;
GRANT ALL PRIVILEGES ON sales TO admin;
GRANT SELECT ON sales TO guest;

```

Limitations of Partitioned Tables

- To be able to enforce a primary key or unique constraint, the primary or unique key columns must start with the partitioning key column.

Loading Partitioned Tables

Once you have created your partitioned table structure, top-level parent tables are always empty. Data is routed to the bottom-level child table partitions only. In a multi-level partition design, only the subpartitions at the bottom of the hierarchy can contain data.

If a row cannot be mapped to a child table partition, it will be rejected and the load will fail. If you do not want unmapped rows to be rejected at load time, you can define your partition hierarchy with a `DEFAULT` partition. Any rows that do not match to an existing partition's `CHECK` constraints will then load into the `DEFAULT` partition. See [“Adding a Default Partition”](#) on page 116.

At runtime, the query planner scans the entire table inheritance hierarchy and uses the `CHECK` table constraints to determine which of the child table partitions to scan in order to satisfy the query's conditions. The `DEFAULT` partition (if your hierarchy has one) is always scanned. If the `DEFAULT` partition contains data, this will slow down the overall scan time.

When you use `COPY` or `INSERT` to load data into a parent table, it automatically gets rerouted to the correct partition by default. Therefore, you can load a partitioned table as you would a regular table.

You can also load data into the child table partitions directly if needed. You can also create an intermediate staging table, load it, and then exchange it into your partition design. See [“Exchanging a Partition”](#) on page 116.

Verifying Your Partition Strategy

The purpose for partitioning a table is to reduce the number of rows that must be scanned in order to satisfy a given query. If a table is partitioned based on the query predicate, you can verify that the query planner is selectively scanning the relevant data by using `EXPLAIN` to look at the query plan.

For example, suppose we have a `sales` table that is date-range partitioned by month and subpartitioned by region as illustrated in [Figure 14.1, “Example Multi-level Partition Design”](#) on page 108. For the following query:

```
EXPLAIN SELECT * FROM sales WHERE date='01-07-08' AND
region='usa';
```

The query plan for this query should show a table scan of the following tables only:

- the parent table (`sales`) returning 0-1 rows
- the default partition returning 0-1 rows (if your partition design has one)
- the January 2008 partition (`sales_1_prt_1`) returning 0-1 rows
- the USA region subpartition (`sales_1_2_prt_usa`) returning *some number* of rows.

Below is an example of the relevant portion of the query plan:

```
-> Seq Scan on sales (cost=0.00..0.00 rows=0 width=0)
Filter: "date"=01-07-08::date AND region='USA'::text
-> Seq Scan on sales_1_prt_1 sales (cost=0.00..0.00 rows=0
width=0)
Filter: "date"=01-07-08::date AND region='USA'::text
-> Seq Scan on sales_1_2_prt_usa sales (cost=0.00..9.87
rows=20
width=40)
```

Make sure that the query planner is not scanning unnecessary partitions or subpartitions (for example, scans of other months or regions not specified in the query predicate), and that scans of the top-level tables are returning 0-1 rows.

Some Limitations of Selective Partition Scanning

If the query plan shows that your partition hierarchy is not being selectively scanned, it may be due to one of the following limitations:

- The query planner is only able to selectively scan partitioned tables when the query contains a direct and simple restriction of the table using immutable operators such as:
= < <= > >= <>
- Selective scanning does not currently recognize `STABLE` or `VOLATILE` functions within a query. For example, `WHERE` clauses such as `date > CURRENT DATE` will not cause the query planner to selectively scan partitioned tables because `CURRENT DATE` is a `STABLE` function.

Viewing Your Partition Design

You can look up information about your partition design using the `pg_partitions` view. For example to see the partition design of the `sales` table:

```
SELECT partitionboundary, partitiontablename, partitionname,
partitionlevel, partitionrank FROM pg_partitions WHERE
tablename='sales';
```

There are also the following views that show information about partitioned tables:

- `pg_partition_templates` - Shows subpartitions that were created using a subpartition template.
- `pg_partition_columns` - Shows the partition key columns used in a partition design.

Maintaining Partitioned Tables

You must maintain a partitioned table using the `ALTER TABLE` command against the top-level parent table. The most common scenario is dropping old partitions and adding new ones in order to maintain a rolling window of data in a range partition design. You may also want to convert (*exchange*) older partitions to the append-only compressed storage format in order to save space. If you have a default partition in your partition design, the procedure for adding a new partition is to *split* the default partition.

- [Adding a New Partition](#)
- [Renaming a Partition](#)
- [Adding a Default Partition](#)
- [Dropping a Partition](#)
- [Truncating a Partition](#)
- [Exchanging a Partition](#)
- [Splitting a Partition](#)
- [Modifying a Subpartition Template](#)



Important: When defining and altering partition designs, use the given *partition name*, not the table object name. Although you can query and load any table (including partitioned tables) directly using SQL commands, you can only modify the structure of a partitioned table using the `ALTER TABLE...PARTITION` clauses.

Adding a New Partition

You can add a new partition to an existing partition design using the `ALTER TABLE` command. If the original partition design included subpartitions defined by a *subpartition template*, then the newly added partition will also be subpartitioned according to that template. For example:

```
ALTER TABLE sales ADD PARTITION
    START (date '2009-02-01') INCLUSIVE
    END (date '2009-03-01') EXCLUSIVE;
```

If a subpartition template was not used when you created the table, you would then define subpartitions when adding a new partition:

```
ALTER TABLE sales ADD PARTITION
    START (date '2009-02-01') INCLUSIVE
    END (date '2009-03-01') EXCLUSIVE
    ( SUBPARTITION usa VALUES ('usa'),
      SUBPARTITION asia VALUES ('asia'),
      SUBPARTITION europe VALUES ('europe') );
```

If you want to add a new subpartition to an existing partition, you can specify a particular partition to alter. For example:

```
ALTER TABLE sales ALTER PARTITION FOR (RANK(12))
    ADD PARTITION africa VALUES ('africa');
```



Note: You cannot add a new partition to a partition design that has a default partition. You must split the default partition in order to add a new partition. See “[Splitting a Partition](#)” on page 117.

Renaming a Partition

Partitioned tables are created using the naming convention:

```
<parentname>_<level>_prt_<partition_name>
```

For example:

```
sales_1_prt_jan08
```

Or for auto-generated range partitions (a number is assigned when no name is given):

```
sales_1_prt_1
```

It is not possible to rename a partitioned child table directly by altering the table name. However, you can rename the top-level parent table, and the associated `<parentname>` will change in the table names of all associated child table partitions.

For example:

```
ALTER TABLE sales RENAME TO globalsales;
```

Would change the associated table names accordingly:

```
globalsales_1_prt_1
```

You can also change the partition name of a partition to make it easier to identify. For example:

```
ALTER TABLE sales RENAME PARTITION FOR ('2008-01-01') TO
  jan08;
```

Would change the associated table name accordingly:

```
sales_1_prt_jan08
```

When altering partitioned tables with the `ALTER TABLE` command, they are always referred to by their partition name (*jan08*) and not their full table name (*sales_1_prt_jan08*).

Adding a Default Partition

You can add a default partition to an existing partition design using the `ALTER TABLE` command.

```
ALTER TABLE sales ADD DEFAULT PARTITION other;
```

If your partition design is multi-levelled, then each level in the hierarchy needs a default partition. For example:

```
ALTER TABLE sales ALTER PARTITION FOR (RANK(1)) ADD DEFAULT
PARTITION other;
ALTER TABLE sales ALTER PARTITION FOR (RANK(2)) ADD DEFAULT
PARTITION other;
ALTER TABLE sales ALTER PARTITION FOR (RANK(3)) ADD DEFAULT
PARTITION other;
```

Partition designs that do not have a default partition will reject incoming rows that do not match to an existing partition's `CHECK` constraint. If a partitioned table has a default partition, incoming data that does not match to an existing partition is instead inserted into the default partition.

Dropping a Partition

You can drop a partition from your partition design using the `ALTER TABLE` command. When you drop a partition that has subpartitions, the subpartitions (and all data in them) are automatically dropped as well. For range partitions, it is common to drop the older partitions from the range as old data is rolled out of the data warehouse. For example:

```
ALTER TABLE sales DROP PARTITION FOR (RANK(1));
```

Truncating a Partition

You can truncate a partition using the `ALTER TABLE` command. When you truncate a partition that has subpartitions, the subpartitions are automatically truncated as well.

```
ALTER TABLE sales TRUNCATE PARTITION FOR (RANK(1));
```

Exchanging a Partition

Exchanging a partition involves swapping in another table in place of an existing partition. You can exchange a partition using the `ALTER TABLE` command. You can only exchange partitions at the lowest level of your partition hierarchy (only partitions that contain data can be exchanged).

This can be useful for data loading. For example, you could load a staging table and then swap the loaded table into your partition design. You can also use exchange to change the storage type of older partitions to append-only tables. For example:

```
CREATE TABLE jan08 (LIKE sales) WITH (appendonly=true);
INSERT INTO jan08 SELECT * FROM sales_1_prt_1 ;
ALTER TABLE sales EXCHANGE PARTITION FOR ('2008-01-01') WITH
TABLE jan08;
```

Splitting a Partition

Splitting a partition involves dividing an existing partition into two. You can split a partition using the `ALTER TABLE` command. You can only split partitions at the lowest level of your partition hierarchy (only partitions that contain data can be split). The split value you specify will go into the *latter* partition.

For example, to split a monthly partition into two with the first partition containing dates January 1-15 and the second partition containing dates January 16-31:

```
ALTER TABLE sales SPLIT PARTITION FOR ('2008-01-01')
AT ('2008-01-16')
INTO (PARTITION jan081to15, PARTITION jan0816to31);
```

If your partition design has a default partition, you must split the default partition in order to add a new partition. You can only split default partitions at the lowest level of your partition hierarchy (only default partitions that contain data can be split).

When using the `INTO` clause, the second partition name specified should always be that of the existing default partition. For example, to split a default range partition to add a new monthly partition for January 2009:

```
ALTER TABLE sales SPLIT DEFAULT PARTITION
START ('2009-01-01') INCLUSIVE
END ('2009-02-01') EXCLUSIVE
INTO (PARTITION jan09, default partition);
```

Modifying a Subpartition Template

Use `ALTER TABLE SET SUBPARTITION TEMPLATE` to modify the subpartition template for an existing partition. After you set a new subpartition template, partitions that you add subsequently will have the new subpartition design. Existing partitions are not modified.

For example, to modify the subpartition design illustrated in [Figure 14.1, “Example Multi-level Partition Design”](#) on page 108:

```
ALTER TABLE sales SET SUBPARTITION TEMPLATE
( SUBPARTITION usa VALUES ('usa'),
  SUBPARTITION asia VALUES ('asia'),
  SUBPARTITION europe VALUES ('europe'),
  SUBPARTITION africa VALUES ('africa')
  DEFAULT SUBPARTITION other );
```

With this example template, when you next add a date-range partition of the table `sales`, it will include the new regional list subpartition for Africa. For example, the following command would create the subpartitions `usa`, `asia`, `europa`, `africa`, and a default partition named `other`:

```
ALTER TABLE sales ADD PARTITION sales_prt_3
  START ('2009-03-01') INCLUSIVE
  END ('2009-04-01') EXCLUSIVE );
```

If you need to remove a subpartition template, use `SET SUBPARTITION TEMPLATE` with empty parentheses. For example, to completely clear the subpartition template used in the above examples:

```
ALTER TABLE sales SET SUBPARTITION TEMPLATE ( )
```

Creating and Using Sequences

Sequences are often used to auto-increment unique ID columns of a table whenever a new record is added.

Creating a Sequence

The `CREATE SEQUENCE` command creates and initializes a new special single-row sequence generator table with the given sequence name. The sequence name must be distinct from the name of any other sequence, table, index, or view in the same schema. For example:

```
CREATE SEQUENCE myserial START 101;
```

Using a Sequence

Once you have created a sequence generator table using `CREATE SEQUENCE`, you can use the `nextval` function to operate on the sequence. For example, to insert a row into a table that gets the next value of a sequence:

```
INSERT INTO vendors VALUES (nextval('myserial'), 'acme');
```

You can also use the `setval` function to operate on a sequence to reset a sequence's counter value. For example:

```
SELECT setval('myserial', 201);
```

A `nextval` operation is never rolled back. Once a value has been fetched it is considered used, even if the transaction that did the `nextval` later aborts. This means that aborted transactions may leave unused holes in the sequence of assigned values. `setval` operations are never rolled back, either.

Note that the `nextval` function is currently not allowed in `UPDATE` or `DELETE` statements if mirroring is enabled, and the `currval` and `lastval` functions are currently not supported in Greenplum Database.

To then examine the current settings of a sequence you can simply query the sequence table directly:

```
SELECT * FROM myserial;
```

Altering a Sequence

The `ALTER SEQUENCE` command changes the parameters of an existing sequence generator. For example:

```
ALTER SEQUENCE myserial RESTART WITH 105;
```

Any parameters not specifically set in the `ALTER SEQUENCE` command retain their prior settings.

Dropping a Sequence

The `DROP SEQUENCE` command removes a sequence generator table. For example:

```
DROP SEQUENCE myserial;
```

Using Indexes in Greenplum Database

In most traditional databases, indexes can greatly improve data access times. However, in a distributed database such as Greenplum, indexes should be used more sparingly. Greenplum Database is very fast at sequential scanning (indexes use a random seek pattern to locate records on disk). Also, unlike a traditional database, the data is distributed across the segments. This means each segment scans a smaller portion of the overall data in order to get the result. If using table partitioning, the total data to scan may be even a fraction of that.

Greenplum recommends that you first try your query workload without adding any additional indexes. Indexes are more likely to improve performance for OLTP type workloads, where the query is returning a single record or a very small data set. Typically, a business intelligence (BI) query workload returns very large data sets, and thus does not make efficient use of indexes.

Note that Greenplum Database will automatically create `PRIMARY KEY` indexes for tables with primary keys. Indexes are not allowed on append-only tables. If you want to create an index on a partitioned table, you must index each partitioned child table directly. Indexes on the parent table are not passed down to child table partitions.

Indexes do add some database overhead — they take up storage space and have to be maintained whenever the table is updated. Make sure that the indexes you create are actually being used by your query workload. Also, check to see that the indexes you add are indeed improving query performance (as compared to a sequential scan of the table). You can look at the `EXPLAIN` plans for a query to determine if indexes are being used. See “[Query Profiling](#)” on page 148.

Some other general considerations when creating indexes are:

- **Your Query Workload.** Indexes are more likely to improve performance for OLTP type workloads, where the query is returning a single record or a very small data set. Typically, a business intelligence (BI) query workload returns very large data sets, and thus does not make efficient use of indexes. For this type of workload, it is better to use sequential scans to locate large chunks of data on disk rather than to randomly seek the disk using index scans.

- **Avoid indexes on frequently updated columns.** Creating an index on a column that is frequently updated increases the amount of writes required when the column is updated.
- **Create selective B-tree indexes.** Index selectivity is a ratio of the number of distinct values a column has divided by the number of rows in a table. For example, if a table has 1000 rows and a column has 800 distinct values, the selectivity of the index is 0.8, which is considered good. Unique indexes always have a selectivity ratio of 1.0, which is the best possible. Note that unique indexes are allowed only if the columns of the index key are the same as (or a superset of) the Greenplum distribution key.
- **Use Bitmap indexes for low selectivity columns.** Greenplum Database has an additional index type called a Bitmap index, which is not available in regular PostgreSQL. See “[About Bitmap Indexes](#)” on page 121 for details.
- **Index columns used in joins.** An index on a column used for frequent joins (such as a foreign key column) may improve join performance, as it can enable additional join methods for the query planner to use.
- **Index columns frequently used in predicates.** For queries on large tables, examine the WHERE predicates for the columns that are referenced most often. These may be good candidates for indexes.
- **Avoid overlapping indexes.** Overlapping indexes (those that have the same leading column) are redundant and unnecessary.
- **Drop indexes for bulk loads.** For mass loads of data into a table, consider dropping the indexes and re-creating them after the load is complete. This is often faster than updating the indexes.
- **Consider a clustered index.** Clustering an index means that the records are physically ordered on disk according to the index. If the records you need are distributed randomly on disk, then the database has to seek across the disk to get the records requested. If those records are stored more closely together, then the fetching from disk is more sequential. A good example for a clustered index is on a date column where the data is ordered sequentially by date. A query against a specific date range will result in an ordered fetch from the disk, which leverages fast sequential access.

To cluster an index in Greenplum Database

For very large tables, using the `CLUSTER` command to physically reorder a table based on an index can take an extremely long time. To achieve the same results much faster, you can manually reorder the data on disk by creating an intermediate table and loading the data in the desired order. For example:

```
CREATE TABLE new_table (LIKE old_table)
    AS SELECT * FROM old_table ORDER BY myixcolumn;
DROP old_table;
ALTER TABLE new_table RENAME TO old_table;
CREATE INDEX myixcolumn_ix ON old_table;
VACUUM ANALYZE old_table;
```

Index Types

Greenplum Database provides the following index types as does PostgreSQL: B-tree, Hash, GiST and GIN. Each index type uses a different algorithm that is best suited to different types of queries. By default, the `CREATE INDEX` command will create a B-tree index, which fits the most common situations. See [Index Types](#) in the PostgreSQL documentation for a description of these types.



Note: Greenplum Database has some special considerations concerning unique indexes on a table. Unique indexes are allowed only if the columns of the index key are the same as (or a superset of) the Greenplum distribution key.

About Bitmap Indexes

In addition to the index types provided by PostgreSQL, Greenplum Database has an additional Bitmap index type. Bitmap indexes are one of the most promising strategies for indexing high dimensional data in data warehousing applications and decision support systems. These types of applications typically have large amounts of data and ad hoc queries, but a low number of DML transactions.

An index provides pointers to the rows in a table that contain a given key value. A regular index stores a list of tuple ids for each key corresponding to the rows with that key value. In a bitmap index, a bitmap for each key value replaces a list of tuple ids.

Fully indexing a large table with a traditional B-tree index can be expensive in terms of space because the indexes can be several times larger than the data in the table. Bitmap indexes are typically only a fraction of the size of the indexed data in the table.

Each bit in the bitmap corresponds to a possible tuple id, and if the bit is set, it means that the row with the corresponding tuple id contains the key value. A mapping function converts the bit position to an actual tuple id, so that the bitmap index provides the same functionality as a regular index. Bitmap indexes store the bitmaps in a compressed way. If the number of distinct key values is small, bitmap indexes compress better and the space saving benefit compared to a B-tree index becomes even better.

Bitmap indexes are most effective for queries that contain multiple conditions in the `WHERE` clause. Rows that satisfy some, but not all, conditions are filtered out before the table itself is accessed. This improves response time, often dramatically.

When to Use Bitmap Indexes

Bitmap indexes perform best for columns in which the ratio of the number of distinct values to the number of rows in the table is small. This ratio is known as the degree of *cardinality*. A marital status column, which has only a few distinct values (single, married, divorced, or widowed), is a good choice for a bitmap index, and is considered *low cardinality*.

However, columns with higher cardinalities can also have bitmap indexes. For example, on a table with one million rows, a column with 10,000 distinct values is also a good candidate for a bitmap index. A bitmap index on this column can outperform a B-tree index, particularly when this column is often queried in

conjunction with other indexed columns. However, the performance gains start to diminish on columns with 5,000 or more unique values, regardless of the number of rows in the table.

Bitmap indexes can dramatically improve query performance for ad hoc queries. `AND` and `OR` conditions in the `WHERE` clause of a query can be resolved quickly by performing the corresponding Boolean operations directly on the bitmaps before converting the resulting bitmap to tuple ids. If the resulting number of rows is small, the query can be answered quickly without resorting to a full table scan.

When Not to Use Bitmap Indexes

Bitmap indexes should not be used for unique columns or columns with high cardinality data, such as customer names or phone numbers. B-tree indexes are a better choice for these types of columns.

Bitmap indexes are primarily intended for data warehousing applications where users query the data rather than update it. They are not suitable for OLTP applications with large numbers of concurrent transactions modifying the data.

Creating an Index

The `CREATE INDEX` command defines a new index on a table. By default, a B-tree index is created if you do not declare an index type. For example, to create a B-tree index on the column `title` in the table `films`:

```
CREATE INDEX title_idx ON films (title);
```

To create a bitmap index on the column `gender` in the table `employee`:

```
CREATE INDEX gender_bmp_idx ON employee USING bitmap
(gender);
```

Examining Index Usage

Although indexes in Greenplum Database do not need maintenance and tuning, it is still important to check which indexes are actually used by the real-life query workload. Examining index usage for an individual query is done with the `EXPLAIN` command.

The query plan shows the different steps or *plan nodes* that the database will take to answer a particular query, along with time estimates for each plan node. To examine the use of indexes, look for the following query plan node types in your `EXPLAIN` output:

- **Index Scan** - A scan of an index.
- **Bitmap Heap Scan** - Retrieves all rows from the bitmap generated by `BitmapAnd`, `BitmapOr`, or `BitmapIndexScan` and accesses the heap to retrieve the relevant rows.
- **Bitmap Index Scan** - Compute a bitmap by OR-ing all bitmaps that satisfy the query predicates from the underlying index.
- **BitmapAnd** or **BitmapOr** - Takes the bitmaps generated from multiple `BitmapIndexScan` nodes, ANDs or ORs them together, and generates a new bitmap as its output.

It is difficult to formulate a general procedure for determining which indexes to set up. A good deal of experimentation will be necessary in most cases.

- Always run `ANALYZE` after creating or updating an index. This command collects table statistics that are used by the query planner. This information is required to guess the number of rows returned by a query, which is needed by the planner to assign realistic costs to each possible query plan.
- Use real data for experimentation. Using test data for setting up indexes will tell you what indexes you need for the test data, but that is all.
- It is especially fatal to use very small test data sets. While selecting 1000 out of 1,000,000 rows could be a candidate for an index, selecting 1 out of 100 rows will hardly be, because the 100 rows will probably fit within a single disk page, and there is no plan that can beat sequentially fetching one disk page.
- Also be careful when making up test data, which is often unavoidable when the application is not in production use yet. Values that are very similar, completely random, or inserted in sorted order will skew the statistics away from the distribution that real data would have.
- When indexes are not used, it can be useful for testing to force their use. There are run-time parameters that can turn off various plan types. For instance, turning off sequential scans (`enable_seqscan`) and nested-loop joins (`enable_nestloop`), which are the most basic plans, will force the system to use a different plan. Time your query with and without indexes. The `EXPLAIN ANALYZE` command can be useful here.

Managing Indexes

In certain conditions, a poorly performing index may need to be rebuilt using the `REINDEX` command. This rebuilds an index using the data stored in the index's table, replacing the old copy of the index.

Updates or delete operations do not update bitmap indexes. Therefore, if you have deleted rows or updated columns in a table that has bitmap indexes, you will need to rebuild the indexes using the `REINDEX` command.

To rebuild all indexes on a table

```
REINDEX my_table;
```

To rebuild a particular index

```
REINDEX my_index;
```

Dropping an Index

The `DROP INDEX` command removes an index. For example:

```
DROP INDEX title_idx;
```

When loading data, it is often faster to drop all indexes, load, then recreate the indexes afterwards.

Creating and Managing Views

Views are a way to save frequently used or complex queries and then access them in a `SELECT` statement as if they were a table. A view is not physically materialized on disk as is a table. The query is instead ran as a subquery whenever the view is accessed.

Creating Views

The `CREATE VIEW` command defines a view of a query. For example:

```
CREATE VIEW comedies AS SELECT * FROM films WHERE kind =
'comedy';
```

Note that currently views ignore `ORDER BY` or `SORT` operations stored in the view.

Dropping Views

The `DROP VIEW` command removes a view. For example:

```
DROP VIEW topten;
```

15. Managing Data

This chapter provides information about managing data and concurrent access in Greenplum Database. It contains the following topics:

- [About Concurrency Control in Greenplum Database](#)
- [Inserting New Rows](#)
- [Updating Existing Rows](#)
- [Deleting Rows](#)
- [Working With Transactions](#)
- [Vacuuming the Database](#)

About Concurrency Control in Greenplum Database

Unlike traditional database systems which use locks for concurrency control, Greenplum Database (as does PostgreSQL) maintains data consistency by using a multiversion model (Multiversion Concurrency Control, MVCC). This means that while querying a database, each transaction sees a snapshot of data which protects the transaction from viewing inconsistent data that could be caused by (other) concurrent updates on the same data rows. This provides transaction isolation for each database session.

MVCC, by eschewing explicit locking methodologies of traditional database systems, minimizes lock contention in order to allow for reasonable performance in multiuser environments. The main advantage to using the MVCC model of concurrency control rather than locking is that in MVCC locks acquired for querying (reading) data do not conflict with locks acquired for writing data, and so reading never blocks writing and writing never blocks reading.

Greenplum Database provides various lock modes to control concurrent access to data in tables. Most Greenplum Database SQL commands automatically acquire locks of appropriate modes to ensure that referenced tables are not dropped or modified in incompatible ways while the command executes. For applications that cannot adapt easily to MVCC behavior, the `LOCK` command can be used to acquire explicit locks. However, proper use of MVCC will generally provide better performance than locks.

Table 15.1 Lock Modes in Greenplum Database

Lock Mode	Associated SQL Commands	Conflicts With
ACCESS SHARE	SELECT	ACCESS EXCLUSIVE
ROW SHARE	N/A	EXCLUSIVE, ACCESS EXCLUSIVE
ROW EXCLUSIVE	INSERT, COPY	SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE, ACCESS EXCLUSIVE
SHARE UPDATE EXCLUSIVE	VACUUM (without FULL), ANALYZE, CREATE INDEX CONCURRENTLY	SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE, ACCESS EXCLUSIVE

Table 15.1 Lock Modes in Greenplum Database

Lock Mode	Associated SQL Commands	Conflicts With
SHARE	CREATE INDEX	ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE ROW EXCLUSIVE, EXCLUSIVE, ACCESS EXCLUSIVE
SHARE ROW EXCLUSIVE		ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE, ACCESS EXCLUSIVE
EXCLUSIVE ¹	UPDATE, DELETE, SELECT FOR UPDATE, SELECT FOR SHARE	ROW SHARE, ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE, ACCESS EXCLUSIVE
ACCESS EXCLUSIVE	ALTER TABLE, DROP TABLE, TRUNCATE, REINDEX, CLUSTER, VACUUM FULL	ACCESS SHARE, ROW SHARE, ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE, ACCESS EXCLUSIVE

1. Greenplum Database takes a more restrictive lock on UPDATE, SELECT FOR UPDATE/SHARE and DELETE operations than regular PostgreSQL.

Inserting New Rows

When a table is created, it contains no data. The first thing to do before a database can be of much use is to insert data. To create a new row, use the `INSERT` command. This command requires the table name and a value for each of the columns of the table. The data values are listed in the order in which the columns appear in the table, separated by commas. For example:

```
INSERT INTO products VALUES (1, 'Cheese', 9.99);
```

If you do not know the order of the columns in the table, you can also list the columns explicitly. Many users consider it good practice to always list the column names. For example:

```
INSERT INTO products (name, price, product_no) VALUES
('Cheese', 9.99, 1);
```

Usually, the data values will be literals (constants), but scalar expressions are also allowed. For example:

```
INSERT INTO films SELECT * FROM tmp_films WHERE date_prod <
'2004-05-07';
```

You can also insert multiple rows in a single command. For example:

```
INSERT INTO products (product_no, name, price) VALUES
(1, 'Cheese', 9.99),
(2, 'Bread', 1.99),
(3, 'Milk', 2.99);
```

When inserting a lot of data at the same time, considering using external tables (`CREATE EXTERNAL TABLE`) or the `COPY` command. These load mechanisms are more efficient than `INSERT` when loading a large number of rows. See “[Parallel Data Loading](#)” on page 153 for more information about bulk data loading.

The storage model of append-only tables is optimized for bulk data loading. Single row `INSERT` statements are not recommended for append-only tables.

Updating Existing Rows

The term *update* refers to the modification of data that is already in the database. You can update individual rows, all the rows in a table, or a subset of all rows. Each column can be updated separately without affecting the other columns.

To perform an update, you need three pieces of information:

1. The name of the table and columns to update,
2. The new values of the columns,
3. The row(s) to update by specifying the conditions a row must meet in order to be updated.

The `UPDATE` command updates rows in a table. For example, this command updates all products that have a price of 5 to have a price of 10:

```
UPDATE products SET price = 10 WHERE price = 5;
```

Using `UPDATE` in Greenplum Database has the following restrictions:

- The Greenplum distribution key columns may not be updated.
- If the `WHERE` clause joins two or more tables to perform the update, the tables must have the same Greenplum distribution key and be joined on the distribution key column(s). This is referred to as an *equijoin*.
- Cannot use `STABLE` or `VOLATILE` functions in an `UPDATE` statement if mirrors are enabled.
- The `RETURNING` clause is not supported in Greenplum Database.

Deleting Rows

The `DELETE` command deletes rows that satisfy the `WHERE` clause from the specified table. If the `WHERE` clause is absent, the effect is to delete all rows in the table. The result is a valid, but empty table. For example, to remove all rows from the products table that have a price of 10:

```
DELETE FROM products WHERE price = 10;
```

Or to delete all rows from a table:

```
DELETE FROM products;
```

Using `DELETE` in Greenplum Database has the following restrictions:

- If the `WHERE` clause joins two or more tables to perform the update, the tables must have the same Greenplum distribution key and be joined on the distribution key column(s). This is referred to as an *equijoin*.
- Cannot use `STABLE` or `VOLATILE` functions in a `DELETE` statement if mirrors are enabled.
- The `RETURNING` clause is not supported in Greenplum Database.

Truncating a Table

If you want to quickly remove all rows in a table, consider using the `TRUNCATE` command. For example:

```
TRUNCATE mytable;
```

This command empties the table of all rows in one operation. Note that `TRUNCATE` does not scan the table, therefore it does not process inherited child tables or `ON DELETE` rewrite rules. Only rows in the named table will be truncated.

Working With Transactions

Transactions allow you to bundle together multiple SQL statements in one all-or-nothing operation.

The SQL commands used to perform transactions in Greenplum Database are:

- `BEGIN` or `START TRANSACTION` to start a transaction block.
- `END` or `COMMIT` to commit the results of the transaction.
- `ROLLBACK` to abandon the transaction without making any changes.
- `SAVEPOINT` to allow you to selectively discard parts of the transaction, while committing the rest. After defining a savepoint with `SAVEPOINT`, you can if needed roll back to the savepoint with `ROLLBACK TO SAVEPOINT`. To destroy a savepoint within a transaction, use `RELEASE SAVEPOINT`.

Transaction Isolation Levels

The SQL standard defines four transaction isolation levels. In Greenplum Database, you can request any of the four standard transaction isolation levels. But internally, there are only two distinct isolation levels — *read committed* and *serializable*:

- **read committed** — When a transaction runs on this isolation level, a `SELECT` query sees only data committed before the query began. It never sees either uncommitted data or changes committed during query execution by concurrent transactions. However, the `SELECT` does see the effects of previous updates executed within its own transaction, even though they are not yet committed. In effect, a `SELECT` query sees a snapshot of the database as of the instant that query begins to run. Notice that two successive `SELECT` commands can see different data, even though they are within a single transaction, if other transactions commit changes during execution of the first `SELECT`. `UPDATE` and `DELETE` commands behave the same as `SELECT` in terms of searching for target rows. They will only find target rows that were committed as of the command start time. However, such a target row may have already been updated (or deleted or locked) by another concurrent transaction by the time it is found. The partial transaction isolation provided by read committed mode is adequate for many applications, and this mode is fast and simple to use. However, for applications that do complex queries and updates, it may be necessary to guarantee a more rigorously consistent view of the database than the read committed mode provides.
- **serializable** — This is the strictest transaction isolation. This level emulates serial transaction execution, as if transactions had been executed one after another, serially, rather than concurrently. Applications using this level must be prepared to retry transactions due to serialization failures. When a transaction is on the serializable level, a `SELECT` query sees only data committed before the transaction began. It never sees either uncommitted data or changes committed during transaction execution by concurrent transactions. However, the `SELECT` does see the effects of previous updates executed within its own transaction, even though they are not yet committed. Successive `SELECT` commands within a single transaction always see the same data. `UPDATE` and `DELETE` commands behave the same as `SELECT` in terms of searching for target rows. They will only find target rows that were committed as of the transaction start time. However, such a target row may have already been updated (or deleted or locked) by another concurrent transaction by the time it is found. In this case, the serializable transaction will wait for the first updating transaction to commit or roll back (if it is still in progress). If the first updater rolls back, then its effects are negated and the serializable transaction can proceed with updating the originally found row. But if the first updater commits (and actually updated or deleted the row, not just locked it) then the serializable transaction will be rolled back.
- **read uncommitted** — Treated the same as *read committed* in Greenplum Database.
- **repeatable read** — Treated the same as *serializable* in Greenplum Database.

The default transaction isolation level in Greenplum Database is *read committed*. To change the isolation level for a transaction, you can declare the isolation level when you `BEGIN` the transaction, or else use the `SET TRANSACTION` command after the transaction is started.

Vacuuming the Database

Because of the MVCC transaction concurrency model, data rows that are deleted or updated still occupy physical space on disk even though they are not visible to any new transactions. If you have a database with lots of updates and deletes, you will generate a lot of expired rows. Periodically running the `VACUUM` command will remove these expired rows. For example:

```
VACUUM mytable;
```

The `VACUUM` command also collects table-level statistics such as number of rows and pages, so it is necessary to vacuum all tables after loading data, including append-only tables. Recommended routine vacuum operations are described in “[Routine Vacuum and Analyze](#)” on page 227.

Configuring the Free Space Map

Expired rows are held in what is called the *free space map*. The free space map must be sized large enough to cover the expired rows of all tables in your database. If not sized large enough, space occupied by expired rows that overflow the free space map cannot be reclaimed by a regular `VACUUM` command.

A `VACUUM FULL` will reclaim all expired row space, but is a very expensive operation and may take an unacceptably long time to finish on large, distributed Greenplum Database tables. If you do get into a situation where the free space map has overflowed, it may be more timely to recreate the table with a `CREATE TABLE AS` statement and drop the old table. A `VACUUM FULL` is not recommended in Greenplum Database.

It is best to size the free space map appropriately. The free space map is configured with the following server configuration parameters:

```
max_fsm_pages  
max_fsm_relations
```

16. Querying Data

This chapter describes the use of the SQL language in Greenplum Database. SQL commands are typically entered using the standard PostgreSQL interactive terminal `psql`, but other programs that have similar functionality can be used as well.

- [Defining Queries](#)
- [Using Functions and Operators](#)
- [Query Profiling](#)

Defining Queries

A query is a SQL command that views, changes or analyzes the data in a database. This section describes how to construct SQL queries in Greenplum Database.

- [SQL Lexicon](#)
- [SQL Value Expressions](#)

SQL Lexicon

SQL (structured query language) is the language used to access the database. The SQL language has a specific lexicon (words, special characters, etc.) used to construct queries or commands that the database engine can understand.

SQL input consists of a sequence of commands. A command is composed of a sequence of tokens, terminated by a semicolon (;). Which tokens are valid depends on the syntax of the particular command. The syntax rules for each command are described in “[SQL Command Reference](#)” on page 259.

Greenplum Database is based on PostgreSQL and adheres to the same SQL structure and syntax (with some minor exceptions). In most cases, the syntax is identical to PostgreSQL, however some commands may have additional or restricted syntax in Greenplum Database. For a complete explanation of the SQL rules and concepts as implemented in PostgreSQL, see [SQL Syntax](#) in the PostgreSQL documentation.

SQL Value Expressions

Value expressions are used in a variety of contexts, such as in the target list of the `SELECT` command, as new column values in `INSERT` or `UPDATE`, or in search conditions in a number of commands. The result of a value expression is sometimes called a *scalar*, to distinguish it from the result of a table expression (which is a table). Value expressions are therefore also called scalar expressions (or even simply expressions). The expression syntax allows the calculation of values from primitive parts using arithmetic, logical, set, and other operations.

A value expression is one of the following:

- A constant or literal value.

- A column reference.
- A positional parameter reference, in the body of a function definition or prepared statement.
- A subscripted expression.
- A field selection expression.
- An operator invocation.
- A function call.
- An aggregate expression.
- A window expression.
- A type cast.
- A scalar subquery.
- An array constructor.
- A row constructor.
- Another value expression in parentheses, useful to group subexpressions and override precedence.

In addition to this list, there are a number of constructs that can be classified as an expression but do not follow any general syntax rules. These generally have the semantics of a function or operator and are explained in [“Using Functions and Operators” on page 140](#).

Column References

A column can be referenced in the form:

```
correlation.columnname
```

Where *correlation* is the name of a table (possibly qualified with a schema name), or an alias for a table defined by means of a `FROM` clause, or one of the key words `NEW` or `OLD`. (`NEW` and `OLD` can only appear in rewrite rules, while other correlation names can be used in any SQL statement.) The correlation name and separating dot may be omitted if the column name is unique across all the tables being used in the current query.

Positional Parameters

A positional parameter reference is used to indicate a value that is supplied externally to an SQL statement. Parameters are used in SQL function definitions and in prepared queries. Some client libraries also support specifying data values separately from the SQL command string, in which case parameters are used to refer to the out-of-line data values. The form of a parameter reference is:

```
$number
```

For example, consider the definition of a function, *dept*, as:

```
CREATE FUNCTION dept(text) RETURNS dept
AS $$ SELECT * FROM dept WHERE name = $1 $$
LANGUAGE SQL;
```

Here the `$1` references the value of the first function argument whenever the function is invoked.

Subscripts

If an expression yields a value of an array type, then a specific element of the array value can be extracted by writing:

```
expression[subscript]
```

Or multiple adjacent elements (an ‘array slice’) can be extracted by writing:

```
expression[lower_subscript:upper_subscript]
```

Here, the brackets [] are meant to appear literally. Each subscript is itself an expression, which must yield an integer value.

In general the array expression must be parenthesized, but the parentheses may be omitted when the expression to be subscripted is just a column reference or positional parameter. Also, multiple subscripts can be concatenated when the original array is multidimensional. For example:

```
mytable.arraycolumn[4]
mytable.two_d_column[17][34]
$1[10:42]
(arrayfunction(a,b))[42]
```

The parentheses in the last example are required.

Field Selection

If an expression yields a value of a composite type (row type), then a specific field of the row can be extracted by writing:

```
expression.fieldname
```

In general the row *expression* must be parenthesized, but the parentheses may be omitted when the expression to be selected from is just a table reference or positional parameter. For example:

```
mytable.mycolumn
$1.somecolumn
(rowfunction(a,b)).col3
```

Thus, a qualified column reference is actually just a special case of the field selection syntax.

Operator Invocations

There are three possible syntaxes for an operator invocation:

```
expression operator expression (binary infix operator)
operator expression (unary prefix operator)
expression operator (unary postfix operator)
```

Where *operator* is an operator token, one of the key words AND, OR, or NOT, or is a qualified operator name in the form:

```
OPERATOR(schema.operatorname)
```

Which particular operators exist and whether they are unary or binary depends on what operators have been defined by the system or the user. “[Built-in Functions and Operators](#)” on page 141 describes the built-in operators.

Function Calls

The syntax for a function call is the name of a function (possibly qualified with a schema name), followed by its argument list enclosed in parentheses:

```
function ([expression [, expression ... ] ] )
```

For example, the following function call computes the square root of 2:

```
sqrt(2)
```

The list of built-in functions is listed in “[Built-in Functions and Operators](#)” on page 141. Other functions may be added by the user.

Aggregate Expressions

An aggregate expression represents the application of an aggregate function across the rows selected by a query. An aggregate function reduces multiple inputs to a single output value, such as the sum or average of the inputs. The syntax of an aggregate expression is one of the following:

```
aggregate_name (expression [ , ... ] ) [FILTER (WHERE condition)]
aggregate_name (ALL expression [ , ... ] ) [FILTER (WHERE condition)]
aggregate_name (DISTINCT expression [ , ... ] ) [FILTER (WHERE condition)]
aggregate_name ( * ) [FILTER (WHERE condition)]
```

Where *aggregate_name* is a previously defined aggregate (possibly qualified with a schema name), and *expression* is any value expression that does not itself contain an aggregate expression.

The first form of aggregate expression invokes the aggregate across all input rows for which the given expression(s) yield non-null values. The second form is the same as the first, since ALL is the default. The third form invokes the aggregate for all distinct non-null values of the expressions found in the input rows. The last form invokes the aggregate once for each input row regardless of null or non-null values; since no particular input value is specified, it is generally only useful for the `count(*)` aggregate function.

For example, `count(*)` yields the total number of input rows; `count(f1)` yields the number of input rows in which `f1` is non-null; `count(distinct f1)` yields the number of distinct non-null values of `f1`.

The FILTER clause allows you to specify a condition to limit the input rows to the aggregate function. For example:

```
SELECT count(*) FILTER (WHERE gender='F') FROM employee;
```

The `WHERE condition` of the `FILTER` clause cannot contain a set returning function, subquery, a window function, or an outer reference. If using a user-defined aggregate function, the state transition function must be declared as `STRICT` (see [CREATE AGGREGATE](#)).

The predefined aggregate functions are described in “[Aggregate Functions](#)” on page 142. Other aggregate functions may be added by the user.

An aggregate expression may only appear in the result list or `HAVING` clause of a `SELECT` command. It is forbidden in other clauses, such as `WHERE`, because those clauses are logically evaluated before the results of aggregates are formed.

When an aggregate expression appears in a subquery (see “[Scalar Subqueries](#)” on page 137 and “[Subquery Expressions](#)” on page 142), the aggregate is normally evaluated over the rows of the subquery. But an exception occurs if the aggregate’s arguments contain only outer-level variables: the aggregate then belongs to the nearest such outer level, and is evaluated over the rows of that query. The aggregate expression as a whole is then an outer reference for the subquery it appears in, and acts as a constant over any one evaluation of that subquery. The restriction about appearing only in the result list or `HAVING` clause applies with respect to the query level that the aggregate belongs to.

Greenplum Database currently does not support `DISTINCT` with more than one input expression.

Window Expressions

Window expressions allow application developers to more easily compose complex online analytical processing (OLAP) queries using standard SQL commands. For example, moving averages or sums can be calculated over various intervals; aggregations and ranks can be reset as selected column values change; and complex ratios can be expressed in simple terms.

A window expression represents the application of a *window function* applied to a *window frame*, which is defined in a special `OVER()` clause. A window partition is a set of rows that are grouped together for the purpose of applying an window function. Unlike aggregate functions, which return a result value for each group of rows, window functions return a result value for every row, but that value is calculated with respect to the rows in a particular window partition. If no partition is specified, the window function is computed over the complete intermediate result set.

The syntax of a window expression is:

```

window_function ( [expression [, ...]] ) OVER (
  window_specification )

```

Where *window_function* is one of the functions listed in “[Window Functions](#)” on page 143, *expression* is any value expression that does not itself contain a window expression, and *window_specification* is:

```

[window_name]
[PARTITION BY expression [, ...]]
[[ORDER BY expression [ASC | DESC | USING operator] [, ...]]
  [{RANGE | ROWS}
   { UNBOUNDED PRECEDING

```

```

| expression PRECEDING
| CURRENT ROW
| BETWEEN window_frame_bound AND window_frame_bound ]]]

```

and where *window_frame_bound* can be one of:

```

UNBOUNDED PRECEDING
expression PRECEDING
CURRENT ROW
expression FOLLOWING
UNBOUNDED FOLLOWING

```

A window expression may only appear in the select list of a `SELECT` command. For example:

```

SELECT count(*) OVER(PARTITION BY customer_id), * FROM
sales;

```

The `OVER` clause is what differentiates window functions from other aggregate or reporting functions. The `OVER` clause defines the *window_specification* to which the window function is applied. A window specification has the following characteristics:

- The `PARTITION BY` clause, which defines the window partitions to which the window function is applied. If omitted, the entire result set is treated as one partition.
- The `ORDER BY` clause defines the expression(s) for sorting rows within a window partition. Note that the `ORDER BY` of a window specification is separate and distinct from the `ORDER BY` clause of a regular query expression (see “[The ORDER BY Clause](#)” on page 532). The `ORDER BY` clause is required for the window functions that calculate rankings, as it identifies the measure(s) for the ranking values. For OLAP aggregations, the `ORDER BY` clause is required in order to use window frames (the `ROWS | RANGE` clause).
NOTE: Columns of data types that lack a coherent ordering, such as `time`, are not good candidates for use in the `ORDER BY` clause of a window specification. `Time`, with or without time zone, lacks a coherent ordering because addition and subtraction do not have the expected effects. For example, the following is not generally true: `x::time < x::time + '2 hour'::interval`
- The `ROWS/RANGE` clause is used to define a window frame for aggregate (non-ranking) window functions. A window frame defines a set of rows within a window partition. When a window frame is defined, the window function is computed with respect to the contents of this moving frame rather than the fixed contents of the entire window partition. Window frames can be row-based (`ROWS`) or value-based (`RANGE`).

Type Casts

A type cast specifies a conversion from one data type to another. Greenplum Database (as does PostgreSQL) accepts two equivalent syntaxes for type casts:

```

CAST ( expression AS type )
expression::type

```

The `CAST` syntax conforms to SQL; the syntax with `::` is historical PostgreSQL usage.

When a cast is applied to a value expression of a known type, it represents a run-time type conversion. The cast will succeed only if a suitable type conversion function has been defined. Notice that this is subtly different from the use of casts with constants. A cast applied to an unadorned string literal represents the initial assignment of a type to a literal constant value, and so it will succeed for any type (if the contents of the string literal are acceptable input syntax for the data type).

An explicit type cast may usually be omitted if there is no ambiguity as to the type that a value expression must produce (for example, when it is assigned to a table column); the system will automatically apply a type cast in such cases. However, automatic casting is only done for casts that are marked “OK to apply implicitly” in the system catalogs. Other casts must be invoked with explicit casting syntax. This restriction is intended to prevent surprising conversions from being applied silently.

Scalar Subqueries

A scalar subquery is an ordinary `SELECT` query in parentheses that returns exactly one row with one column. The `SELECT` query is executed and the single returned value is used in the surrounding value expression. It is an error to use a query that returns more than one row or more than one column as a scalar subquery. The subquery can refer to variables from the surrounding query, which will act as constants during any one evaluation of the subquery. For example, the following finds the largest city population in each state:

```
SELECT name, (SELECT max(pop) FROM cities WHERE cities.state
= states.name) FROM states;
```

Array Constructors

An array constructor is an expression that builds an array value from values for its member elements. A simple array constructor consists of the key word `ARRAY`, a left square bracket `[`, one or more expressions (separated by commas) for the array element values, and finally a right square bracket `]`. For example,

```
SELECT ARRAY[1,2,3+4];
   array
-----
  {1,2,7}
```

The array element type is the common type of the member expressions, determined using the same rules as for `UNION` or `CASE` constructs.

Multidimensional array values can be built by nesting array constructors. In the inner constructors, the key word `ARRAY` may be omitted. For example, these two `SELECT` statements produce the same result:

```
SELECT ARRAY[ARRAY[1,2], ARRAY[3,4]];
SELECT ARRAY[[1,2],[3,4]];
   array
-----
  {{1,2},{3,4}}
```

Since multidimensional arrays must be rectangular, inner constructors at the same level must produce sub-arrays of identical dimensions.

Multidimensional array constructor elements can be anything yielding an array of the proper kind, not only a sub-ARRAY construct. For example:

```
CREATE TABLE arr(f1 int[], f2 int[]);
INSERT INTO arr VALUES (ARRAY[[1,2],[3,4]],
ARRAY[[5,6],[7,8]]);
SELECT ARRAY[f1, f2, '{{9,10},{11,12}}'::int[]] FROM arr;
           array
-----
{{{1,2},{3,4}},{5,6},{7,8}},{9,10},{11,12}}
```

It is also possible to construct an array from the results of a subquery. In this form, the array constructor is written with the key word ARRAY followed by a parenthesized (not bracketed) subquery. For example:

```
SELECT ARRAY(SELECT oid FROM pg_proc WHERE proname LIKE
'bytea%');
           ?column?
-----
{2011,1954,1948,1952,1951,1244,1950,2005,1949,1953,2006,31}
```

(The subquery must return a single column. The resulting one-dimensional array will have an element for each row in the subquery result, with an element type matching that of the subquery's output column. The subscripts of an array value built with ARRAY always begin with one.

Row Constructors

A row constructor is an expression that builds a row value (also called a composite value) from values for its member fields. A row constructor consists of the key word ROW, a left parenthesis, zero or more expressions (separated by commas) for the row field values, and finally a right parenthesis. For example,

```
SELECT ROW(1,2.5,'this is a test');
```

A row constructor can include the syntax rowvalue.*, which will be expanded to a list of the elements of the row value, just as occurs when the .* syntax is used at the top level of a SELECT list. For example, if table t has columns f1 and f2, these are the same:

```
SELECT ROW(t.*, 42) FROM t;
SELECT ROW(t.f1, t.f2, 42) FROM t;
```

By default, the value created by a ROW expression is of an anonymous record type. If necessary, it can be cast to a named composite type — either the row type of a table, or a composite type created with CREATE TYPE AS. An explicit cast may be needed to avoid ambiguity. For example:

```
CREATE TABLE mytable(f1 int, f2 float, f3 text);
CREATE FUNCTION getf1(mytable) RETURNS int AS 'SELECT $1.f1'
LANGUAGE SQL;
```

-- No cast needed since only one *getf1()* exists

```
SELECT getf1(ROW(1,2.5,'this is a test'));
```

```

    getf1
-----
    1
CREATE TYPE myrowtype AS (f1 int, f2 text, f3 numeric);
CREATE FUNCTION getf1(myrowtype) RETURNS int AS 'SELECT
$1.f1' LANGUAGE SQL;

-- Now we need a cast to indicate which function to call:

SELECT getf1(ROW(1,2.5,'this is a test'));
ERROR:  function getf1(record) is not unique
SELECT getf1(ROW(1,2.5,'this is a test')::mytable);
    getf1
-----
    1
SELECT getf1(CAST(ROW(11,'this is a test',2.5) AS
myrowtype));
    getf1
-----
    11

```

Row constructors can be used to build composite values to be stored in a composite-type table column, or to be passed to a function that accepts a composite parameter.

Expression Evaluation Rules

The order of evaluation of subexpressions is not defined. In particular, the inputs of an operator or function are not necessarily evaluated left-to-right or in any other fixed order.

Furthermore, if the result of an expression can be determined by evaluating only some parts of it, then other subexpressions might not be evaluated at all. For instance, if one wrote:

```
SELECT true OR somefunc();
```

then `somefunc()` would (probably) not be called at all. The same would be the case if one wrote:

```
SELECT somefunc() OR true;
```

Note that this is not the same as the left-to-right ‘forced evaluation order’ of Boolean operators that is found in some programming languages.

As a consequence, it is unwise to use functions with side effects as part of complex expressions. It is particularly dangerous to rely on side effects or evaluation order in `WHERE` and `HAVING` clauses, since those clauses are extensively reprocessed as part of developing an execution plan. Boolean expressions (`AND/OR/NOT` combinations) in those clauses may be reorganized in any manner allowed by the laws of Boolean algebra.

When it is essential to force evaluation order, a `CASE` construct may be used. For example, this is an untrustworthy way of trying to avoid division by zero in a `WHERE` clause:

```
SELECT ... WHERE x <> 0 AND y/x > 1.5;
```

But this is safe:

```
SELECT ... WHERE CASE WHEN x <> 0 THEN y/x > 1.5 ELSE false
END;
```

A `CASE` construct used in this fashion will defeat optimization attempts, so it should only be done when necessary.

Using Functions and Operators

- [Using Functions in Greenplum Database](#)
- [User-Defined Functions](#)
- [Built-in Functions and Operators](#)
- [Window Functions](#)
- [Advanced Analytic Functions](#)

Using Functions in Greenplum Database

A function can be one of three types: `IMMUTABLE`, `STABLE`, or `VOLATILE`. Greenplum Database offers full support of all `IMMUTABLE` functions. An immutable function is a function that relies only on information directly present in its argument list and will always return the same result when given the same argument values.

The use of `STABLE` and `VOLATILE` functions is restricted in Greenplum Database. `STABLE` indicates that within a single table scan the function will consistently return the same result for the same argument values, but that its result could change across SQL statements. Functions whose results depend on database lookups or parameter variables are classified as `STABLE`. Also note that the `current_timestamp` family of functions qualify as stable, since their values do not change within a transaction.

`VOLATILE` indicates that the function value can change even within a single table scan. Relatively few database functions are volatile in this sense; some examples are `random()`, `currval()`, `timeofday()`. But note that any function that has side-effects must be classified volatile, even if its result is quite predictable (for example, `setval()`).

In Greenplum Database, the data is divided up across the segments — each segment is, in a sense, its own distinct PostgreSQL database. To prevent data from becoming out-of-sync across the segments, any function classified as `STABLE` or `VOLATILE` cannot be executed at the segment level if it contains SQL or modifies the database in any way. For example, functions such as `random()` or `timeofday()` are not allowed to execute on distributed data in Greenplum Database because they could potentially cause inconsistent data between the segment instances.

Also, `STABLE` and `VOLATILE` functions cannot be used in `UPDATE` or `DELETE` statements at all if mirrors are enabled. For example, the following statement would not be allowed if you had mirror segments because `UPDATE` and `DELETE` statements are run on the primary segments and the mirror segments independently. The following statement would cause a mirror and its primary to be out of sync:

```
UPDATE mytable SET id = nextval(myseq);
```

To ensure data consistency, `VOLATILE` and `STABLE` functions can safely be used in statements that are evaluated on and execute from the master. For example, the following statements are always executed on the master (statements without a `FROM` clause):

```
SELECT setval('myseq', 201);
SELECT foo();
```

The following statement executes on the segments (has a `FROM` clause), however it would be allowed provided that the function used in the `FROM` clause simply returns a set of rows:

```
SELECT * FROM foo();
```

One exception to this rule are functions that return a table reference (`rangeFuncs`) or functions that use the `refCursor` datatype. These types of functions cannot be used at all in Greenplum Database. However, they are not very commonly used anyway.

User-Defined Functions

Greenplum Database supports user-defined functions as does PostgreSQL. See the section on [Extending SQL](#) in the PostgreSQL documentation for more information. You can use the `CREATE FUNCTION` command to register user-defined functions as long as they are used as described in “[Using Functions in Greenplum Database](#)” on [page 140](#). By default, functions are declared as `VOLATILE`, so it is important to specify the correct volatility level if you are registering a user-defined function that is `IMMUTABLE` or `STABLE`.

When creating user-defined functions, avoid using fatal errors or any kind of destructive call. The Greenplum Database server may respond to such errors with a sudden shutdown or restart.

Note that in Greenplum Database, the shared library files for user-created functions must reside in the same library path location on every host in the Greenplum Database array (masters, segments, and mirrors).

Built-in Functions and Operators

The following table lists the categories of built-in functions and operators supported by PostgreSQL. All functions and operators are supported in Greenplum Database as in PostgreSQL with the exception of `STABLE` and `VOLATILE` functions, which are

subject to the restrictions noted in “Using Functions in Greenplum Database” on page 140. See the [Functions and Operators](#) section of the PostgreSQL documentation for more information about these built-in functions and operators.

Table 16.1 Built-in functions and operators

Operator/Function Category	VOLATILE Functions	STABLE Functions
Logical Operators		
Comparison Operators		
Mathematical Functions and Operators	random setseed	
String Functions and Operators	<i>All built-in conversion functions</i>	convert pg_client_encoding
Binary String Functions and Operators		
Bit String Functions and Operators		
Pattern Matching		
Data Type Formatting Functions		to_char to_timestamp
Date/Time Functions and Operators	timeofday	age current_date current_time current_timestamp localtime localtimestamp now
Geometric Functions and Operators		
Network Address Functions and Operators		
Sequence Manipulation Functions	currval lastval nextval setval	
Conditional Expressions		
Array Functions and Operators		<i>All array functions</i>
Aggregate Functions		
Subquery Expressions		
Row and Array Comparisons		
Set Returning Functions	generate_series	

Table 16.1 Built-in functions and operators

Operator/Function Category	VOLATILE Functions	STABLE Functions
System Information Functions		<i>All session information functions</i> <i>All access privilege inquiry functions</i> <i>All schema visibility inquiry functions</i> <i>All system catalog information functions</i> <i>All comment information functions</i>
System Administration Functions	set_config pg_cancel_backend pg_reload_conf pg_rotate_logfile pg_start_backup pg_stop_backup pg_size_pretty pg_ls_dir pg_read_file pg_stat_file	current_setting <i>All database object size functions</i>

Window Functions

Greenplum Database has the following built-in window functions that are not available in PostgreSQL. Window functions are often used to compose complex OLAP (online analytical processing) queries. Window functions are applied to partitioned result sets within the scope of a single query expression. A window partition is a subset of rows returned by a query, as defined in a special `OVER ()` clause. See “[Window Expressions](#)” on page 135. All window functions are *immutable* functions.

Note that any aggregate function (as described in [Aggregate Functions](#) of the PostgreSQL documentation) may also be used with an `OVER` clause, thereby making it a window aggregate function. A window aggregate function returns the aggregated value of the expression argument(s) for the specified window frame corresponding to a particular row. If the `OVER` clause does not specify ordering (`ORDER BY`) or window framing (`ROWS | RANGE`), the value is aggregated over the defined window partition. In the case where ordering and framing are not specified, if the aggregate function allows

the `DISTINCT` qualifier, then so does its corresponding window aggregate function. `DISTINCT` is not allowed for window aggregate functions that specify ordering or framing.

Table 16.2 Window functions

Function	Return Type	Full Syntax	Description
<code>cume_dist()</code>	double precision	<code>CUME_DIST() OVER ([PARTITION BY <i>expr</i>] ORDER BY <i>expr</i>)</code>	Calculates the cumulative distribution of a value in a group of values. Rows with equal values always evaluate to the same cumulative distribution value.
<code>dense_rank()</code>	bigint	<code>DENSE_RANK () OVER ([PARTITION BY <i>expr</i>] ORDER BY <i>expr</i>)</code>	Computes the rank of a row in an ordered group of rows without skipping rank values. Rows with equal values are given the same rank value.
<code>first_value(<i>expr</i>)</code>	same as input <i>expr</i> type	<code>FIRST_VALUE(<i>expr</i>) OVER ([PARTITION BY <i>expr</i>] ORDER BY <i>expr</i> [ROWS RANGE <i>frame_expr</i>])</code>	Returns the first value in an ordered set of values.
<code>lag(<i>expr</i> [, <i>offset</i>] [, <i>default</i>])</code>	same as input <i>expr</i> type	<code>LAG(<i>expr</i> [, <i>offset</i>] [, <i>default</i>]) OVER ([PARTITION BY <i>expr</i>] ORDER BY <i>expr</i>)</code>	Provides access to more than one row of the same table without doing a self join. Given a series of rows returned from a query and a position of the cursor, <code>LAG</code> provides access to a row at a given physical offset prior to that position. If <i>offset</i> is not specified, the default offset is 1. <i>default</i> sets the value that is returned if the offset goes beyond the scope of the window. If <i>default</i> is not specified, the default value is null.
<code>last_value(<i>expr</i>)</code>	same as input <i>expr</i> type	<code>LAST_VALUE(<i>expr</i>) OVER ([PARTITION BY <i>expr</i>] ORDER BY <i>expr</i> [ROWS RANGE <i>frame_expr</i>])</code>	Returns the last value in an ordered set of values.
<code>lead(<i>expr</i> [, <i>offset</i>] [, <i>default</i>])</code>	same as input <i>expr</i> type	<code>LEAD(<i>expr</i> [, <i>offset</i>] [, <i>default</i>]) OVER ([PARTITION BY <i>expr</i>] ORDER BY <i>expr</i>)</code>	Provides access to more than one row of the same table without doing a self join. Given a series of rows returned from a query and a position of the cursor, <code>LAG</code> provides access to a row at a given physical offset after that position. If <i>offset</i> is not specified, the default offset is 1. <i>default</i> sets the value that is returned if the offset goes beyond the scope of the window. If <i>default</i> is not specified, the default value is null.

Table 16.2 Window functions

Function	Return Type	Full Syntax	Description
<code>ntile(expr)</code>	bigint	<code>NTILE(expr) OVER ([PARTITION BY expr] ORDER BY expr)</code>	Divides an ordered dataset into a number of buckets (as defined by <i>expr</i>) and assigns a bucket number to each row.
<code>percent_rank()</code>	double precision	<code>PERCENT_RANK () OVER ([PARTITION BY expr] ORDER BY expr)</code>	Calculates the rank of a hypothetical row <i>R</i> minus 1, divided by 1 less than the number of rows being evaluated (within a window partition).
<code>rank()</code>	bigint	<code>RANK () OVER ([PARTITION BY expr] ORDER BY expr)</code>	Calculates the rank of a row in an ordered group of values. Rows with equal values for the ranking criteria receive the same rank. The number of tied rows are added to the rank number to calculate the next rank value. Ranks may not be consecutive numbers in this case.
<code>row_number()</code>	bigint	<code>ROW_NUMBER () OVER ([PARTITION BY expr] ORDER BY expr)</code>	Assigns a unique number to each row to which it is applied (either each row in a window partition or each row of the query).

Advanced Analytic Functions

Greenplum Database has the following built-in advanced analytic functions that are not available in PostgreSQL. All of these analytic functions are *immutable* functions.

Table 16.3 Advanced Analytic Functions

Function	Return Type	Full Syntax	Description
<code>matrix_add(array[], array[])</code>	smallint[], int[], bigint[], float[]	<code>matrix_add(array[[1,1],[2,2]], array[[3,4],[5,6]])</code>	Adds two two-dimensional matrices. The matrices must be conformable.
<code>matrix_multiply(array[], array[])</code>	smallint[], int[], bigint[], float[]	<code>matrix_multiply(array[[2,0,0],[0,2,0],[0,0,2]], [[3,0,3],[0,3,0]])</code>	Multiplies two two-dimensional arrays. The matrices must be conformable.
<code>matrix_multiply(array[], expr)</code>	int[], float[]	<code>matrix_multiply(array[[1,1,1],[2,2,2],[3,3,3]], 2)</code>	Multiplies a two-dimensional array and a scalar numeric value.
<code>matrix_transpose(array[])</code>	Same as input array type.	<code>matrix_transpose(array[[1,1,1],[2,2,2]])</code>	Transposes a two-dimensional array.

Table 16.3 Advanced Analytic Functions

Function	Return Type	Full Syntax	Description
<code>pinv(array[])</code>	<code>smallint[]</code> <code>int[]</code> , <code>bigint[]</code> , <code>float[]</code>	<code>pinv(array[[2.5,0,0],[0,1,0],[0,0,.5]])</code>	Calculates the Moore-Penrose pseudoinverse of a matrix.
<code>unnest(int[])</code>	set of anyelement	<code>unnest(array['one', 'row', 'per', 'item'])</code>	Transforms a one dimensional array into rows. Returns a set of anyelement, a polymorphic pseudotype in PostgreSQL.

Table 16.4 Advanced Aggregate Functions

Function	Return Type	Full Syntax	Description
<code>sum(array[])</code>	<code>smallint[]</code> <code>int[]</code> , <code>bigint[]</code> , <code>float[]</code>	<code>sum([array[1,2],[3,4]])</code>	Performs matrix summation. Can take as input a two-dimensional array that is treated as a matrix.
<code>pivot_sum(label[], label, expr)</code>	<code>int[]</code> , <code>bigint[]</code> , <code>float[]</code>	<code>pivot_sum(array['A1','A2'], attr, value)</code>	A pivot aggregation using sum to resolve duplicate entries.
<code>mregr_coef(expr, array[])</code>	<code>float[]</code>	<code>mregr_coef(y, array[1, x1, x2])</code>	The functions <code>mregr_coef</code> and <code>mregr_r2</code> are both multiple linear regression aggregates. <code>mregr_coef</code> calculates the regression coefficients.
<code>mregr_r2(expr, array[])</code>	<code>float[]</code>	<code>mregr_r2(y, array[1, x1, x2])</code>	The functions <code>mregr_coef</code> and <code>mregr_r2</code> are both multiple linear regression aggregates. <code>mregr_r2</code> calculates the r-squared error value.
<code>nb_classify(text[], bigint, bigint[], bigint[])</code>	<code>text[]</code>	<code>nb_classify(classes, attr_count, class_count, class_total)</code>	Classify rows using a naive Bayes classifier. This function is used with a baseline of training data to predict the classification of new rows.

Advanced Analytic Function Examples

These examples illustrate selected advanced analytic functions in queries on simplified example data. Examples are given for the multiple linear regression aggregate functions, and for Naive Bayes Classification with `nb_classify`.

Linear Regression Functions Example

This example uses the linear regression functions `mregr_coef` and `mregr_r2` in a query on the example table `regr_example`. In this example query, both functions take the dependent variable as the first parameter and an array of independent variables as the second parameter.

```
SELECT mregr_coef(y, array[1, x1, x2]), mregr_r2(y, array[1,
x1, x2]) from regr_example;
```

Table `regr_example`:

id	y	x1	x2
1	5	2	1
3	10	4	2
2	6	3	1

Running the example query against this simple table yields one row of data displaying these values:

`mregr_coef`:

```
{-3.90798504668055e-14,1.000000000000003,2.99999999999994}
```

`mregr_r2`:

```
0.999999999999953
```

If an intercept term is desired, it must be manually entered.

Naive Bayes Classification Example

The function `nb_classify` is used within a larger classification process that involves the creation of tables and views for training data. This example begins with the normalized data in the example table `class_example` and proceeds through four discrete steps, as described below.

Table `class_example`:

id	class	a1	a2	a3
1	C1	1	2	3
3	C1	1	4	3
5	C2	0	2	2
2	C1	1	2	1
4	C2	1	2	2
6	C2	0	1	3

1. Unpivot the data

For use as training data, the data in `class_example` must be unpivoted. In cases where the data is already in denormalized form, this step is not required.

```
CREATE view class_example_unpivot AS
```

```
SELECT id, class, unnest(array['A1', 'A2', 'A3']) as attr,
unnest(array[a1,a2,a3]) as value FROM class_example;
```

2. Create a training table from the unpivoted data

```
CREATE table class_example_nb_training AS
SELECT attr, value, pivot_sum(array['C1', 'C2'], class, 1)
as class_count
FROM class_example_unpivot
GROUP BY attr, value
DISTRIBUTED by (attr);
```

3. Create a summary view over the training data

```
CREATE VIEW class_example_nb_classify AS
SELECT attr, value, class_count, array['C1', 'C2'] as classes,
sum(class_count) over (wa)::integer[] as class_total,
count(distinct value) over (wa) as attr_count
FROM class_example_nb_training
WINDOW wa as (partition by attr);
```

4. Classify rows with nb_classify

Once the view is prepared, the training data is ready for use as a baseline for determining the class of incoming rows. The following query predicts whether rows are of class C1 or C2 by using the `nb_classify` aggregate:

```
SELECT nb_classify(classes, attr_count, class_count,
class_total) as class
FROM class_example_nb_classify
where (attr = 'A1' and value = 0) or (attr = 'A2' and value =
2) or (attr = 'A3' and value = 1);
```

This query yields the expected single-row result of C1.

```
class
-----
C2
(1 row)
```

Actual data in production scenarios will be more extensive than this simple example data, and will therefore yield better results. Accuracy of classification with `nb_classify` improves significantly with larger sets of training data.

Query Profiling

Greenplum Database devises a query plan for each query it is given. Choosing the right query plan to match the query and data structure is absolutely critical for good performance. A query plan defines how the query will be executed in Greenplum Database's parallel execution environment. By examining the query plans of poorly performing queries, you can identify possible performance tuning opportunities.

The query planner uses the database statistics it has to choose a query plan with the lowest possible cost. Cost is measured in disk I/O and CPU effort (shown as units of disk page fetches). The goal is to minimize the total execution cost for the plan.

You can view the plan for a given query using the `EXPLAIN` command. This will show the query planner's estimated plan for the query. For example:

```
EXPLAIN SELECT * FROM names WHERE id=22;
```

`EXPLAIN ANALYZE` causes the statement to be actually executed, not only planned. This is useful for seeing whether the planner's estimates are close to reality. For example:

```
EXPLAIN ANALYZE SELECT * FROM names WHERE id=22;
```

Reading EXPLAIN Output

Query plans are a tree plan of nodes. Each node in the plan represents a single operation, such as table scan, join, aggregation or a sort.

Plans should be read from the bottom up as each node feeds rows into the node directly above it. The bottom nodes of a plan are usually table scan operations (sequential, index or bitmap index scans). If the query requires joins, aggregations, or sorts (or other operations on the raw rows) then there will be additional nodes above the scan nodes to perform these operations. The topmost plan nodes are usually the Greenplum Database motion nodes (redistribute, broadcast, or gather motions). These are the operations responsible for moving rows between the segment instances during query processing.

The output of `EXPLAIN` has one line for each node in the plan tree, showing the basic node type plus the following cost estimates that the planner made for the execution of that plan node:

- **cost** - measured in units of disk page fetches; that is, 1.0 equals one sequential disk page read. The first estimate is the start-up cost (cost of getting to the first row) and the second is the total cost (cost of getting all rows). Note that the total cost assumes that all rows will be retrieved, which may not always be the case (if using `LIMIT` for example).
- **rows** - the total number of rows output by this plan node. This is usually less than the actual number of rows processed or scanned by the plan node, reflecting the estimated selectivity of any `WHERE` clause conditions. Ideally the top-level nodes estimate will approximate the number of rows actually returned, updated, or deleted by the query.
- **width** - total bytes of all the rows output by this plan node.

It is important to note that the cost of an upper-level node includes the cost of all its child nodes. The topmost node of the plan has the estimated total execution cost for the plan. This is this number that the planner seeks to minimize. It is also important to realize that the cost only reflects things that the query planner cares about. In particular, the cost does not consider the time spent transmitting result rows to the client.

EXPLAIN Example

To illustrate how to read an `EXPLAIN` query plan, consider the following example for a very simple query:

```
EXPLAIN SELECT * FROM names WHERE name = 'Joelle';
```

QUERY PLAN

```

-----
Gather Motion 2:1 (slice1) (cost=0.00..20.88 rows=1 width=13)
  -> Seq Scan on 'names' (cost=0.00..20.88 rows=1 width=13)
      Filter: name::text ~~ 'Joelle'::text

```

If we read the plan from the bottom up, the query planner starts by doing a sequential scan of the *names* table. Notice that the `WHERE` clause is being applied as a *filter* condition. This means that the scan operation checks the condition for each row it scans, and outputs only the ones that pass the condition.

The results of the scan operation are passed up to a *gather motion* operation. In Greenplum Database, a gather motion is when segments send rows up to the master. In this case we have 2 segment instances sending to 1 master instance (2:1). This operation is working on *slice1* of the parallel query execution plan. In Greenplum Database a query plan is divided into *slices* so that portions of the query plan can be worked on in parallel by the segments.

The estimated startup cost for this plan is 00.00 (no cost) and a total cost of 20.88 disk page fetches. The planner is estimating that this query will return one row.

Reading EXPLAIN ANALYZE Output

`EXPLAIN ANALYZE` causes the statement to be actually executed, not only planned. The `EXPLAIN ANALYZE` plan shows the actual results along with the planner's estimates. This is useful for seeing whether the planner's estimates are close to reality. In addition to the information shown in the `EXPLAIN` plan, `EXPLAIN ANALYZE` will show the following additional information:

- The total elapsed time (in milliseconds) that it took to run the query.
- The number of *workers* (segments) involved in a plan node operation. Only segments that return rows are counted.
- The maximum number of rows returned by the segment that produced the most rows for an operation. If multiple segments produce an equal number of rows, the one with the longest *time to end* is the one chosen.
- The segment id number of the segment that produced the most rows for an operation.
- For relevant operations, the *work_mem* used by the operation. If `work_mem` was not sufficient to perform the operation in memory, the plan will show how much data was spilled to disk and how many passes over the data were required for the lowest performing segment. For example:

```

Work_mem used: 64K bytes avg, 64K bytes max (seg0).
Work_mem wanted: 90K bytes avg, 90K bytes max (seg0) to abate workfile
I/O affecting 2 workers.
[seg0] pass 0: 488 groups made from 488 rows; 263 rows written to
workfile
[seg0] pass 1: 263 groups made from 263 rows

```

- The time (in milliseconds) it took to retrieve the first row from the segment that produced the most rows, and the total time taken to retrieve all rows from that segment. The *<time> to first row* may be omitted if it is the same as the *<time> to end*.

EXPLAIN ANALYZE Example

To illustrate how to read an EXPLAIN ANALYZE query plan, we will use the same simple query we used in the “EXPLAIN Example” on page 149. Notice that there is some additional information in this plan that is not in a regular EXPLAIN plan. The parts of the plan in **bold** show the actual timing and rows returned for each plan node:

```

EXPLAIN ANALYZE SELECT * FROM names WHERE name = 'Joelle';
                                QUERY PLAN
-----
Gather Motion 2:1 (slicel) (cost=0.00..20.88 rows=1 width=13)
  recv: Total 1 rows with 0.305 ms to first row, 0.537 ms to end.
    -> Seq Scan on 'names' (cost=0.00..20.88 rows=1 width=13)
      Total 1 rows (seg0) with 0.255 ms to first row, 0.486 ms to end.
        Filter: name::text ~~ 'Joelle'::text
22.548 ms elapsed

```

If we read the plan from the bottom up, you will see some additional information for each plan node operation. The total elapsed time it took to run this query was *22.548* milliseconds.

The *sequential scan* operation had only one segment (*seg0*) that returned rows, and it returned just *1 row*. It took *0.255* milliseconds to find the first row and *0.486* to scan all rows. Notice that this is pretty close to the planner’s estimate — the query planner estimated that it would return one row for this query, which it did. The *gather motion* operation then received 1 row (segments sending up to the master). The total elapsed time for this operation was *0.537* milliseconds.

What to Look for in a Query Plan

If a query is performing poorly, looking at its query plan can help identify problem areas. Here are some things to look for:

- **Is there one operation in the plan that is taking exceptionally long?** When looking through the query plan, is there one operation that is consuming the majority of the query processing time? For example, here is a snippet of a node from an EXPLAIN ANALYZE plan. Most of the time of the query (about 20 minutes) was spent on this bitmap index scan operation. Why? Perhaps the index is out-of-date and needs to be reindexed. You could also temporarily experiment with the `enable_` parameters to see if you can force the planner to choose a different (and potentially better) plan.

```

Bitmap Index Scan on date_idx (cost=0.00..339096.56 rows=35712593 width=0)
  Avg 35765315.83 rows x 6 workers. Max 35771652 rows (seg1) with 1190570 ms to end.
  Index Cond: ((date_id >= 50720) AND (date_id <= 60720))

```

- **Are the planner’s estimates close to reality?** Run an EXPLAIN ANALYZE and see if the number of rows estimated by the planner is close to the number of rows actually returned by the query operation. If there is a huge discrepancy, you may need to collect more statistics on the relevant columns. See “[Maintaining Database Statistics](#)” on page 235.

- **Are selective predicates applied early in the plan?** The most selective filters should be applied early in the plan so that less rows move up the plan tree. If the query plan is not doing a good job at estimating the selectivity of a query predicate, you may need to collect more statistics on the relevant columns. See [“Maintaining Database Statistics” on page 235](#). You can also try reordering the `WHERE` clause of your SQL statement.
- **Is the planner choosing the best join order?** When you have a query that joins multiple tables, make sure that the planner is choosing the most selective join order. Joins that eliminate the largest number of rows should be done earlier in the plan so that less rows move up the plan tree. If the plan is not choosing the optimal join order, you can set `join_collapse_limit=1` and use explicit `JOIN` syntax in your SQL statement to force the planner to the specified join order. You can also collect more statistics on the relevant join columns. See [“Maintaining Database Statistics” on page 235](#).
- **Is the planner selectively scanning partitioned tables?** If you are using table partitioning, is the planner selectively scanning only the child tables required to satisfy the query predicates? Do scans of the parent tables return 0 rows (they should, since the parent tables should not contain any data). For an example of a query plan that shows a selective partition scan, see [“Verifying Your Partition Strategy” on page 113](#).
- **Is the planner choosing hash aggregate and hash join operations where applicable?** Hash operations are typically much faster than other types of joins or aggregations. Row comparison and sorting is done in memory rather than reading/writing from disk. In order for hash operations to be chosen, there has to be sufficient work memory available to hold the number of estimated rows. Try increasing work memory to see if you can get better performance for a given query. If possible run an `EXPLAIN ANALYZE` for the query, which will show you which plan operations spilled to disk, how much work memory they used, and how much was required to not spill to disk. For example:

```
Work_mem used: 23430K bytes avg, 23430K bytes max (seg0).
Work_mem wanted: 33649K bytes avg, 33649K bytes max (seg0) to lessen
workfile I/O affecting 2 workers.
```

Note that the bytes wanted message from `EXPLAIN ANALYZE` is only a hint, based on the amount of data written to work files and is not exact. The minimum `work_mem` needed could be more or less than the suggested value.

17. Parallel Data Loading

One challenge of large scale, multi-terabyte data warehouses is getting large amounts of data loaded within a given maintenance window. Greenplum supports fast, parallel data loading with its external and web tables feature.

- [About External Tables and Web Tables](#)
- [Creating and Using External Tables](#)
- [Creating and Using Web Tables](#)
- [Non-Parallel Data Loading](#)
- [Other Data Loading Performance Tips](#)

About External Tables and Web Tables

External tables and web tables provide an easy way to perform basic extraction, transformation, and loading (ETL) tasks that are common in data warehousing. External and web table data is read in parallel by the Greenplum Database segment instances, so they also provide a means for fast data loading.

Once an external or web table is defined, its data can be queried directly (and in parallel) using SQL commands. You can, for example, select, join, or sort external or web table data. You can also create views for external or web tables. However external and web tables are read-only. DML operations (`UPDATE`, `INSERT`, `DELETE`, or `TRUNCATE`) are not allowed, and you cannot create indexes on external or web tables.

An external or web table definition can be thought of as a view that allows running any SQL query against external data sources without requiring that data to first be loaded into the database. External and web tables provide an easy way to perform basic extraction, transformation, and loading (ETL) tasks that are common in data warehousing. External and web table files are read in parallel by the Greenplum Database segment instances, so they also provide a means for fast data loading.

The main difference between external tables and web tables are their data sources. External tables access static flat files, whereas web tables access dynamic data sources — either on a web server or by executing OS commands or scripts. When a query is planned using an external table, the external table is considered rescannable since the data is thought to be static for the course of the query. For web tables, the data is not rescannable because there is the possibility that the data could change during the course of the query's execution.

External Tables and Query Planner Statistics

Because the data sources for external and web tables are outside of the database, statistics needed by the query planner are not collected when you run the `ANALYZE` command. You can set some rough statistics for an external or web table by manually editing the system catalog table `pg_class` and specifying the number of rows and database pages (calculated as `data_size / 32K`). By default, `pg_class.reltuples`

(number of rows) is set to 1000000 and `pg_class.relpages` (number of pages) is set to 1000 for all external tables and web tables when they are first defined. To change these defaults, you can update these values for your external table in `pg_class` (as the database superuser). For example:

```
=# UPDATE pg_class SET reltuples=500000, relpages=150
   WHERE relname='my_ext_table';
```

Creating and Using External Tables

The `CREATE EXTERNAL TABLE` command creates an external table definition in Greenplum Database. External table data is considered *static* (meaning the data does not change midstream during the execution of a query). This allows the query planner to choose plans that allow for rescanning of the external table data.

You may specify multiple external data sources or URIs (uniform resource identifiers) with the `LOCATION` clause — up to the number of *primary* segment instances in your Greenplum Database array. Each URI points to an external data file or data source. These URIs do not need to exist prior to defining an external table (`CREATE EXTERNAL TABLE` does not validate the URIs specified). However you will get an error if they cannot be found when querying the external table.

There are two protocols that you can use to access the external table data sources. You may use one of the following protocols per `CREATE EXTERNAL TABLE` statement (cannot mix protocols):

- gpfdist** — If using the `gpfdist://` protocol, you must have the Greenplum file distribution program (`gpfdist`) running on the host where the external data files reside. This program points to a given directory on the file host and serves external data files to all Greenplum Database segments in parallel. All primary segments access the external file(s) in parallel regardless of how many URIs you specify when defining the external table. When specifying which files to get using `gpfdist`, you can use the wildcard character (`*`) or other C-style pattern matching to denote multiple files. The files specified are assumed to be relative to the directory from which `gpfdist` is serving files (the directory specified when you started the `gpfdist` program).

`gpfdist` is located in `$GPHOME/bin` on your Greenplum Database master host. See the `gpfdist` help for more information on using this file distribution program with external tables.
- file** — If using the `file://` protocol the external data file(s) must reside on a segment host in a location accessible by the Greenplum super user (`gpadmin`). The number of URIs specified corresponds to the number of segment instances that will work in parallel to access the external table. So for example, if you have a Greenplum Database system with 8 primary segments and you specify 2 external files, only 2 of the 8 segments will access the external table in parallel at query time. The number of external files per segment host cannot exceed the number of primary segment instances on that host. For example, if your array has 4 primary segment instances per segment host, you may place 4 external files on each segment host. Also, the host name used in the URI must match the segment host name as registered in the `gp_configuration` system catalog table.

Formatting of Input Data

The `FORMAT` clause is used to describe how the external table files are formatted. The files can be in either delimited text (`TEXT`) or comma separated values (`CSV`) format. If the data in the file does not use the default column delimiter, escape character, null string and so on, you must specify the additional formatting options so that the data in the external file is read correctly by Greenplum Database. The format of the external data files is similar to the formatting requirements of files used by the `COPY` command.

The input data must follow a specific field-delimited format, and conform to the following syntax rules:

- A row is determined by a line feed character (`0x0a`). Blank rows in the data file (a row with a single line feed character) are allowed and will be ignored.
- The `DELIMITER` character must only appear between any two data value fields. Do not place a delimiter at the beginning or end of a row. For example:

```
data value 1 | data value 2 | data value 3
```
- `NULL` values are represented by the `NULL` string (`\N`) in `TEXT` mode, or by the default convention of nothing between two delimiters without quotations in `CSV` mode.
- Special characters such as a line feed (`0x0a`), `CSV` quotation character, or the delimiter character should be escaped (using the escape character specified) when used as data values within a field. The default escape character is a backslash (`\`).
- Microsoft Windows formatted files should be converted to remove any Windows only characters prior to loading into Greenplum Database. Try running the `dos2unix` system command on any Windows generated data files.

About Escaping

The data file has two reserved characters that have special meaning to Greenplum Database:

- The designated delimiter character (such as a tab or `|`), which is used to separate fields in the data file.
- A line feed (`0x0a`), which is used to designate a new row in the data file.

If your data contains either of these characters, you must escape the character so Greenplum treats it as data and not as a field separator or new row. By default, the escape character is a `\` (backslash).

For example, suppose you have a table with three columns and you want to load the following three fields:

- percentage sign = `%`
- vertical bar = `|`
- exclamation point = `!`

Your designated delimiter character is `|` (pipe character), and your designated escape character is `\` (backslash). The formatted row in your data file would look like this:

```
percentage sign = % | vertical bar = \| | exclamation point = !
```

Notice how the pipe character that is part of the data has been escaped using the backslash character.

If you want to use a different escape character, use the `ESCAPE AS` clause. In cases where your selected escape character is present in your data, you can use it to escape itself. In the following example, the escape character is `"` (double quote) and the delimiter is `|` (pipe character):

- Free trip to A|B
- 5.89
- Special rate "1.79"

In the formatted data file, the double quote character can be used to escape the pipe in the field `Free trip to A|B` and also to escape itself in the field `Special rate "1.79"`.

```
Free trip to A"|B | 5.89 | Special rate ""1.79""
```

If there is no need to escape the data, you can disable escaping by using:

```
ESCAPE 'OFF'
```

Using the Greenplum File Server (gpfdist)

One advantage of using the `gpfdist` file server program (as opposed to the file protocol) is that it ensures that all of the segments in your Greenplum Database system are fully utilized when reading from external table data files.

The `gpfdist` program can serve data to the segment instances at an average rate of about 350 MB/s for delimited text formatted files and 200 MB/s for CSV formatted files. Therefore, you should consider the following options when running `gpfdist` in order to maximize the network bandwidth of your ETL systems:

- If your ETL machine is configured with multiple network interface cards (NICs) as described in “[Network Layout](#)” on page 32, run one instance of `gpfdist` on your ETL host and then define your external table definition so that the host name of each NIC is declared in the `LOCATION` clause (see “[Defining an External Table](#)” on page 159 for an example). This allows network traffic between your Greenplum segment hosts and your ETL host to use all NICs simultaneously.

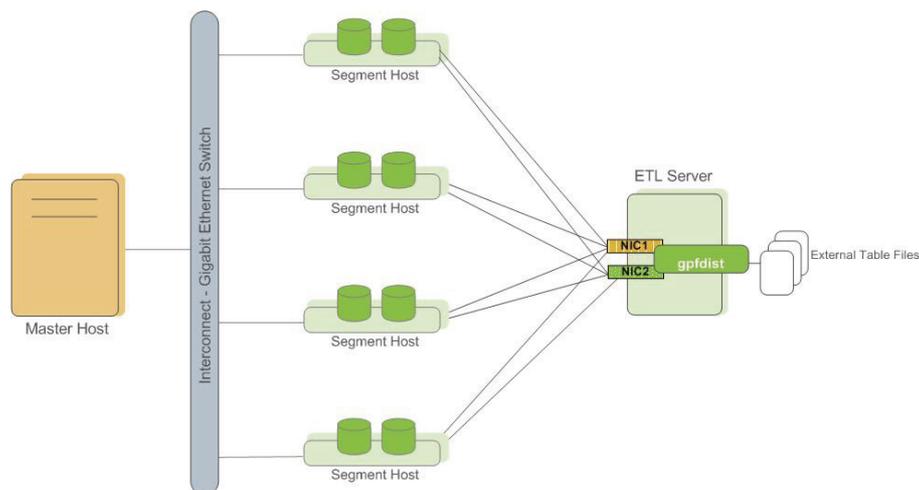


Figure 17.1 External Table Using Single `gpfdist` Instance with Multiple NICs

- Run multiple `gpfdist` instances on your ETL host and divide your external data files equally between each instance. For example, if you have an ETL system with two network interface cards (NICs), then you could run two `gpfdist` instances on that machine to maximize your load performance. You would then divide the external table data files evenly between the two `gpfdist` programs.

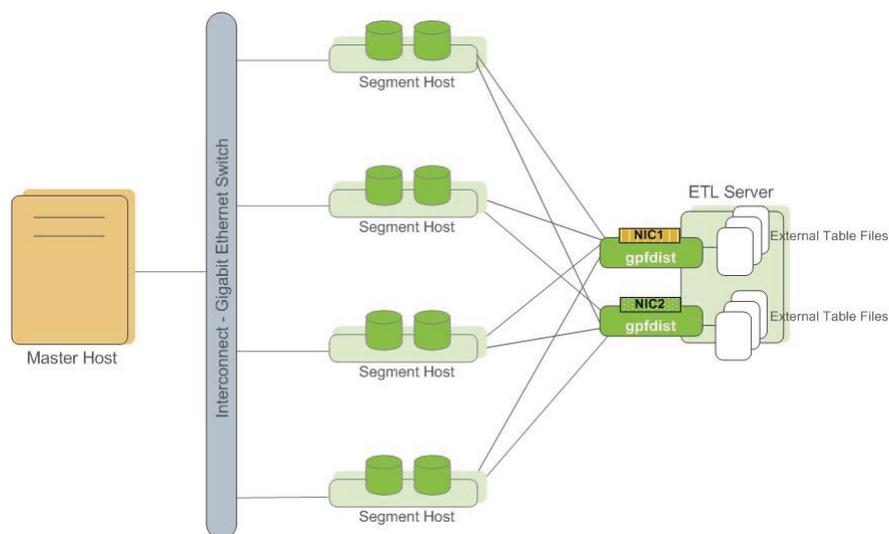


Figure 17.2 External Tables Using Multiple `gpfdist` Instances with Multiple NICs

Controlling Segment Parallelism

You can use the following server configuration parameter to control how many segment instances access a single `gpfdist` program at a time. 64 is the default. This allows you to control the number of segments processing external table files, while reserving some segments for other database processing. This parameter can be set in the `postgresql.conf` file of your master instance:

```
gp_external_max_segs
```

Installing gpfdist

The `gpfdist` program is installed in `$GPHOME/bin` of your Greenplum Database master host installation. Most likely you will want to run `gpfdist` from a machine other than your Greenplum Database master, such as on a machine devoted to ETL processing. To install `gpfdist` on another machine:

1. Copy `gpfdist` from `$GPHOME/bin` in your Greenplum installation to the remote machine.
2. Add `gpfdist` to your `$PATH`.

Starting and Stopping gpfdist

To start `gpfdist`, you must tell it from which directory it will be serving files and optionally the port to run on (defaults to HTTP port 8080).

To start `gpfdist` in the background (and log output messages and errors to a log file):

```
$ gpfdist -d /var/load_files -p 8081 -l /home/gpadmin/log &
```

If starting multiple `gpfdist` instances on the same ETL host (see [Figure 17.2](#)), then each should use a different port and base directory. For example:

```
$ gpfdist -d /var/load_files1 -p 8081 -l /home/gpadmin/log1 &
$ gpfdist -d /var/load_files2 -p 8082 -l /home/gpadmin/log2 &
```

To stop `gpfdist` when its running in the background:

--First find its process id:

```
$ ps ax | grep gpfdist (Linux)
$ ps -ef | grep gpfdist (Solaris)
```

--Then kill the process, for example:

```
$ kill 3456
```

Troubleshooting gpfdist

Keep in mind that `gpfdist` is accessed at runtime by the segment instances. Therefore, you must ensure that the Greenplum segment hosts have network access to `gpfdist`. The `gpfdist` program is a simple web server, so to test connectivity you can run the following command from each host in your Greenplum array (segments and master):

```
$ wget http://gpfdist_hostname:port/filename
```

Also, make sure that your `CREATE EXTERNAL TABLE` definition has the correct host name, port, and file names for `gpfdist`. The file names and paths specified should be relative to the directory from which `gpfdist` is serving files (the directory path used when you started the `gpfdist` program). See “[Defining an External Table](#)” on page 159.

Defining an External Table

Use the `CREATE EXTERNAL TABLE` command to define the external table and specify the location and format of the external table data files.

For example, if using a single Greenplum file server (`gpfdist`) instance on a machine with multiple NICs (see [Figure 17.1](#)):

```
=# CREATE EXTERNAL TABLE ext_expenses ( name text,
    date date, amount float4, category text, desc text )
    LOCATION ('gpfdist://etlhost-1:8081/*',
             'gpfdist://etlhost-2:8081/*')
    FORMAT 'TEXT' (DELIMITER ',');
```

For example, if using multiple Greenplum file server (`gpfdist`) instances (see [Figure 17.2](#)):

```
=# CREATE EXTERNAL TABLE ext_expenses ( name text,
    date date, amount float4, category text, desc text )
    LOCATION ('gpfdist://etlhost-1:8081/*',
             'gpfdist://etlhost-2:8082/*')
    FORMAT 'TEXT' (DELIMITER ',');
```

Defining External Tables in Single Row Error Isolation Mode

The most common use of external tables is selecting data from them to load into regular database tables. This is typically done by issuing a `CREATE TABLE AS SELECT` or `INSERT INTO SELECT` command, where the `SELECT` statement queries external table data. By default, if the external table data contains an error, the entire command fails and no data is loaded into the target database table. To isolate data errors in external table data while still loading correctly formatted rows, you can define an external table with a `SEGMENT REJECT LIMIT` clause. The user can specify the number of error rows acceptable (on a per-segment basis), after which the entire external table operation will be aborted and no rows will be processed or loaded. Note that the count of error rows is per-segment, not per entire operation. If the per-segment reject limit is not reached, then all rows not containing an error will be processed. If the limit is not reached, all good rows will be processed and any error rows discarded. If you would like to keep error rows for further examination, you can optionally declare an error table using the `LOG ERRORS INTO` clause. Any rows containing a format error would then be logged to the specified error table. For example:

```
=# CREATE EXTERNAL TABLE ext_expenses ( name text,
    date date, amount float4, category text, desc text )
    LOCATION ('gpfdist://etlhost-1:8081/*',
             'gpfdist://etlhost-2:8082/*')
```

```

FORMAT 'TEXT' (DELIMITER '|')
LOG ERRORS INTO err_expenses SEGMENT REJECT LIMIT 10 ROWS;

```

When `SEGMENT REJECT LIMIT` is used, then the external data will be scanned in single row error isolation mode. This can be helpful in isolating errors when loading data from an external table using `CREATE TABLE AS SELECT` or `INSERT INTO SELECT`. In this release, single row error isolation mode only applies to external data rows with format errors — for example, extra or missing attributes, attributes of a wrong data type, or invalid client encoding sequences. Constraint errors are not checked, however it is easy to filter out constraint errors when using external tables by limiting the `SELECT` from an external table at runtime. For example, to eliminate duplicate key errors:

```

=# INSERT INTO table_with_pkeys
  SELECT DISTINCT * FROM external_table;

```

Loading Data From an External Table

Once you have defined the external table, and have placed the necessary data files in the correct location (and have started the Greenplum files server(s) if using the `gpfdist` protocol), you can then select from the external table as you would an ordinary database table. For example, if you wanted to load a portion of the external table data into a database table, you could do something like:

```

=# INSERT INTO expenses_travel
  SELECT * from ext_expenses where category='travel';

```

Or if you wanted to quickly load all data into a new database table:

```

=# CREATE TABLE expenses AS SELECT * from ext_expenses;

```

Investigating Load Errors

If you are using the single row error isolation feature (see [“Defining External Tables in Single Row Error Isolation Mode”](#) on page 159 or [“Running COPY in Single Row Error Isolation Mode”](#) on page 164), any rows with formatting errors are logged into an error table. The error table has the following columns:

Table 17.1 error table format

column	type	description
cmdtime	timestampz	Timestamp when the error occurred.
relname	text	The name of the external table that was scanned or the target table of a COPY command.
filename	text	The name of the load file that contained the error.

Table 17.1 error table format

column	type	description
linenum	int	If COPY was used, the line number in the load file where the error occurred. For external tables using file:// protocol or gpfdist:// protocol and CSV format, the file name and line number is logged.
bytenum	int	If using external tables with the gpfdist:// protocol and data is in TEXT format, the byte offset in the load file where the error occurred is logged. gpfdist parses TEXT files in blocks, so logging a line number is not possible. CSV files are parsed a line at a time so line number tracking is possible for CSV files but not for TEXT files.
errmsg	text	The error message text.
rawdata	text	The raw data of the rejected row.
rawbytes	bytea	In cases where there is a database encoding error (the client encoding used cannot be converted to a server-side encoding), it is not possible to log the encoding error as <i>rawdata</i> . Instead the raw bytes are stored and you will see the octal code for any non seven bit ASCII characters.

Identifying Invalid CSV Files

If a comma separated values (CSV) file has invalid formatting, the *rawdata* field in the error table may contain several combined rows. For example, if a closing quote for a specific field is missing, all the following newlines are treated as embedded newlines. When this happens, Greenplum stops parsing a row when it reaches 64K, puts that 64K of data into the error table as a single row, resets the quote flag, and continues. If this happens three times during load processing, the load file is determined to be invalid and the entire load fails with the message “rejected *N* or more rows”.

Creating and Using Web Tables

`CREATE EXTERNAL WEB TABLE` creates a web table definition in Greenplum Database. A web table is a type of external table that allows you to access dynamic data sources as though they were a regular database table. Web table data is considered *dynamic* (meaning the data could possibly change midstream during the execution of a query). Therefore, the query planner must choose plans that do not allow for rescanning of the web table data.

- **Web URLs.** Specify the `LOCATION` of files on a web server using the `http://` protocol. The web data file(s) must reside on a web server that is accessible by the Greenplum segment hosts. The number of URLs specified corresponds to the number of segment instances that will work in parallel to access the web table. So for example, if you have a Greenplum Database system with 8 primary segments and you specify 2 external files, only 2 of the 8 segments will access the web table in parallel at query runtime.
- **OS Command.** Specify a shell command or script to `EXECUTE` on one or more segments, the output of which will comprise the web table's data at the time of access. A web table defined with an `EXECUTE` clause will execute the given OS shell command or script on the specified segment host or hosts. By default, the command is executed by all active segment instances on all segment hosts. For example, if each segment host has four primary segment instances running, the command will be executed four times per segment host. You can optionally limit the number of segment instances that execute the web table command.
Web table data is comprised of the output of the command at the time the web table statement is executed. All segment instances included in the web table definition (as specified by the `ON` clause) execute the command in parallel.

Defining Command-Based Web Tables

If you use environment variables in external web table commands (such as `$PATH`), keep in mind that the command is executed from within the database and not from a login shell. Therefore the `.bashrc` or `.profile` of the current user will not be sourced. However, you can set desired environment variables from within the `EXECUTE` clause of your external web table definition, for example:

```

=# CREATE EXTERNAL WEB TABLE blah (blah text)
    EXECUTE 'export BLAH=blah-blah-blah; echo $BLAH'
    FORMAT 'TEXT';

=# SELECT * FROM blah;
   blah
-----
blah-blah-blah
blah-blah-blah
(2 rows)

```

Also, any scripts that you want to execute must be present in the same location on the segment hosts and be executable by the `gpadmin` user.

For example, here is a command that defines a web table that executes a script once per segment host:

```

=# CREATE EXTERNAL WEB TABLE log_output
    (linenum int, message text)
    EXECUTE '/var/load_scripts/get_log_data.sh' ON HOST
    FORMAT 'TEXT' (DELIMITER '|');

```

Defining URL-Based Web Tables

A URL-based web table is similar to a file-based external table except that the data is accessed from a web server via the HTTP protocol. Also, web table data is assumed to be dynamic, so unlike file-based external tables, the data is not rescannable.

For example, here is a command that defines a web table that gets data from several different URLs:

```
=# CREATE EXTERNAL WEB TABLE ext_expenses (name text,
      date date, amount float4, category text, description text)
      LOCATION (
        'http://intranet.company.com/expenses/sales/file.csv',
        'http://intranet.company.com/expenses/exec/file.csv',
        'http://intranet.company.com/expenses/finance/file.csv',
        'http://intranet.company.com/expenses/ops/file.csv',
        'http://intranet.company.com/expenses/marketing/file.csv',
        'http://intranet.company.com/expenses/eng/file.csv',
      )
      FORMAT 'CSV' ( HEADER );
```

Loading Data from Web Tables

Once your web table is defined, you can use SQL commands to access it just as you would a regular external table. See [“Loading Data From an External Table”](#) on page 160. One important thing to note about web tables is that since they are not rescannable during query execution time, it is more important to manually set statistics for web tables in order to get better query plans. See [“External Tables and Query Planner Statistics”](#) on page 153.

Non-Parallel Data Loading

Greenplum Database also offers the standard PostgreSQL `COPY` and `INSERT` commands to load data. These are non-parallel load mechanisms, meaning that data is loaded in a single process via the Greenplum master instance.

Loading Data with COPY

`COPY` copies data from a file (or standard input) into a table (appending the data to whatever is in the table already). If copying data from a file, the file must be accessible to the master host and the name must be specified from the viewpoint of the master host. When `STDIN` or `STDOUT` is specified, data is transmitted via the connection between the client and the master server.

To maximize the performance and throughput of `COPY`, consider running multiple `COPY` commands concurrently in separate sessions and dividing the data to be loaded evenly across all concurrent processes. To maximize throughput, run one concurrent `COPY` operation per CPU.

Running COPY in Single Row Error Isolation Mode

By default, COPY stops an operation at the first error, meaning if the data being loaded contains an error, the entire operation fails and no data is loaded. Optionally, a COPY FROM command can be run in *single row error isolation mode*. In this mode, rows containing format errors — for example, extra or missing attributes, attributes of a wrong data type, or invalid client encoding sequences — can be skipped over while still loading all properly formatted rows. In this release, single row error isolation mode only applies to rows in the input file with format errors. Constraint errors such as violation of a NOT NULL, CHECK, or UNIQUE constraint will still be handled in ‘all-or-nothing’ input mode.

The SEGMENT REJECT LIMIT clause when added to a COPY FROM operation will run the command in single row error isolation mode. The user can specify the number of error rows acceptable (on a per-segment basis), after which the entire COPY FROM operation will be aborted and no rows will be loaded. Note that the count of error rows is per Greenplum segment, not per entire load operation. If the per-segment reject limit is not reached, then all rows not containing an error will be loaded. If the limit is not reached, all good rows will be loaded and any error rows discarded. If you would like to keep error rows for further examination, you can optionally declare an error table using the LOG ERRORS INTO clause. Any rows containing a format error would then be logged to the specified error table. For example:

```
=> COPY country FROM '/data/gpdb/country_data'
    WITH DELIMITER '|' LOG ERRORS INTO err_country
    SEGMENT REJECT LIMIT 10 ROWS;
```

Loading Data with INSERT

If you just have a small number of rows to load, you can use the INSERT command. See “[Inserting New Rows](#)” on page 126.



Note: The storage model of append-only tables is optimized for bulk data loading. Single row INSERT statements are not recommended.

Other Data Loading Performance Tips

Drop Indexes Before Loading — If you are loading a freshly created table, the fastest way is to create the table, load the data, then create any indexes needed for the table. Creating an index on pre-existing data is quicker than updating it incrementally as each row is loaded. If you are adding large amounts of data to an existing table, it may be a faster to drop the index, load the table, and then recreate the index. Temporarily increasing the *maintenance_work_mem* server configuration parameter will help to speed up CREATE INDEX commands, although it will not help performance of the load itself. Dropping and recreating indexes should be done when there are no users on the system.

Run `ANALYZE` After Loads — Whenever you have significantly altered the data in a table, running `ANALYZE` is strongly recommended. Running `ANALYZE` (or `VACUUM ANALYZE`) ensures that the query planner has up-to-date statistics about the table. With no statistics or obsolete statistics, the planner may make poor decisions during query planning, leading to poor performance on any tables with inaccurate or nonexistent statistics.

Run `VACUUM` After Load Errors — If not running in single row error isolation mode, a load operation stops at the first encountered error. The target table will already have received earlier rows before the error occurred. These rows will not be visible or accessible, but they still occupy disk space. This may amount to a considerable amount of wasted disk space if the failure happened well into a large load operation. Invoking the `VACUUM` command will recover the wasted space.

Section V: System Administration

This section describes how to monitor and administer a Greenplum Database system.

- [Starting and Stopping Greenplum](#)
- [Configuring Your Greenplum System](#)
- [Enabling High Availability Features](#)
- [Backing Up and Restoring Databases](#)
- [Expanding a Greenplum System](#)
- [Monitoring a Greenplum System](#)
- [Routine System Maintenance Tasks](#)

18. Starting and Stopping Greenplum

This chapter describes how to start, stop, or restart a Greenplum Database system. This chapter contains the following topics:

- [Overview](#)
- [Starting Greenplum Database](#)
- [Stopping Greenplum Database](#)

Overview

Because a Greenplum Database system is distributed across many machines, the process for starting and stopping a Greenplum Database DBMS is different than a regular PostgreSQL DBMS. With a regular PostgreSQL DBMS, you run a utility called `pg_ctl`, which starts, stops, or restarts the database server process (`postgres`). `pg_ctl` also takes care of tasks such as redirecting log output and properly detaching from the terminal and process group.

In a Greenplum Database DBMS, each database server instance (the master and all segments) must be started or stopped across all of the hosts in the system in such a way that they can all work together as a unified DBMS.

Greenplum provides similar functionality of the `pg_ctl` utility with the `gpstart` and `gpstop` utilities, which are located in `$GPHOME/bin` of your Greenplum Database master host installation.

Starting Greenplum Database

The database initialization process (`gpinitssystem`) will start your Greenplum Database system for you once it has completed successfully. However, there may be times when you will need to restart the system, for example to reset operational configuration parameters or to troubleshoot a failed segment.

Use the `gpstart` utility to start your Greenplum Database system. This utility starts all of the `postgres` database listener processes in the system (the master and all of the segment instances). `gpstart` is always run on the master host.

Each instance is started in parallel (up to 60 parallel processes by default).

To start Greenplum Database

```
$ gpstart
```

Restarting Greenplum Database

The `gpstop` utility has a restart option that will restart the Greenplum Database system once a successful shutdown has completed.

To restart Greenplum Database

```
$ gpstop -r
```

Uploading Configuration File Changes Only

The `gpstop` utility has an option that will upload changes made to the `pg_hba.conf` configuration file and to *runtime* parameters in the master `postgresql.conf` file without interruption of service. Note that any active sessions will not pickup the changes until they reconnect to the database, and that many server configuration parameters require a full system restart (`gpstop -r`) to be activated. For more information, see “[Configuration Parameter Categories](#)” on page 172.

To upload configuration file changes without restarting

```
$ gpstop -u
```

Starting the Master in Maintenance Mode

There may be cases where you want to start only the master. This is called *maintenance mode*. In this mode, you can do things such as connect to a database on the master instance only in utility mode and edit settings in the system catalog, without affecting user data on the segment instances. See “[System Catalog Reference](#)” on page 817 for more information about the system catalog tables.

To start the master in utility mode

1. Run `gpstart` using the `-m` option:

```
$ gpstart -m
```

2. Connect to the master in utility mode to do catalog maintenance. For example:

```
$ PGOPTIONS='-c gp_session_role=utility' psql template1
```

3. After completing your administrative tasks, you must stop the master in utility mode before you can restart it again in production mode.

```
$ gpstop -m
```

Stopping Greenplum Database

Use the `gpstop` utility to stop or restart your Greenplum Database system. This utility stops all of the `postgres` processes in the system (the master and all of the segment instances). `gpstop` is always run on the master host.

Each instance is stopped in parallel (up to 60 parallel processes by default). By default the system will wait for any active transactions to finish before shutting down, and will not shutdown if the database has any active client connections.

To stop Greenplum Database

```
$ gpstop
```

To stop Greenplum Database in fast mode

(all active transactions are interrupted and rolled back, all active client sessions are cancelled)

```
$ gpstop -M fast
```

19. Configuring Your Greenplum System

There are many server configuration parameters that affect the behavior of a Greenplum Database system. Most of these parameters are the same as in regular PostgreSQL, but there are also a few Greenplum-specific configuration parameters.

- [About Greenplum Master, Global, and Local Parameters](#)
- [Setting Configuration Parameters](#)
- [Configuration Parameter Categories](#)

About Greenplum Master, Global, and Local Parameters

In most database management systems, you have a server configuration file that configures various aspects of the server. In Greenplum Database (as in PostgreSQL) this file is called `postgresql.conf`. This configuration file is located in the data directory of the database instance.

In Greenplum Database, the master and each segment instance has its own `postgresql.conf` file (located in their respective data directories). Some parameters are considered *local* parameters, meaning that each segment instance looks to its own `postgresql.conf` file to get the value of that parameter. You must set local parameters on every instance in the system (master and segments).

Others parameters are considered *global* or *master* parameters. Global and master parameters need only be set at the master instance. A parameter classified as *global* is passed down to the segment instances at query run time. A parameter classified as *master* is not passed to the segment instances, but are only relevant to master processes such as query planning and client authentication.

See “[Server Configuration Parameters](#)” on page 755 to see if a particular parameter is classified as *local*, *master* or *global*.

Setting Configuration Parameters

Many of the configuration parameters have limitations on who can change them and where or when they can be set. For example, to change certain parameters, you must be a Greenplum Database superuser. Other parameters can only be set in the `postgresql.conf` file or require a restart of the system for the changes to take effect.

Many configuration parameters are considered *runtime* parameters. A runtime parameter can be set at the system-level, the database-level, the role-level or the session-level. Most runtime parameters can be changed by any database user within their session, but a few may require superuser permissions. See “[Server Configuration Parameters](#)” on page 755 to see the set classifications of particular parameters.

Setting a Local Configuration Parameter

If you want to change a configuration parameter that is classified as *local*, you must change it in every `postgresql.conf` file where you want that change to take effect. In most cases, this means making the change at the master and at every segment (primary and mirror). The last value found in the `postgresql.conf` file is the one that is read by the server. This means that you can append changed parameters to the end of the file and previous entries in the file will be ignored. For example, to edit several segment's `postgresql.conf` files at once, you could use a `gpssh` command such as this:

```
$ gpssh -f seg_hosts_file "echo 'parameter=value' | cat - >>
/gpdata/p*/*/postgresql.conf"
```

Then restart Greenplum Database to make the configuration changes effective:

```
$ gpstop -r
```

Setting a Global or Master Configuration Parameter

If a parameter is classified as *global* or *master*, you only need to set it at the Greenplum master instance. If it is also a *runtime* parameter, you have the additional flexibility of setting the parameter for a particular database, role or session. If a parameter is set at multiple levels, then the more granular level takes precedence. For example, session overrides role, role overrides database, and database overrides system.

Setting Parameters at the System Level

Setting global or master parameters in the master `postgresql.conf` file makes that setting the new system-wide default.

1. Edit the `$MASTER_DATA_DIRECTORY/postgresql.conf` file.
2. Find the parameter you want to change, uncomment it (remove the preceding # character), and set it to the desired value.
3. Save and close the file.
4. For *runtime* parameters that do not require a server restart, you can upload the `postgresql.conf` changes as follows:


```
$ gpstop -u
```
5. For parameter changes that require a server restart, restart Greenplum Database as follows:


```
$ gpstop -r
```

Note: See “[Server Configuration Parameters](#)” on page 755 for detailed descriptions of the server configuration parameters.

Setting Parameters at the Database Level

When a runtime parameter is set at the database level, every session that connects to that database will pick up that parameter setting. Settings at the database level override those at the system level. Use the `ALTER DATABASE` command to set a parameter at the database level. For example:

```
=# ALTER DATABASE mydatabase SET search_path TO myschema;
```

Setting Parameters at the Role Level

When a runtime parameter is set at the role level, every session initiated by that role will pick up that parameter setting. Settings at the role level override those at the database level. Use the `ALTER ROLE` command to set a parameter at the role level. For example:

```
=# ALTER ROLE bob SET search_path TO bobschema;
```

Setting Parameters in a Session

Any runtime parameter can also be set in an active database session using the `SET` command. That parameter setting is then valid for the rest of that session (or until a `RESET` command is issued). Settings at the session level override those at the role level. For example:

```
=# SET work_mem TO 200MB;
=# RESET work_mem;
```

Configuration Parameter Categories

There are many configuration parameters that affect the behavior of a Greenplum Database system. This section explains the general categories of configuration parameters. See “[Server Configuration Parameters](#)” on page 755 to see the details of particular parameters.

- [File Location Parameters](#)
- [Connection and Authentication Parameters](#)
- [System Resource Consumption Parameters](#)
- [Write Ahead Log Parameters](#)
- [Query Tuning Parameters](#)
- [Error Reporting and Logging Parameters](#)
- [Runtime Statistics Collection Parameters](#)
- [Automatic Statistics Collection Parameters](#)
- [Automatic Vacuuming Parameters](#)
- [Client Connection Default Parameters](#)
- [Lock Management Parameters](#)
- [Workload Management Parameters](#)
- [External Table Parameters](#)

- [Past PostgreSQL Version Compatibility Parameters](#)
- [Greenplum Array Configuration Parameters](#)

File Location Parameters

These parameters control the locations of the server configuration files for a Greenplum instance. All file location parameters are *local* parameters. The default location of these files is the segment or master instance's data directory. The location of the master and segment data directories are initially set by `gpinitssystem`.

- [data_directory](#)
- [config_file](#)
- [hba_file](#)
- [ident_file](#)
- [external_pid_file](#)

Connection and Authentication Parameters

These parameters control how clients connect and authenticate to Greenplum Database. See also, “[Configuring Client Authentication](#)” on page 73.

Connection Parameters

- [listen_addresses](#)
- [port](#) (see “[Changing the Master Port](#)”)
- [max_connections](#)
- [max_prepared_transactions](#)
- [superuser_reserved_connections](#)
- [unix_socket_directory](#)
- [unix_socket_group](#)
- [unix_socket_permissions](#)
- [bonjour_name](#)
- [tcp_keepalives_count](#)
- [tcp_keepalives_idle](#)
- [tcp_keepalives_interval](#)

Changing the Master Port

Greenplum Database comes configured with the default master port of 5432. Administrators should follow this process to change the master port number. Perform these steps on your Greenplum master host.

1. Shut down your Greenplum system in fast mode, interrupting and rolling back any active transactions. There cannot be active queries in the system when the system configuration is being altered.

```
$ gpstop -M fast
```

2. Edit the `postgresql.conf` file of the master. For example:

```
$ vi $MASTER_DATA_DIRECTORY/postgresql.conf
```

3. Change the following parameter to the new port number. For example if the new port is 54321:

```
port=54321
```

NOTE: If you have a standby master host, you must also edit the `$MASTER_DATA_DIRECTORY/postgresql.conf` file on your standby master host. After you are finished, return to your primary master host to complete the process.

- Restart your Greenplum Database system.

```
$ gpstart
```

- Start a `psql` client session using the new port and edit the `gp_configuration` system catalog to change the master port. For example, if the new port is 54321 and the old port is 5432:

```
psql -p 54321 -c "UPDATE gp_configuration SET port=54321
WHERE port=5432"
```

- Make sure that any client programs you are using are updated to connect to the new master port. For example, if using `psql` update the `$PGPORT` environment variable in the profile of your Greenplum superuser (`gpadmin`).

Security and Authentication Parameters

- `authentication_timeout`
- `ssl`
- `password_encryption`
- `krb_caseins_users`
- `krb_server_hostname`
- `krb_server_keyfile`
- `krb_srvname`
- `db_user_namespace`

System Resource Consumption Parameters

Memory Consumption Parameters

These parameters control system memory usage. The most commonly used tuning parameter is `work_mem`, but you may also want to adjust `gp_vmem_protect_limit` to avoid running out of memory at the segment hosts during query processing.

- `shared_buffers`
- `temp_buffers`
- `max_prepared_transactions`
- `work_mem`
- `maintenance_work_mem`
- `max_stack_depth`
- `gp_vmem_protect_gang_cache_limit`
- `gp_vmem_protect_limit`
- `max_appendonly_tables`

Free Space Map Parameters

These parameters control the sizing of the *free space map* where expired rows are held. Disk space in the free space map is reclaimed by a `VACUUM` command. See also, “Vacuuming the Database” on page 130.

- `max_fsm_pages`
- `max_fsm_relations`

OS Resource Parameters

- [max_files_per_process](#)
- [shared_preload_libraries](#)

Cost-Based Vacuum Delay Parameters

Standard PostgreSQL has a number of parameters for configuring the execution cost of `VACUUM` and `ANALYZE` commands. The intent of this feature is to allow administrators to reduce the I/O impact of these commands on concurrent database activity. When the accumulated cost of the various I/O operations reaches the limit, the process performing the operation will sleep for a while. Then it will reset the counter and continue execution.



Warning: Cost-based vacuum delay is not recommended in Greenplum Database because it runs asynchronously between the segment instances. The vacuum cost limit and delay is invoked locally at the segment level without taking into account the state of the entire Greenplum array.

- [vacuum_cost_delay](#)
- [vacuum_cost_limit](#)
- [vacuum_cost_page_dirty](#)
- [vacuum_cost_page_hit](#)
- [vacuum_cost_page_miss](#)

Background Writer Parameters

The background writer server process issues writes of dirty shared buffers. The intent is that server processes handling user queries should seldom or never have to wait for a write to occur, because the background writer will do it. The background writer will continuously trickle out dirty pages to disk, so that only a few pages will need to be forced out when checkpoint time arrives. In most situations a continuous low I/O load is preferable to periodic spikes, but these parameters can be used to tune the behavior.

- [bgwriter_delay](#)
- [bgwriter_lru_percent](#)
- [bgwriter_lru_maxpages](#)
- [bgwriter_all_maxpages](#)
- [bgwriter_all_percent](#)

Write Ahead Log Parameters

Write-Ahead Logging (WAL) is a standard approach to transaction logging. WAL's central concept is that changes to data files (where tables and indexes reside) must be written only after those changes have been logged, that is, when log records describing the changes have been flushed to permanent storage. WAL significantly reduces the number of disk writes, because only the log file needs to be flushed to disk at the time

of transaction commit, rather than every data file changed by the transaction. Most likely, you will not need to change these defaults in Greenplum Database, as they have already been tuned for Greenplum's distributed transaction management.

- fsync
- wal_sync_method
- full_page_writes
- wal_buffers
- commit_delay
- commit_siblings
- checkpoint_segments
- checkpoint_timeout
- checkpoint_warning

Query Tuning Parameters

Query Plan Operator Control Parameters

The following parameters control the types of plan operations the query planner has to choose from. Enabling or disabling certain plan operations is a way to force the planner to choose a different plan. These can be useful in testing a query using different plan types to see which plan offers the best performance.

- enable_bitmapscan
- enable_groupagg
- enable_hashagg
- enable_hashjoin
- enable_indexscan
- enable_mergejoin
- enable_nestloop
- enable_seqscan
- enable_sort
- enable_tidscan
- gp_enable_adaptive_nestloop
- gp_enable_agg_distinct
- gp_enable_agg_distinct_pruning
- gp_enable_fallback_plan
- gp_enable_fast_sri
- gp_enable_groupext_distinct_gather
- gp_enable_groupext_distinct_pruning
- gp_enable_multiphase_agg
- gp_enable_predicate_propagation
- gp_enable_preunique
- gp_enable_sequential_window_plans
- gp_enable_sort_distinct
- gp_enable_sort_limit

Query Planner Costing Parameters



Important: Greenplum recommends that you do not adjust these query costing parameters. They have already been tuned to reflect Greenplum Database hardware configurations and typical workloads. All of these parameters are related. Changing one without changing the others can have adverse affects on performance.

- `effective_cache_size`
- `seq_page_cost`
- `random_page_cost`
- `cpu_index_tuple_cost`
- `cpu_operator_cost`
- `cpu_tuple_cost`
- `gp_motion_cost_per_row`
- `gp_segments_for_planner`

Database Statistics Sampling Parameters

These parameters adjust the amount of data sampled by an `ANALYZE` operation. Adjusting these parameters will affect statistics collection system-wide. You can also configure statistics collection on particular tables and columns by using the `ALTER TABLE SET STATISTICS` clause.

- `default_statistics_target`
- `gp_analyze_relative_error`

Sort Operator Configuration Parameters

- `gp_enable_sort_distinct`
- `gp_enable_sort_limit`

Aggregate Operator Configuration Parameters

- `gp_enable_preunique`
- `gp_enable_agg_distinct`
- `gp_enable_agg_distinct_pruning`
- `gp_enable_multiphase_agg`
- `gp_hashagg_compress_spill_files`
- `gp_enable_grouptext_distinct_gather`
- `gp_enable_grouptext_distinct_pruning`

Join Operator Configuration Parameters

- `join_collapse_limit`
- `gp_statistics_use_fkeys`
- `gp_adjust_selectivity_for_outerjoins`

Other Query Planner Configuration Parameters

- `from_collapse_limit`
- `constraint_exclusion`
- `gp_enable_predicate_propagation`
- `gp_statistics_pullup_from_child_partition`
- `cursor_tuple_fraction`

Error Reporting and Logging Parameters

Where to Log

- `log_filename`
- `log_rotation_age`
- `log_rotation_size`
- `log_truncate_on_rotation`

When to Log

- `client_min_messages`
- `log_min_messages`
- `log_error_verbosity`
- `log_min_error_statement`
- `log_min_duration_statement`
- `silent_mode`

What to Log

- `debug_print_parse`
- `debug_print_rewritten`
- `debug_print_plan`
- `debug_print_prelim_plan`
- `debug_print_slice_table`
- `debug_pretty_print`
- `log_autostats`
- `log_connections`
- `log_disconnections`
- `log_duration`
- `log_statement`
- `log_timezone`
- `log_hostname`
- `gp_log_format`
- `gp_max_csv_line_length`
- `gp_log_gang`
- `gp_log_interconnect`
- `gp_debug_linger`
- `gp_reraise_signal`

Runtime Statistics Collection Parameters

These parameters control the PostgreSQL server statistics collection feature. When statistics collection is enabled, the data that is produced can be accessed via the `pg_stat` and `pg_statio` family of system catalog views.

- `stats_command_string`
- `update_process_title`
- `stats_start_collector`
- `stats_block_level`
- `stats_row_level`
- `stats_queue_level`
- `stats_reset_server_on_start`
- `log_statement_stats`
- `log_parser_stats`
- `log_planner_stats`
- `log_executor_stats`
- `log_dispatch_stats`



Note: Note that in Greenplum Database, the statistics that are collected are those of the master instance only, therefore enabling row-level (`stats_row_level`) or block-level (`stats_block_level`) statistics collection will not gather the relevant table statistics from all of the segments.

Automatic Statistics Collection Parameters

When automatic statistics collection is enabled, `ANALYZE` can be run automatically in the same transaction as an `INSERT`, `UPDATE`, `DELETE`, `COPY` or `CREATE TABLE...AS SELECT` statement when a certain threshold of rows is affected (`on_change`), or when a newly generated table has no statistics (`on_no_stats`). To enable this feature, set the following server configuration parameters in your Greenplum master `postgresql.conf` file and restart Greenplum Database:

- `gp_autostats_mode`
- `gp_autostats_on_change_threshold`
- `log_autostats`



Warning: Depending on the specific nature of your database operations, automatic statistics collection may have a negative performance impact. Carefully evaluate whether the default setting of `on_no_stats` is appropriate for your system.

Automatic Vacuuming Parameters

Standard PostgreSQL has a number of parameters for configuring the autovacuum daemon. This autovacuum daemon automatically initiates a `VACUUM` of the database at configured intervals.



Warning: Autovacuum is currently disabled in Greenplum Database. Enabling the `autovacuum` parameter will output an error.

The configuration parameters to configure automatic vacuuming are:

- `autovacuum`
- `autovacuum_analyze_scale_factor`
- `autovacuum_analyze_threshold`
- `autovacuum_freeze_max_age`
- `autovacuum_naptime`
- `autovacuum_vacuum_cost_delay`
- `autovacuum_vacuum_cost_limit`
- `autovacuum_vacuum_scale_factor`
- `autovacuum_vacuum_threshold`

Client Connection Default Parameters

Statement Behavior Parameters

- `search_path`
- `default_tablespace`
- `check_function_bodies`
- `default_transaction_isolation`
- `default_transaction_read_only`
- `statement_timeout`
- `vacuum_freeze_min_age`

Locale and Formatting Parameters

- `DateStyle`
- `IntervalStyle`
- `TimeZone`
- `extra_float_digits`
- `client_encoding`
- `lc_messages`
- `lc_monetary`
- `lc_numeric`
- `lc_time`

Other Client Default Parameters

- `dynamic_library_path`
- `gin_fuzzy_search_limit`
- `local_preload_libraries`
- `explain_pretty_print`

Lock Management Parameters

- `deadlock_timeout`
- `max_locks_per_transaction`

Workload Management Parameters

The following configuration parameters are used to configure the Greenplum Database workload management feature (resource queues) and control concurrency.

- `max_resource_queues`
- `max_resource_portals_per_transaction`
- `resource_scheduler`
- `resource_select_only`
- `resource_cleanup_gangs_on_wait`
- `stats_queue_level`
- `gp_vmem_protect_limit`
- `gp_vmem_protect_gang_cache_limit`

External Table Parameters

The following parameters are used to configure the external tables feature of Greenplum Database. See “[About External Tables and Web Tables](#)” on page 153.

- `gp_external_enable_exec`
- `gp_external_grant_privileges`
- `gp_external_max_segs`
- `gp_reject_percent_threshold`

Append-Only Table Parameters

The following parameters are used to configure the append-only tables feature of Greenplum Database. See “[Choosing the Table Storage Model](#)” on page 101 for more information about append-only tables.

- `max_appendonly_tables`

Past PostgreSQL Version Compatibility Parameters

The following parameters are for compatibility with older PostgreSQL versions. Most likely, you do not need to change these in Greenplum Database.

- `add_missing_from`
- `array_nulls`
- `backslash_quote`
- `default_with_oids`
- `escape_string_warning`
- `regex_flavor`
- `sql_inheritance`
- `standard_conforming_strings`
- `transform_null_equals`

Greenplum Array Configuration Parameters

The parameters in this section control the configuration of the Greenplum Database array and its various components (segments, master, distributed transaction manager, and interconnect).

Interconnect Configuration Parameters

- `gp_interconnect_hash_multiplier`
- `gp_interconnect_queue_depth`
- `gp_interconnect_setup_timeout`
- `gp_interconnect_type`
- `gp_max_packet_size`

Dispatch Configuration Parameters

- `gp_set_proc_affinity`
- `gp_connections_per_thread`
- `gp_cached_segworkers_threshold`
- `gp_segment_connect_timeout`

Fault Operation Parameters

- `gp_fault_action`
- `gp_set_read_only`
- `gp_fts_probe_interval`
- `gp_fts_probe_threadcount`

Greenplum Performance Monitor Parameters

- `gp_enable_gpperfmon`
- `gpperfmon_port`
- `gp_gpperfmon_send_interval`

Distributed Transaction Management Parameters

- `gp_max_local_distributed_cache`

Read-Only Parameters

- `gp_command_count`
- `gp_role`
- `gp_session_id`

20. Enabling High Availability Features

This chapter describes the high availability features of Greenplum Database, and explains the tasks involved for segment or master instance recovery. The following topics are covered in this chapter:

- [Overview of High Availability in Greenplum Database](#)
- [Enabling Mirroring in Greenplum Database](#)
- [Setting the Fault Operational Mode](#)
- [Knowing When a Segment is Down](#)
- [Recovering a Failed Segment](#)
- [Recovering a Failed Master](#)

Overview of High Availability in Greenplum Database

Greenplum Database provides several optional features to ensure maximum uptime and high-availability of your system. This section provides an overview of these features:

- [Overview of Segment Mirroring](#)
- [Overview of Master Mirroring](#)
- [Overview of Fault Detection and Recovery](#)

Overview of Segment Mirroring

Mirror segments allow database queries to fail over to a backup segment if the primary segment is unavailable. To configure mirroring, you must have enough nodes in your Greenplum Database system so that the mirror segment always resides on a different host than its primary. [Figure 20.1](#) shows how table data is distributed across the

segments when mirroring is configured. The mirror segment for a distributed table resides on a different host than its primary segment. Primary segments and mirror segments are served by different segment instances.

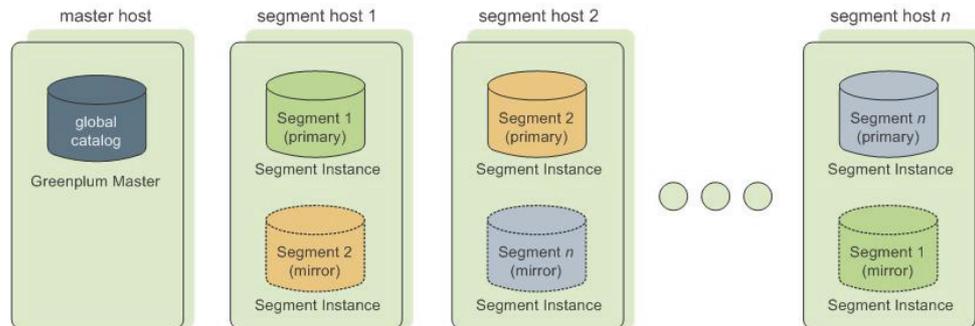


Figure 20.1 Data Mirroring in Greenplum Database

Overview of Master Mirroring

You can also optionally deploy a backup or ‘mirror’ of the master instance on a separate host machine. A backup master (or *standby master*) host serves as a ‘warm standby’ in the event of the primary master host becoming unoperational.

The standby master is kept up to date by a transaction log replication process (`gpsyncagent`), which runs on the standby master host and keeps the data between the primary and standby masters synchronized. Until a failure occurs in the primary master, no actual Greenplum Database master server is running on the standby master host -- only the replication process.

If the primary master fails, the log replication process is shut down, and the standby master can be activated in its place. Upon activation of the standby master, the replicated logs are used to reconstruct the state of the master host at the time of the last successfully committed transaction. The activated backup functions as the Greenplum Database master, accepting connections on the same port used by the failed primary master.

Since the master does not contain any user data, only the system catalog tables need to be synchronized between the primary and standby copies. These tables are not updated frequently, but when they are, changes are automatically copied over to the standby master so that it is always kept current with the primary.

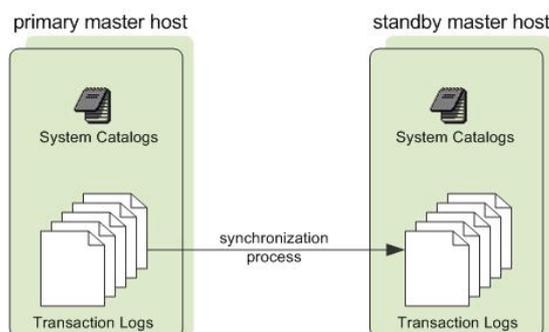


Figure 20.2 Master Mirroring in Greenplum Database

Overview of Fault Detection and Recovery

Fault detection is handled by a backend process named *fts_prober*. This process monitors the Greenplum array, scanning all segments and database processes at configurable intervals.

Whenever the master cannot connect to a segment instance, the *fts_prober* process marks that segment instance as *invalid* in the Greenplum Database system catalog. The segment instance will remain invalid and out of operation until steps are taken to bring that segment back online. Once a segment is back online, you can either manually mark the segment as valid again, or the master will mark it as valid again the next time it successfully connects to the segment.

When mirroring is enabled in a Greenplum Database system, the system will automatically failover to the mirror copy whenever a primary copy becomes unavailable. A Greenplum Database system can remain operational if a segment instance or host goes down as long as all portions of data are available on the remaining active segments.

Recovery of a failed segment depends on the configured *fault operational mode*. If the system is running in *read-only* mode (the default), users will not be able to issue DDL or DML commands when there are failed segments in the system. If the system is running in *continue* mode, all operations will continue as long as there is one active segment instance alive per portion of data. In this mode, the data on the failed segment must be reconciled with the active segment before it can be brought back into operation. The system must be shut down to recover failed segments for both *read-only* and *continue* mode, although the downtime is significantly shorter for a read-only mode recovery.

If you do not have mirroring enabled, the system will automatically shut down if a segment instance becomes invalid. You must recover all failed segments before operations can continue.

Enabling Mirroring in Greenplum Database

You can either configure your Greenplum Database system with mirroring at setup time, or enable mirroring in an existing system that was initially configured without mirroring. There are two types of mirroring you can configure:

- [Enabling Segment Mirroring](#)
- [Enabling Master Mirroring](#)

Enabling Segment Mirroring

Mirror segments allow database queries to fail over to a backup segment if the primary segment is unavailable. To configure mirroring, you must have enough nodes in your Greenplum Database system so that the mirror segment always resides on a different host than its primary. By default, mirrors are configured on the same array of hosts as your primary segments. You can also optionally choose a completely different set of hosts for your mirror segments so they do not share machines with your primary segments.

To add segment mirrors at setup time

1. Make sure that the Greenplum Database is installed on all hosts, and that you have allocated data storage areas for both primary and mirror data. See [“Installing the Greenplum Software”](#) on page 40.
2. Before initializing your Greenplum Database system, make sure you have set the configuration parameters for mirroring in the `gp_init_config` file. See [“Optional Parameters for Mirror Segments”](#) on page 790.
3. Initialize your Greenplum Database system with mirroring configured. See [“Initializing a Greenplum Database System”](#) on page 57.

To add segment mirrors to an existing system (same hosts as primaries)

1. Allocate the data storage area for mirror data on all segment hosts. See [“Creating the Data Storage Areas on the Segment Hosts”](#) on page 47.
2. The segment hosts must be able to SSH and SCP to each other without a password prompt. See [“Setting Up a Trusted Host Environment”](#) on page 44.
3. Run the `gpaddmirrors` utility to enable mirroring in your Greenplum Database system. For example (where `-p` specifies the number to add to your primary segment port numbers to calculate the mirror segment port numbers):

```
$ gpaddmirrors -p 1000
```

To add segment mirrors to an existing system (different hosts from primaries)

1. Allocate the data storage area for mirror data on all segment hosts. See [“Creating the Data Storage Areas on the Segment Hosts”](#) on page 47 for instructions.

2. The segment hosts must be able to SSH and SCP to each other without a password prompt. See [“Setting Up a Trusted Host Environment”](#) on page 44.
3. Create a configuration file that lists the host names, ports, and data directories where you want your mirrors to be created. You can create a sample configuration file to use as a starting point by running:

```
$ gpaddmirrors -o filename
```

The format of the mirror configuration file is:

```
mirror[content_id]=hostname:port:datadir_location
```

For example:

```
mirror[0]=sdw2:50100:/gpdata/m1
```

```
mirror[1]=sdw2:50101:/gpdata/m2
```

```
mirror[2]=sdw1:50100:/gpdata/m1
```

```
mirror[3]=sdw1:50101:/gpdata/m2
```

4. Run the `gpaddmirrors` utility to enable mirroring in your Greenplum Database system (where `-i` names the mirror configuration file you just created):

```
$ gpaddmirrors -i mirror_config_file
```

Enabling Master Mirroring

You can either configure your Greenplum Database system with a standby master at setup time, or enable it in an existing system that was initially configured without a standby master.

To add a standby master at setup time

1. Make sure you have installed the Greenplum Database software on both the primary and standby master hosts. See [“Installing Greenplum Database on the Master Host”](#) on page 40.
2. Initialize your Greenplum Database system with the optional standby master options. See [“Initializing Greenplum Database”](#) on page 57. For example:

```
$ gpinitssystem -c gp_init_config -s host09
```

To add a standby master to an existing system

1. Make sure the standby master host is installed and configured. See [“Installing Greenplum Database on the Master Host”](#) on page 40.
2. Run the `gpinitstandby` utility on the currently active *primary* master host. This will add a standby master host to your Greenplum Database system. For example (where `-s` specifies the standby master host name):

```
$ gpinitstandby -s host09
```

3. To switch operations to a standby master, see [“Recovering a Failed Master”](#) on page 193.

To check the status of the log synchronization process

If the synchronization process (`gpsyncagent`) fails on the standby master, it may not always be noticeable to users of the system. The `gp_master_mirroring` catalog is a place where Greenplum Database administrators can check to see if the standby master is currently synchronized. For example:

```
$ psql dbname -c 'SELECT * FROM gp_master_mirroring;'
```

If the result indicates that the standby master is “Not Synchronized,” check the `detail_state` and `error_message` columns to attempt to determine the cause of the errors.

To recover a standby master that has fallen behind:

```
$ gpinitstandby -s standby_master_hostname -n
```

Setting the Fault Operational Mode

The fault operational mode in Greenplum Database determines the behavior of the system in the event of a segment failure. If mirroring is enabled, a Greenplum Database system can remain operational if a segment instance or host goes down (as long as all portions of data are available on the remaining active segments). The default mode of operation is *readonly*, meaning the database cannot be modified when there are failed segment instances in the system.

The `gp_fault_action` parameter in the master host’s `postgresql.conf` file tells the Greenplum Database system how to behave when a segment instance goes down. The fault operational modes are as follows:

- **none** - Mirroring is not enabled. In this mode, the system will shut down if a segment instance becomes invalid. You must recover the failed segment before operations can continue.
- **readonly** - When any segment instance in the system is down, the system will not allow DDL or DML commands that alter data (`INSERT`, `UPDATE`, `DELETE`, etc.). In this mode, read-only operations will continue as long as there is one active segment instance per portion of data. The system must be shut down to recover failed segments, but the downtime is relatively short. This is the default mode.
- **continue** - The system will allow DDL and DML commands when there are failed segment instances in the system (read-write mode). In this mode, all operations will continue as long as there is one active segment instance alive per portion of data. The data on the failed segment must be reconciled with the active segment before it can be brought back into operation. The system must be shut down to recover failed segments.

If a segment and its corresponding mirror are down, or if mirroring is not enabled, then Greenplum Database will not be operational until the failed segments are recovered. See [“Recovering a Failed Segment”](#) on page 190.

To change the fault operational mode

1. Shut down your Greenplum Database system by running `gpstop` on the master host. For example:

```
$ gpstop
```

2. Edit the `postgresql.conf` file found in the master host data directory, and uncomment and set the `gp_fault_action` parameter. For example, to put the system in read-only mode:


```
gp_fault_action=readonly
```
3. Save and close the `postgresql.conf` file.
4. Restart your Greenplum Database system by running `gpstart` on the master host. For example:


```
$ gpstart
```

Knowing When a Segment is Down

If mirroring is enabled, Greenplum Database will automatically failover to a mirror segment when a primary segment goes down. As long as one segment instance is alive per portion of data, it will not always be apparent to users of the system that a segment instance is down.

Users may see errors in their client program indicating that a segment is down when they try to perform SQL commands. For example:

```
WARNING: Greenplum Database detected segment failure(s),
system is reconnected in read-write mode
```

```
ERROR: could not open relation 1663/16384/1259: No such file or
directory (seg1 slice4 localhost:5323 pid=30942)
```

If the entire Greenplum Database system becomes unoperational due to a segment failure (for example if mirroring is not enabled or there are not enough segments alive to serve all the portions of user data), users will see errors when trying to connect to a database. The errors returned to the client program may give some indication of the failure. For example:

```
FATAL: Not all primary segment instances are active and
connected (cdbgang.c:1364)server closed the connection
unexpectedly.
```

Checking for Failed Segments

With mirroring enabled, you may have failed segments in the system without interruption of service or any indication that a failure has occurred. One way to verify the status of your system is to use the `gpstate` utility. This utility provides the status of each individual component of a Greenplum Database system (primary segments, mirror segments, master, and standby master).

To check for failed segments

1. On the master, run the `gpstate` utility:


```
$ gpstate
```
2. Look in the output for any segment instances marked as *invalid*, *not running*, or *failed*.

3. For invalid or failed segment instances, note the host, port, primary or mirror status, and data directory. This will help you determine the host and segment instances to troubleshoot.
4. To see the primary to mirror segment instance mapping, run:


```
$ gpstate -c
```

Checking the Log Files

The log files may provide more information to help you determine the cause of an error. The master and each segment instance has its own log file, which is located in `pg_log` of its data directory. The master log file contains the most information and should always be checked first.

You can use the `gplogfilter` utility to check the Greenplum Database log files for any additional information. To check the segment log files, you can run `gplogfilter` on the segment hosts using `gpssh`.

To check the log files

1. Check the master log file for `WARNING`, `ERROR`, `FATAL` or `PANIC` log level messages:


```
$ gplogfilter -t
```
2. Using `gpssh`, check for `WARNING`, `ERROR`, `FATAL` or `PANIC` log level messages on each segment instance:


```
$ gpssh -f seg_hosts_file -e 'source /usr/local/greenplum-db/greenplum_path.sh ; gplogfilter -t /gpdata/*/pg_log/gpdb*.log' > seglog.out
```

Recovering a Failed Segment

Whenever the master cannot connect to a segment instance, it marks that segment instance as *invalid* in the Greenplum Database system catalog. The segment instance will remain invalid and out of operation until steps are taken to bring that segment back online.

The process for recovering a failed segment instance or host depends on the cause of the failure and whether or not you have mirroring enabled. A segment instance can become unavailable for several reasons, such as:

- A segment host is not available (network errors, hardware failures).
- A segment instance is not running (`postgres` database listener process was killed).
- The data directory of the segment instance is corrupted or missing (data not accessible, corrupted file system, disk failure).

The following chart shows the high-level steps to take for each of these failure scenarios.

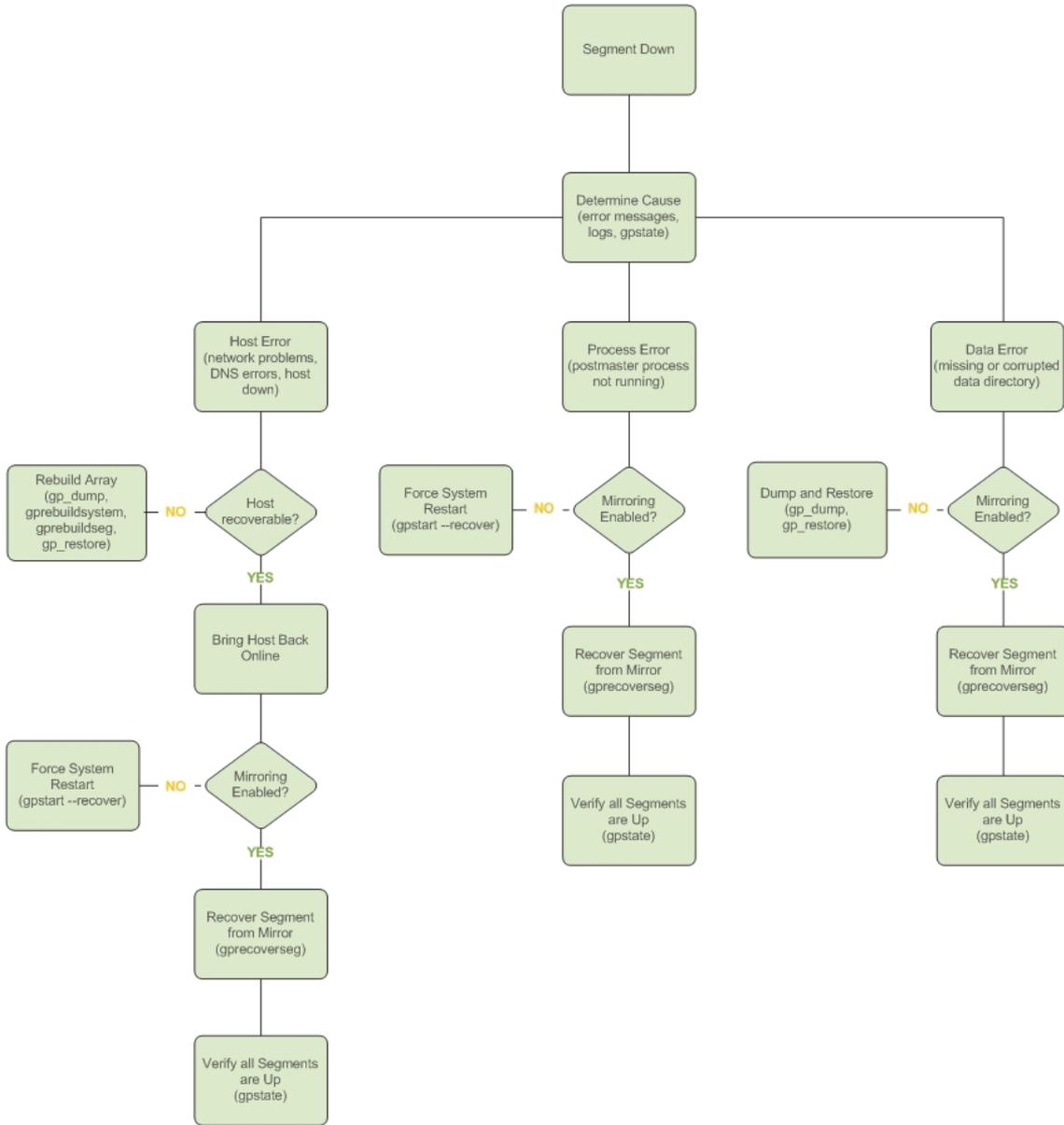


Figure 20.3 Segment Failure Troubleshooting Matrix

Recovering From Segment Failures

When a segment host goes down, this could possibly make several segment instances inactive, as all primary and mirror segments on that host will be marked as invalid and taken out of operation. If your Greenplum Database system was deployed without mirroring enabled, the system will automatically become unoperational whenever a segment host goes down.

To recover with mirroring enabled

1. First, make sure that you can connect to the segment host from the master host.
For example:

```
$ ssh failed_seg_host
```
2. Troubleshoot the problem that is preventing the master host from connecting to the segment host. For example, the host machine may need to be restarted.
3. Once the host is up again, and you have verified that you can connect to it, run the `gprecoverseg` utility to reactivate the failed segment instances on that host. For example:

```
$ gprecoverseg
```
4. The recovery process will need to shut down the system for both read-only and continue mode. The system cannot recover failed segments without shutting down.
5. When the recovery process is complete, run the `gpstate` utility to verify that all segment instances are up:

```
$ gpstate
```

To recover without mirroring enabled

1. First, make sure that you can connect to the segment host from the master host.
For example:

```
$ ssh failed_seg_host
```
2. Troubleshoot the problem that is preventing the master host from connecting to the segment host. For example, the host machine may need to be restarted.
3. Once the host is up again, and you have verified that you can connect to it, do a forced restart of the Greenplum Database system. For example:

```
$ gpstart --recover
```
4. Run the `gpstate` utility to verify that all segment instances are up:

```
$ gpstate
```

When a segment host is not recoverable

If a host is no longer operational (due to hardware failures, for example) you will need to rebuild your Greenplum Database system using alternate hosts. See [“Rebuilding a New Greenplum System From Backup”](#) on page 203.

Recovering from a Data Corruption Failure

If a disk fails, a file system becomes corrupted, or a data directory gets accidentally deleted, the segment instance serving that data becomes unoperational.

To recover with mirroring enabled

1. Troubleshoot and correct the problem that caused the failure. For example, you may need to replace the disk or rebuild the file system.

2. Run the `gprecoverseg` utility to recover the failed segment instances. For example:


```
$ gprecoverseg
```
3. The recovery process will shut down the system in either read-only or continue mode. The system cannot recover failed segments without shutting down.
4. When the recovery process is complete, run the `gpstate` utility to verify that all segment instances are up:


```
$ gpstate
```

To recover without mirroring enabled

If you do not have mirroring enabled, there is no way to recover a single segment that has corrupted or missing data. In this scenario, you will have to restore the entire database from your most recent Greenplum Database backup to ensure data integrity across all segments. See “[Backing Up a Database](#)” on page 197 and “[Restoring From Parallel Backup Files](#)” on page 200.

Recovering a Failed Master

If the primary master fails, the log replication process is shut down, and the standby master can be activated in its place by using the `gpactivatestandby` utility. Upon activation of the standby master, the replicated logs are used to reconstruct the state of the master host at the time of the last successfully committed transaction. You can also specify a new standby master host when you activate your currently configured standby master.

To activate the standby master

1. First, you must have a standby master host configured for the system. See “[Enabling Master Mirroring](#)” on page 187.
2. Run the `gpactivatestandby` utility from the standby master host you are activating. For example, where `-d` specifies the data directory of the master host you are activating:

```
$ gpactivatestandby -d /gpdata
```

Note that once you activate your standby, it then becomes the *active* or *primary* master for your Greenplum Database array. If you want to configure another host to be your new standby at this time, you can optionally use the `-c` option when running the `gpactivatestandby` utility. For example:

```
$ gpactivatestandby -d /gpdata -c new_standby_hostname
```

3. After the utility has finished, run `gpstate` to check the status:

```
$ gpstate -f
```

The activated master should have *Active* status, and if you configured a new standby host, it should have *Passive* status (if not configured, the status is displayed as *Not Configured*):

```
Master instance = Active
```

```
Master instance standby = Passive
```

4. After switching over, run `ANALYZE` on the newly active master host. For example:

```
$ psql dbname -c 'ANALYZE;'
```

5. Check the `gp_master_mirroring` system catalog table to see when the standby master was last updated. For example:

```
$ psql dbname -c 'SELECT * FROM gp_master_mirroring;'
```

6. (optional) If you did not specify a new standby host when activating, you can use `gpinitstandby` to configure a new standby master at a later time. Run this utility on your currently active master host. For example:

```
$ gpinitstandby -s new_standby_master_hostname
```

To resynchronize a standby master

There may be times when the log synchronization process between the primary and standby master has stopped or has fallen behind, and your standby master is then out of date. To recover a standby master and bring it up to date again, run the following `gpinitstandby` command (using the `-n` option):

```
$ gpinitstandby -s standby_master_hostname -n
```

21. Backing Up and Restoring Databases

This chapter provides information on backing up and restoring databases and user data in a Greenplum Database system. It contains the following topics:

- [Overview of Backup and Restore Operations](#)
- [Backing Up a Database](#)
- [Restoring From Parallel Backup Files](#)

Overview of Backup and Restore Operations

Greenplum recommends that you take regular backups of your databases. These backups can be used to restore your data or to rebuild a Greenplum Database system in the event of a system failure or data corruption. You can also use the backups to migrate data from one Greenplum Database system to another.

About Parallel Backups

Greenplum provides a parallel dump utility called `gp_dump`. This utility backs up the Greenplum master instance and each active segment instance at the same time. See “[Backing Up a Database with gp_dump](#)” on page 198.

Because the segments are dumped in parallel, the time it takes to do a backup should scale regardless of the number of segments in your system. The dump files on the master host consist of DDL statements and the Greenplum-specific system catalog tables (such as `gp_configuration`). On the segment hosts, there is one dump file created for each segment instance. The segment dump files contain the data for an individual segment instance. All of the dump files that comprise a total backup set are identified by a unique 14-digit timestamp key.

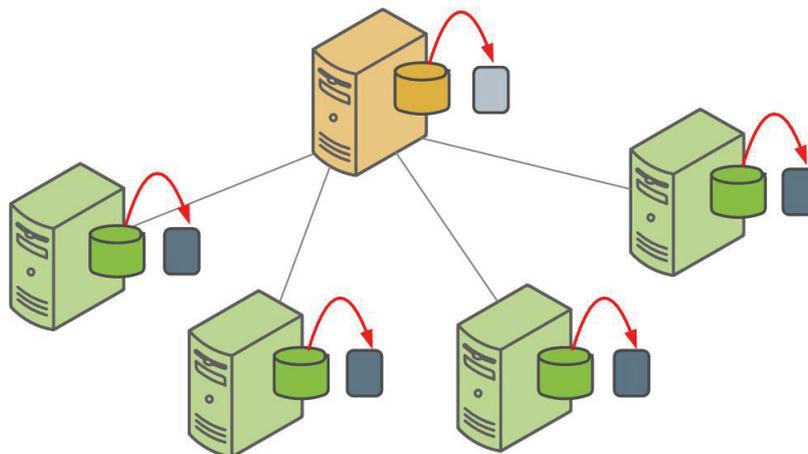


Figure 21.1 Parallel Backups in Greenplum Database

In order to automate routine backups, Greenplum also provides the `gpccrondump` utility, which is a wrapper for `gp_dump` that can be called directly or from a scheduled CRON job. `gpccrondump` also allows you to backup additional objects besides your databases and data, such as database roles and server configuration files. See “Automating Parallel Backups with `gpccrondump`” on page 199.

About Non-Parallel Backups

Greenplum also supports the regular PostgreSQL dump utilities: `pg_dump` and `pg_dumpall`. The PostgreSQL dump utilities (when used on Greenplum Database) will create one big dump file on the master host containing the data from all active segments. In most cases, this is probably not practical, as there is most likely not enough disk space on the master host to create a single backup file of an entire distributed database. These utilities are mostly supported for users who are migrating from regular PostgreSQL to Greenplum Database.

Another useful command for getting data out of a database is the `COPY TO SQL` command. This allows you to copy all or a portion of a table out of the database to a text-delimited file on the master host.

If you are migrating your data to another Greenplum Database system with a different segment configuration (for example, if the system you are migrating to has greater or fewer segment instances), Greenplum recommends using your parallel dump files created by `gp_dump` or `gpccrondump` and following the restore process described in “Restoring to a Different Greenplum System Configuration” on page 202.

About Parallel Restores

To do a parallel restore, you must have a complete backup set created by `gp_dump` or `gpccrondump`. Greenplum provides a parallel restore utility called `gp_restore`. This utility takes the timestamp key generated by `gp_dump`, validates the backup set, and restores the database objects and data into a distributed database. As with a parallel dump, each segment’s data is restored in parallel.

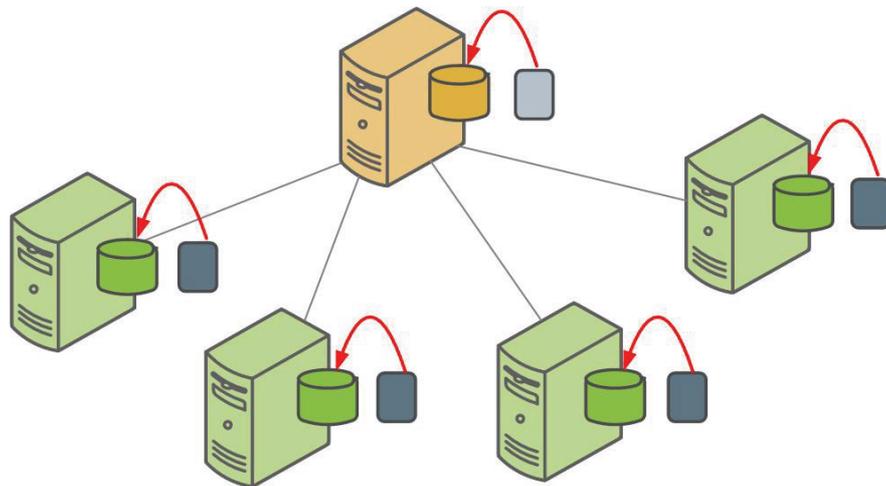


Figure 21.2 Parallel Restores in Greenplum Database

Greenplum also provides the `gpdbrestore` utility, which is a wrapper for `gp_restore`. `gpdbrestore` provides additional flexibility and verification options, which are useful if you are using automated backup files produced by `gpcrondump`, or have moved your backup files off of the Greenplum array to an alternate location. See “Restoring a Database Using `gpdbrestore`” on page 201.

About Non-Parallel Restores

Greenplum also supports the regular PostgreSQL restore utility: `pg_restore`. This utility is mostly supported for users who are migrating to Greenplum Database from regular PostgreSQL, and have compressed dump files created by `pg_dump` or `pg_dumpall`. Before restoring PostgreSQL dump files into Greenplum Database, make sure to modify the `CREATE TABLE` statements in the dump files to include the Greenplum `DISTRIBUTED` clause.

It may also sometimes be necessary to do a non-parallel restore from a parallel backup set. For example, suppose you are migrating from a Greenplum system that has four segments to one that has five segments. You cannot do a parallel restore in this case, because your backup set only has four backup files and would not be evenly distributed across the new expanded system. A non-parallel restore using parallel backup files involves collecting each backup file from the segment hosts, copying them to the master host, and loading them through the master. See “Restoring to a Different Greenplum System Configuration” on page 202.

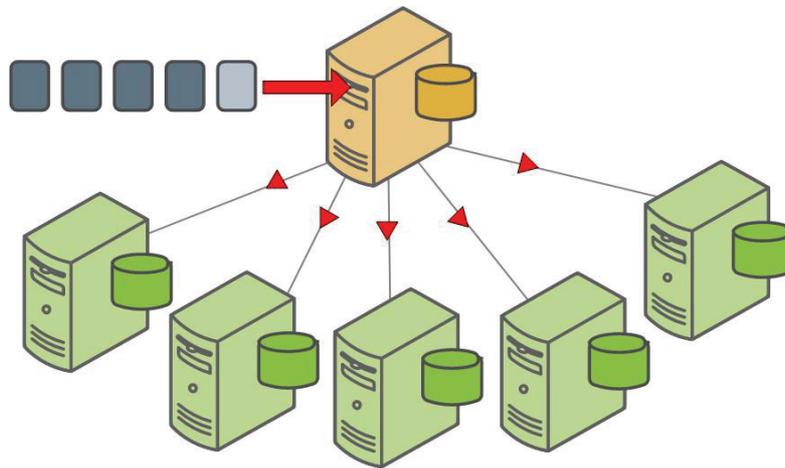


Figure 21.3 Non-parallel Restore Using Parallel Backup Files

Backing Up a Database

There are three ways to backup a database:

- **Create a dump file for each segment with `gp_dump`.** Use this option if you are dumping a database for backup purposes, or if you want to migrate your data to another system with the same segment configuration (for example, same number of segment instances, but on different hosts). To restore, you must use the corresponding `gp_restore` utility. See “[Backing Up a Database with `gp_dump`](#)” on page 198. You can also use dump files created by `gp_dump` to restore to a different Greenplum system configuration if needed.
- **Schedule routine dumps with `gpcrondump`.** This is a wrapper utility for `gp_dump`. It allows you to schedule routine backups of a Greenplum Database database using `cron` (a scheduling utility for Unix operating systems). Cron jobs that call `gpcrondump` should be scheduled on the Greenplum master host. `gpcrondump` also allows you to backup additional objects besides your databases and data, such as database roles and server configuration files.
- **Create a single dump file with `pg_dump` or `pg_dumpall`.** Use this option if you are migrating your data to another database vendor’s system. If restoring to a PostgreSQL or Greenplum database, you can use the corresponding `pg_restore` utility (if the dump file is in archive format), or you can use a client program such as `psql` (if the dump file is in plain text format). If you plan to restore to another Greenplum Database system, Greenplum recommends that you do a parallel dump using `gp_dump` or `gpcrondump` and then do a non-parallel restore.

Backing Up a Database with `gp_dump`

The `gp_dump` utility dumps the contents of a Greenplum Database system into a series of SQL utility files, which can then be used to restore a Greenplum Database system configuration, database, and data. During a dump operation users will still have access to the database.

The `gp_dump` utility performs the following actions and produces the following dump files:

On the master host

- Dumps the Greenplum configuration system catalog tables into a SQL file in the master data directory. The default naming convention of this file is `gp_catalog_1_<dbid>_<timestamp>`. This file can be used by the `gprebuildsystem` utility to recover an entire Greenplum Database system.
- Dumps a `CREATE DATABASE` SQL statement into a file in the master data directory. The default naming convention of this file is `gp_cdatabase_1_<dbid>_<timestamp>`. This statement can be run on the master instance to recreate the database.
- Dumps the database schema(s) into a SQL file in the master data directory. The default naming convention of this file is `gp_dump_1_<dbid>_<timestamp>`. This file is used to recreate the database objects.
- Creates a log file in the master data directory named `gp_dump_status_1_<dbid>_<timestamp>`.
- `gp_dump` launches a `gp_dump_agent` for each segment instance to be backed up. `gp_dump_agent` processes run on the segment hosts and report status back to the `gp_dump` process running on the master host.

On the segment hosts

- Dumps the data for each segment instance into a SQL file in the segment instance's data directory. By default, only primary (or active) segment instances are backed up. The default naming convention of this file is `gp_dump_0_<dbid>_<timestamp>`. This file is used to recreate that particular segment of data.
- Creates a log file in each segment instance's data directory named `gp_dump_status_0_<dbid>_<timestamp>`.

Note that the 14 digit timestamp is the number that uniquely identifies the backup job, and is part of the filename for each dump file created by a `gp_dump` operation. This timestamp must be passed to the `gp_restore` utility when restoring a Greenplum database.

To backup a Greenplum database using `gp_dump`

1. From the master, run the `gp_dump` utility. For example (where `mydatabase` is the name of the database you are backing up):

```
$ gp_dump mydatabase
```

Automating Parallel Backups with `gpcrondump`

`gpcrondump` is a wrapper utility for `gp_dump`, which can be called directly or from a `crontab` entry. It also allows you to backup additional objects besides your databases and data, such as database roles and server configuration files.

`gpcrondump` creates the dump files in the master and each segment instance's data directory in `<data_directory>/db_dumps/YYYYMMDD`. The segment data dump files are compressed using `gzip`.

To schedule a dump operation using CRON

1. On the master, log in as the Greenplum super user (`gpadmin`).
2. Define a `crontab` entry that calls `gpcrondump`. For example, to schedule a nightly dump of the `sales` database at one minute past midnight (note that the `SHELL` is set to `/bin/bash` and the `PATH` includes the location of the Greenplum Database management utilities):

Linux Example:

```
SHELL=/bin/bash
GPHOME=/usr/local/greenplum-db-3.3.7.x
MASTER_DATA_DIRECTORY=/data/gpdb_p1/gp-1
PATH=$PATH:$GPHOME/bin
01 0 * * * gpadmin gpcrondump -x sales -c -g -G -a -q >>
gp_salesdump.log
```

Solaris Example (no line breaks):

```
01 0 * * * SHELL=/bin/bash
GPHOME=/usr/local/greenplum-db-3.3.7.x PATH=$PATH:$GPHOME/bin
HOME=/export/home/gpadmin
MASTER_DATA_DIRECTORY=/data/gpdb_p1/gp-1
```

```
/usr/local/greenplum-db/bin/gpcrondump -x sales -c -g -G -a
-q >> gp_salesdump.log
```

3. Create a file named `mail_contacts` in either the Greenplum superuser's home directory or in `$GPHOME/bin`. For example:

```
$ vi /home/gpadmin/mail_contacts
$ vi /export/home/gpadmin/mail_contacts
```

4. In this file, type one email address per line. For example:

```
dba@mycompany.com
jjones@mycompany.com
```

5. Save and close the `mail_contacts` file. `gpcrondump` will send email notifications to the email addresses listed in this file.

Restoring From Parallel Backup Files

The procedure for restoring a database from parallel backup files depends on a few factors. To determine the restore procedure to use, determine your answers to the following questions:

1. **Where are your backup files located?** If your backup files reside in their original location on the segment hosts where they were created by `gp_dump`, you can do a simple restore using `gp_restore`. If you have moved your backup files off of your Greenplum array, for example to an archive server, use `gpdbrstore`.
2. **Do you need to restore your entire system, or just your data?** If you have your Greenplum Database up and running and just need to restore your data, you can do a restore using `gp_restore` or `gpdbrstore`. If you have lost your entire array and need to rebuild the entire system from backup, use `gprebuildsystem`.
3. **Are you restoring to a system with the same number of segment instances as your backup set?** If you are restoring to an array with the same number of segment hosts and segment instances per host, use `gp_restore` or `gpdbrstore`. If you are migrating to a different array configuration, you must do a non-parallel restore. See “Restoring to a Different Greenplum System Configuration” on page 202.

Restoring a Database with `gp_restore`

The `gp_restore` utility recreates the data definitions (schema) and user data in a database using the backup files created by a `gp_dump` operation. To do a restore, you must have:

1. Backup files created by a `gp_dump` operation.
2. The backup files reside on the segment hosts in the location where `gp_dump` created them.
3. The Greenplum Database system up and running.

4. A Greenplum Database system with the *exact* same number of primary segment instances as the system that was backed up using `gp_dump`.
5. The database you are restoring to is created in the system.
6. If you used the options `-s` (schema only), `-a` (data only), `--gp-c` (compressed), `--gp-d` (alternate dump file location) when performing the `gp_dump` operation, you must specify these options when doing the `gp_restore` as well.

The `gp_restore` utility performs the following actions:

On the master host

- Runs the SQL DDL commands in the `gp_dump_1_<dbid>_<timestamp>` file created by `gp_dump` to recreate the database schema and objects.
- Creates a log file in the master data directory named `gp_restore_status_1_<dbid>_<timestamp>`.
- `gp_restore` launches a `gp_restore_agent` for each segment instance to be restored. `gp_restore_agent` processes run on the segment hosts and report status back to the `gp_restore` process running on the master host.

On the segment hosts

- Restores the user data for each segment instance using the `gp_dump_1_<dbid>_<timestamp>` files created by `gp_dump`. Each primary and mirror segment instance on a host is restored.
- Creates a log file for each segment instance named `gp_restore_status_1_<dbid>_<timestamp>`.

Note that the 14 digit timestamp is the number that uniquely identifies the backup job to be restored, and is part of the filename for each dump file created by a `gp_dump` operation. This timestamp must be passed to the `gp_restore` utility when restoring a database.

To restore from a backup created by `gp_dump`

1. Make sure the backup files created by `gp_dump` reside on the master host and segment hosts for the Greenplum Database system you are restoring.
2. Make sure the database you are restoring to has been created in the system. For example:


```
$ createdb database_name
```
3. From the master, run the `gp_restore` utility. For example (where `--gp-k` specifies the timestamp key of the backup job and `-d` specifies the database to connect to):

```
$ gp_restore -gp-k=2007103112453 -d database_name
```

Restoring a Database Using `gpdbrestore`

The `gpdbrestore` utility is a wrapper around `gp_restore`, which provides some convenience and flexibility in restoring from a set of backup files created by `gpcrondump`. To do a restore using `gpdbrestore`, you must have:

1. Backup files created by a `gpccrondump` operation.
2. The Greenplum Database system up and running.
3. A Greenplum Database system with the *exact* same number of primary segment instances as the system that was backed up.
4. The database you are restoring to is created in the system.

To restore from an archive host using `gpdrestore`

(This procedure assumes your backup set has been moved off of your Greenplum array to another host in the network)

1. First, make sure that the archive host is reachable from the Greenplum master host:


```
$ ping archive_host
```
2. Make sure the database you are restoring to has been created in the system. For example:


```
$ createdb database_name
```
3. From the master, run the `gpdrestore` utility. For example (where `-R` specifies the host name and path to a complete backup set):


```
$ gpdrestore -R archive_host:/gpdb/backups/archive/20080714
```

Restoring to a Different Greenplum System Configuration

In order to do a parallel restore operation using `gp_restore` or `gpdrestore`, the system you are restoring to must be the same configuration as the system that was backed up. If you want to restore your database objects and data into a different system configuration (for example, if you are expanding to a system with more segments), you can still use your parallel backup files and restore them by loading them through the Greenplum master. To do a non-parallel restore, you must have:

1. A complete backup set created by a `gp_dump` or `gpccrondump` operation. The backup file of the master contains the DDL to recreate your database objects. The backup files of the segments contain the data.
2. A Greenplum Database system up and running.
3. The database you are restoring to is created in the system.

If you look at the contents of a segment dump file, it simply contains a `COPY` command for each table followed by the data in delimited text format. If you collect all of the dump files for all of the segment instances and run them through the master, you will have restored all of your data and redistributed it across the new system configuration.

To restore a database to a different system configuration

1. First make sure you have a complete backup set. This includes a dump file of the master (`gp_dump_1_1_<timestamp>`) and one for each segment instance (`gp_dump_0_2_<timestamp>`, `gp_dump_0_3_<timestamp>`,

`gp_dump_0_4_<timestamp>`, and so on). The individual dump files should all have the same timestamp key. By default, `gp_dump` creates the dump files in each segment instance's data directory, so you will need to collect all of the dump files and move them to a place on the master host. If you do not have a lot of disk space on the master, you can copy each segment dump file to the master, load it, and then delete it once it has loaded successfully.

2. Make sure the database you are restoring to has been created in the system. For example:

```
$ createdb database_name
```

3. Load the master dump file to restore the database objects. For example:

```
$ psql database_name -f /gpdb/backups/gp_dump_1_1_20080714
```

4. Load each segment dump file to restore the data. For example:

```
$ psql database_name -f /gpdb/backups/gp_dump_0_2_20080714
```

```
$ psql database_name -f /gpdb/backups/gp_dump_0_3_20080714
```

```
$ psql database_name -f /gpdb/backups/gp_dump_0_4_20080714
```

```
$ psql database_name -f /gpdb/backups/gp_dump_0_5_20080714
```

```
...
```

Rebuilding a New Greenplum System From Backup

In some cases, you may need to rebuild your entire Greenplum Database system from a set of backup files. You can either rebuild on the same array of machines as your original system, or on a different array. The only requirement is that the Greenplum Database system you are rebuilding has the *exact* same number of segment instances as the system that was backed up using `gp_dump`.

If you need to rebuild your system on a completely different segment configuration, reinitialize a new array using `gpinitssystem`, then do a non-parallel restore as described in “[Restoring to a Different Greenplum System Configuration](#)” on page 202.

To rebuild your Greenplum array

1. First, you must have a backup of your data. It is good practice to make backups of your databases on a regular basis. See “[Backing Up a Database](#)” on page 197.
2. Make sure the Greenplum Database software is installed and configured on all hosts in the array. See “[Installing the Greenplum Software](#)” on page 40.
3. Use the `gprebuildsystem` utility to reconstruct your Greenplum Database system from the catalog backup file created by `gp_dump`. For example:

```
$ gprebuildsystem -f gp_catalog_0_1_2007103112453 -d /tmp
```

If rebuilding the array on a different set of hosts than the system that was originally backed up, use the `-c` option. The utility will then prompt you for the new segment host configuration information. For example:

```
$ gprebuildsystem -f gp_catalog_0_1_2007103112453 -d /tmp -c
```

4. You should now have your reconfigured Greenplum Database system up and running, and ready to restore your data from your backup files. See [“Restoring a Database with gp_restore”](#) on page 200 or [“Restoring a Database Using gpdbrestore”](#) on page 201.

22. Expanding a Greenplum System

This chapter provides information on adding additional resources to an existing Greenplum Database system in order to scale performance and storage capacity. It contains the following topics:

- [Planning Greenplum System Expansion](#)
- [Preparing and Adding Nodes](#)
- [Initializing New Segments](#)
- [Redistributing Tables](#)
- [Removing the Expansion Schema](#)

Though this chapter provides some general information on preparing hardware platforms, it focuses chiefly on software aspects of expansion. Greenplum recommends that you work with our platform engineers when configuring hardware resources for expanding Greenplum Database.

Planning Greenplum System Expansion

Careful planning is critical to the success of a system expansion operation. By thoroughly preparing all new hardware and carefully planning all the steps of the expansion procedure, you can minimize risk and down time for Greenplum Database.

This section provides an overview and a checklist for the system expansion process.

System Expansion Overview

System expansion consists of three phases:

- Adding and testing new hardware platforms
- Initializing new segments
- Redistributing tables

Adding and testing new hardware — General considerations for standing up new hardware are described in “[Planning New Hardware Platforms](#)” on page 207. For more information on hardware platforms, consult Greenplum platform engineers. After the new hardware platforms are provisioned and networked, you must run performance tests using Greenplum utilities.

Initializing new segments— Once Greenplum Database is installed on new hardware, you must initialize the new segments using `gpexpand` (not `gpinitssystem`). In this process, the utility creates a data directory and copies all user tables on the new segments, capturing metadata for each table in an expansion schema for status tracking. Optionally, you can initialize additional segments on existing hosts during this process, increasing the number of segments per host across the expanded array. These operations are performed with the system offline. The `gpexpand` utility will shut down the database during initialization if you have not already done so.

Redistributing tables — As part of the initialization process, `gpexpand` nullifies hash distribution policies (except for the parent tables of a partitioned table) and sets the policy for all tables to random distribution. Users can continue to access the Greenplum Database after initialization is complete and the system is back online, though they may experience some performance degradation on systems that rely heavily on hash distribution of tables.

To complete system expansion, you must run `gpexpand` to redistribute data tables across the newly added segments. Depending on the size and scale of your system, this might be accomplished in a single session during low-use hours, or it might require you to divide the process into batches over an extended period. Each table or partition will be unavailable for read or write operations during the period in which it is being redistributed. As each table is successfully redistributed across the new segments according to its distribution key (if any), the performance of the database should incrementally improve until it equals and then exceeds pre-expansion performance levels.

In a typical operation, you will run the `gpexpand` utility four times with different options during the complete expansion process.

- To interactively create an expansion input file:
`gpexpand -f hosts_file`
- To initialize segments and create expansion schema:
`gpexpand -i input_file -D database_name`
- To redistribute tables:
`gpexpand -d duration`
- To remove the expansion schema:
`gpexpand -c`

In systems whose large scale requires multiple redistribution sessions, `gpexpand` may need to be run several more times to complete the expansion. For more information, see the utility reference for “`gpexpand`” on page 619.

System Expansion Checklist

This checklist provides a quick overview of the steps required for a system expansion.

Online Pre-Expansion Preparation	
<i>* System is up and available</i>	
<input type="checkbox"/>	Devise and execute a plan for ordering, building, and networking new hardware platforms.
<input type="checkbox"/>	Devise a database expansion plan. Map the number of segments per host, schedule the offline period for testing performance and creating the expansion schema, and schedule the intervals for table redistribution.
<input type="checkbox"/>	Install Greenplum Database binaries on new hosts.

<input type="checkbox"/>	Copy SSH keys to the new hosts (<code>gpssh-exkeys</code>).
<input type="checkbox"/>	Validate the operating system environment of the new hardware (<code>gpcheckos</code>).
<input type="checkbox"/>	Validate disk I/O and memory bandwidth of the new hardware (<code>gpcheckperf</code>).
<input type="checkbox"/>	Prepare an expansion input file (<code>gpexpand</code>).
Offline Expansion Tasks	
<i>* The system will be locked down to all user activity during this process</i>	
<input type="checkbox"/>	Validate the operating system environment of the combined existing and new hardware (<code>gpcheckos</code>).
<input type="checkbox"/>	Validate disk I/O and memory bandwidth of the combined existing and new hardware (<code>gpcheckperf</code>).
<input type="checkbox"/>	Initialize new segments into the array and create an expansion schema (<code>gpexpand -i input_file</code>).
Online Expansion and Table Redistribution	
<i>* System is up and available</i>	
<input type="checkbox"/>	Redistribute tables through the expanded system (<code>gpexpand</code>).
<input type="checkbox"/>	Remove expansion schema (<code>gpexpand -c</code>).

Planning New Hardware Platforms

Careful preparation of new hardware for system expansion is extremely important. Deliberate and thorough deployment of compatible hardware can greatly minimize the risk of issues developing later in the system expansion process.

All new segment hosts for the expanded Greenplum Database array should have hardware resources and configurations matching those of the existing hosts. For more information and details about hardware configuration, see “[About Greenplum and Hardware Platforms](#)” on page 30. Greenplum recommends that you work with our platform engineers prior to making a hardware purchase decision for expanding Greenplum Database.

The steps to plan and set up new hardware platforms will vary greatly for each unique deployment. Some of the possible considerations include:

- Preparing the physical space for the new hardware. Consider cooling, power supply, and other physical factors.
- Determining the physical networking and cabling required to connect the new and existing hardware.

- Mapping the existing IP address spaces and developing a networking plan for the expanded system.
- Capturing the system configuration (users, profiles, NICs, etc.) from existing hardware to list it in detail for ordering the new hardware.
- Creating a custom build plan for deploying hardware with the desired configuration in the particular site and environment.

After selecting and adding new hardware to your network environment, make sure you perform the burn-in tasks described in [“Verifying OS Settings”](#) on page 213.

Planning Initialization of New Segments

Expanding Greenplum Database requires a limited period of system down time. During this period, you must run `gpexpand` to initialize new segments into the array and create an expansion schema.

The time required will depend on the number of schema objects in the Greenplum system, and other factors related to hardware performance. In most environments, the initialization of new segments will require less than thirty minutes offline.



Note: After you begin initializing new segments, you can no longer restore the system using `gp_dump` files created for the pre-expansion system.

Planning Mirror Segments

If your existing array has mirror segments, the new segments are required to have mirroring configured. Conversely, if there are no mirrors configured for existing segments, you cannot add mirrors to new hosts with the `gpexpand` utility.

For Greenplum Database arrays with mirror segments, you must make sure that you have added enough new host machines to accommodate the new mirror segments. The number of new hosts required depends on your mirroring strategy:

Spread Mirroring — add at least one more host to the array than the number of segments per host. The number of separate hosts must be greater than the number of segment instances per host to ensure even spreading.

Grouped Mirroring — add no fewer than two new hosts. In a minimum case with two hosts, this ensures that the mirrors for the first host can reside on the second host, and vice versa.

For more information, see [“About Segment Mirroring”](#) on page 8.

Increasing Segments Per Host

By default, new hosts are initialized with the same number of primary segments as existing hosts. Optionally, you can increase the number of segments per host, or add new segments only to existing hosts.

For example, if existing hosts currently have two segments per host, you can use `gpexpand` to initialize two additional segments on existing hosts (for a total of four), and four new segments on new hosts.

The interactive process for creating an expansion input file prompts for this option, and the input file format allows you to specify new segment directories manually as well. For more information, see [“Creating an Input File for System Expansion”](#) on page 214.

About the Expansion Schema

At initialization time, `gpexpand` creates an expansion schema. If you do not specify a particular database at initialization time (`gpexpand -D`), the schema is created in the database indicated by the `PGDATABASE` environment variable.

The expansion schema stores metadata for each table in the system so that its status can be tracked throughout the expansion process. It consists of two tables and a view for tracking the progress of an expansion operation:

- `gpexpand.status`
- `gpexpand.status_detail`
- `gpexpand.expansion_progress`

You can control aspects of the expansion process by modifying `gpexpand.status_detail`. For example, removing a record from this table prevents the table from being expanded across new segments. By updating the `rank` value for a record, you can control the order in which tables are processed for redistribution. For more information, see [“Ranking Tables for Redistribution”](#) on page 218.

Planning Table Redistribution

The redistribution of tables is performed with the system online. For many Greenplum systems, table redistribution can be completed in a single `gpexpand` session scheduled during a low-use period. Larger systems may require you to plan multiple sessions and to set the order of table redistribution in order to minimize the performance impact. Greenplum recommends completing the table redistribution in one session if your database size and design permit it.



Important: To perform table redistribution, you must have enough disk space on your segment hosts to temporarily hold a copy of your largest table. Each table is unavailable for read and write operations while `gpexpand` is redistributing it among the segments.

The performance impact of table redistribution depends on the size, storage type, and partitioning design of a table. Redistributing a table with `gpexpand` takes approximately as much time per table as a `CREATE TABLE AS SELECT` operation would take. When redistributing a terabyte-scale fact table, the expansion utility may use a significant portion of available system resources, with resulting impact on the performance of queries or other database workload.

Managing Redistribution in Large-Scale Greenplum Systems

You can manage the order in which tables are redistributed by adjusting their ranking as described in “[Ranking Tables for Redistribution](#)” on page 218. Manipulating the redistribution order can help you adjust for limited disk space and restore optimal query performance more quickly.

In systems with abundant free disk space (required to store a copy of the largest table), you can focus on restoring optimum query performance as soon as possible by first redistributing important tables that are heavily used by common queries. Accordingly, assign high ranking to these tables, and schedule the redistribution operations for times of low system usage.

If your existing hosts have limited disk space, you may prefer to first redistribute smaller tables (such as dimension tables) in order to clear the space needed to store a copy of the largest table. Disk space on the original segments will increase as each table is redistributed across the expanded array. Once the amount of free space on all segments is adequate to store a copy of the largest table, you can redistribute large and critical tables.

In either case:

- Run only one redistribution process at a time until large or critical tables have been successfully redistributed.
- Run multiple parallel redistribution processes in off-hours to maximize the available system resources.
- When running multiple process, make sure you operate within the connection limits for your Greenplum system. For more information, see “[Limiting Concurrent Connections](#)” on page 75.

Redistributing Append-Only and Compressed Tables

Append-only and compressed append-only tables are redistributed by `gpexpand` at different rates from heap tables. The CPU capacity required to compress and decompress data tends to increase the impact on system performance. For similar-sized tables with similar data, you may find overall performances differences like the following:

- Uncompressed append-only tables expand 10% faster than heap tables
- Zlib-compressed append-only tables expand at a significantly slower rate than uncompressed append-only tables, potentially up to 80% slower.

Redistributing Tables with Primary Key Constraints

Between the initialization of new segments the successful redistribution of tables, there is a window during which primary key constraints cannot be enforced. Any duplicate data inserted into tables during this window will prevent the expansion utility from redistributing the affected tables. Once a table is successfully redistributed, the primary key constraint is again properly enforced.

If constraints are violated during the expansion process, the expansion utility logs errors and prints warnings to the screen after attempting to redistribute all tables. You have the following options to remedy constraint violations:

- Clean up duplicate data in the primary key columns, and re-run `gpexpand`.

- Drop the primary key constraints, and re-run `gpexpand`.

Redistributing Tables with User-Defined Data Types

With tables that have dropped columns of user-defined data types, you cannot perform redistribution with the expansion utility. To redistribute tables with dropped columns of user-defined types, first re-create the table using `CREATE TABLE AS SELECT`. Once the dropped columns are removed by this process, you can proceed to redistribute the table with the expansion utility.

Redistributing Partitioned Tables

Because the expansion utility can process a large table partition by partition, an efficient partition design reduces the performance impact of table redistribution. Only the child tables of a partitioned table are set to a random distribution policy, and the read/write lock for redistribution applies to only one child table at a time.

Preparing and Adding Nodes

To prepare new system nodes for expansion, exchange the required SSH keys and run performance tests. Greenplum recommends running performance tests at least twice: first on the new nodes only, and then on both the new and existing nodes together. The second set of tests must be run with the system offline in order to prevent user activity from distorting test results.

Beyond these general guidelines, Greenplum recommends running performance tests any time that the networking of nodes is modified, or for any special conditions in the system environment. For example, if you plan to run the expanded system on two network clusters, run the performance tests on each cluster.

As part of your expansion planning, review [“About Greenplum and Hardware Platforms”](#) on page 30. This chapter describes the requirements and optimal settings for the hardware on which Greenplum Database runs.

This rest of this section describes how to run Greenplum administrative utilities to verify that your new nodes are ready for integration into the existing Greenplum system.

Adding New Nodes to the Trusted Host Environment

New nodes must exchange SSH keys with the existing nodes in order to enable Greenplum administrative utilities to connect to all segments without a password prompt.

Greenplum recommends performing the key exchange process twice: once as `root` (for administration convenience) and once as the `gpadmin` user (required for the Greenplum management utilities). Perform the following tasks in this order:

- [“To exchange SSH keys as root”](#) on page 212
- [“To create the gpadmin user”](#) on page 212
- [“To exchange SSH keys as the gpadmin user”](#) on page 213

To exchange SSH keys as root

1. Create two separate host list files: one that has all of the existing host names in your Greenplum Database array, and one that has all of the new expansion hosts. For existing hosts, you can use the same host file that you used for the initial setup of SSH keys in the system.

The files should include all hosts (master, backup master and segment hosts) and list one host name per line. If using a multi-NIC configuration, make sure to exchange SSH keys using all of the configured host names for a given host. Make sure there are no blank lines or extra spaces. For example:

```

mdw                OR      masterhost
sdw1-1             seghost1
sdw1-2             seghost2
sdw1-3             seghost3
sdw1-4
sdw2-1
sdw2-2
sdw2-3
sdw2-4
sdw3-1
sdw3-2
sdw3-3
sdw3-4

```

2. Log in as `root` on the master host, and source the `greenplum_path.sh` file from your Greenplum installation.

```

$ su -
# source /usr/local/greenplum-db/greenplum_path.sh

```

3. Run the `gpssh-exkeys` utility referencing the host list files. For example:

```

# gpssh-exkeys -e /home/gpadmin/existing_hosts_file -x
/home/gpadmin/new_hosts_file

```

4. `gpssh-exkeys` will check the remote hosts and perform the key exchange between all hosts. Enter the `root` user password when prompted. For example:

```

***Enter password for root@hostname: <root_password>

```

To create the `gpadmin` user

1. Use `gpssh` to create the `gpadmin` user on all of the new segment hosts (if it does not exist already). Use the list of new hosts that you created for the key exchange. For example:

```

# gpssh -f new_hosts_file '/usr/sbin/useradd gpadmin -d
/home/gpadmin -s /bin/bash'

```

2. Set the new `gpadmin` user's password. On Linux, you can do this on all segment hosts at once using `gpssh`. For example:

```
# gpssh -f new_hosts_file 'echo gpadmin_password | passwd
gpadmin --stdin'
```

On Solaris, you must log in to each segment host and set the gpadmin user's password on each host. For example:

```
# ssh segment_hostname
# passwd gpadmin
# New password: <gpadmin_password>
# Retype new password: <gpadmin_password>
```

3. Verify that the gpadmin user has been created by looking for its home directory:

```
# gpssh -f new_hosts_file ls -l /home
```

To exchange SSH keys as the gpadmin user

1. Log in as gpadmin, and run the `gpssh-exkeys` utility referencing the host list files. For example:

```
# gpssh-exkeys -e /home/gpadmin/existing_hosts_file -x
/home/gpadmin/new_hosts_file
```

2. `gpssh-exkeys` will check the remote hosts and perform the key exchange between all hosts. Enter the gpadmin user password when prompted. For example:

```
***Enter password for gpadmin@hostname: <gpadmin_password>
```

Verifying OS Settings

Use the `gpcheckos` utility to verify that all the new hosts in your array have the correct OS settings for running the Greenplum Database software.

To run gpcheckos

1. Log in on the master host as the user who will be running your Greenplum Database system (for example, gpadmin).

2. Run the `gpcheckos` utility using your host file for new hosts. For example:

```
$ gpcheckos -f new_hosts_file
```

3. Look for lines prefixed with `[FIX username@hostname]`. These lines will explain OS-level fixes that need to be made before you initialize Greenplum Database.

4. Run the `gpcheckos` utility again, this time only checking the master host. For example:

```
$ gpcheckos -h mdw-1
```

Validating Disk I/O and Memory Bandwidth

Use the `gpcheckperf` utility to test disk I/O and memory bandwidth.

To run gpcheckperf

1. Run the `gpcheckperf` utility using the host file for new hosts. Use the `-d` option to specify the file systems you want to test on each host (you must have write access to these directories). For example:

```
$ gpcheckperf -f new_hosts_file -d /data1 -d /data2 -v
```

2. The utility may take a while to perform the tests as it is copying very large files between the hosts. When it is finished, you will see the summary results for the Disk Write, Disk Read, and Stream tests.

If your network is divided into subnets, repeat this procedure with a separate host file for each subnet.

Integrating New Hardware into the System

Before initializing the system with all new segments, repeat the performance tests on all nodes in the system, new and existing. Shut down the system and run these same tests using host files that include *all* nodes, existing and new:

- [Verifying OS Settings](#)
- [Validating Disk I/O and Memory Bandwidth](#)

Because user activity may skew the results of these test, you must shut down Greenplum Database (`gpstop`) before running them.

Initializing New Segments

Use the `gpexpand` utility to initialize the new segments, create the expansion schema, and set a system-wide random distribution policy for the database. The utility performs these tasks by default the first time you run it with a valid input file on a Greenplum Database master. Subsequently, it will detect that an expansion schema has been created, and will perform table redistribution.

Creating an Input File for System Expansion

To begin expansion, the `gpexpand` utility requires an input file containing information about the new segments and hosts. If you run `gpexpand` without specifying an input file, the utility displays an interactive interview that collects the required information and automatically creates an input file for you.

If you choose to create the input file using the interactive interview, you can optionally specify a file containing a list of expansion hosts. If your platform or command shell limits the length of the list of hostnames that you can type when prompted in the interview, specifying the hosts with `-f` may be mandatory.

Creating an input file in Interactive Mode

Before running `gpexpand` to create an input file in interactive mode, make sure you have the required information:

- Number of new hosts (or a hosts file)

- New hostnames (or a hosts file)
- The mirroring strategy used in existing hosts, if any
- Number of segments to add per host, if any

The utility automatically generates an input file based on this information and *dbid*, *content* ID, and data directory values stored in *gp_configuration*, and saves it in the current directory.

To create an input file in interactive mode

1. Log in on the master host as the user who will be running your Greenplum Database system (for example, *gpadmin*).
2. Run `gpexpand`. The utility displays messages about preparing for an expansion operation and prompts you to quit or continue.
Optionally, specify a hosts file using `-f`. For example:

```
$ gpexpand -f /home/gpadmin/new_hosts_file
```

3. At the prompt, select `Y` to continue.
4. Unless you specified a hosts file using `-f`, you will be prompted to enter hostnames. Enter a comma separated list of the hostnames of the new expansion hosts. Do not include interface hostnames. For example:

```
> sdw4-1, sdw4-2, sdw4-3, sdw4-4
```

To add segments to existing hosts only, enter a blank line at this prompt. Do not specify `localhost` or any existing host name.

5. Enter the mirroring strategy used in your system, if any. Options are `spread|grouped|none`, default is `grouped`.
Make sure you have enough hosts for your selected grouping strategy. For more information, see “[Planning Mirror Segments](#)” on page 208.
6. Enter the number of new primary segments to add, if any. By default, new hosts are initialized with the same number of primary segments as existing hosts. Optionally, you can increase the number of segments per host.
If you want to increase the number of segments per host, enter a number greater than zero. This number of additional segments will be initialized on all hosts. For example, if existing hosts currently have two segments per host, entering a value of 2 will initialize two additional segments on existing hosts, and four new segments on new hosts.
7. If you are adding new primary segments, enter the new primary data directory root for the new segments. Do not specify the actual data directory name, which is created automatically by `gpexpand` based on the existing data directory names in *gp_configuration*.

For example, if your existing data directories are as follows:

```
/usr/local/gp/data/primary/gp0
/usr/local/gp/data/primary/gp1
```

you should enter the following (one at each prompt) to specify the data directories for two new primary segments:

```
/usr/local/gp/data/primary
/usr/local/gp/data/primary
```

When the initialization is run, the utility will create the new directories `gp2` and `gp3` under `/usr/local/gp/data/primary`

8. If you are adding new mirror segments, enter the new mirror data directory root for the new segments. Do not specify the actual data directory name, which is created automatically by `gpexpand` based on the existing data directory names in `gp_configuration`.

For example, if your existing data directories are as follows:

```
/usr/local/gp/data/mirror/gp0
/usr/local/gp/data/mirror/gp1
```

you should enter the following (one at each prompt) to specify the data directories for two new mirror segments:

```
/usr/local/gp/data/mirror
/usr/local/gp/data/mirror
```

When the initialization is run, the utility will create the new directories `gp2` and `gp3` under `/usr/local/gp/data/mirror`



Important: These primary and mirror root directories for new segments must exist on the hosts, and the user running `gpexpand` must have permissions to create directories in them.

After you have entered all required information, the utility generates an input file and saves it in the current directory. For example:

```
gpexpand_inputfile_yyyymmdd_145134
```

Expansion Input File Format

You can create your own input file in the required format. Unless you have special needs for your expansion scenario, Greenplum recommends creating the input file using the interactive interview process.

The format for expansion input files is:

```
hostname:port:datadir:dbid:content:definedprimary
```

For example:

```
myserver:50011:/usr/local/gp/data/gp9:11:9:t
myserver:50012:/usr/local/gp/data/gp10:12:10:t
myserver:60011:/usr/local/gp/data/gp9:13:9:f
myserver:60012:/usr/local/gp/data/gp10:14:10:f
```

An expansion input file in this format requires the following information for each new segment:

Table 22.1 Data for the expansion configuration file

Parameter	Valid Values	Description
hostname	Hostname	Hostname for the segment host.
port	An available port number	Port for the segment, incremented on the existing segment port base number.
datadir	Directory name	Data directory to create on the segment.
dbid	Integer. Must not conflict with existing <i>dbid</i> values.	Database ID for the segment. The values you enter should be incremented sequentially from existing <i>dbid</i> values shown in the system catalog gp_configuration . For example, to add four nodes to an existing ten-segment array with <i>dbid</i> values of 1-10, list new <i>dbid</i> values of 11, 12, 13 and 14.
content	Integer. Must not conflict with existing <i>content</i> values.	The content ID of the segment. A primary segment and its mirror should have the same content ID, incremented sequentially from existing values. For more information, see <i>content</i> in the reference for gp_configuration .
definedprimary	t f	Boolean to determine whether this segment is a primary or mirror. Specify <i>f</i> for primary, <i>t</i> for mirror.

Running gpexpand to Initialize New Segments

After you have created an input file, run [gpexpand](#) to initialize new segments. The utility will automatically stop Greenplum Database for the time required to initialize the segments, and then restart the system when finished.

To run gpexpand with an input file

1. Log in on the master host as the user who will be running your Greenplum Database system (for example, `gpadmin`).
2. Run the [gpexpand](#) utility, specifying the input file with `-i`. Optionally, use `-D` to specify the database in which to create the expansion schema. For example:

```
$ gpexpand -i input_file -D database1
```

The utility detects if there is an existing expansion schema for the Greenplum system. If there is an existing schema, you must remove it with `gpexpand -c` before beginning a new expansion operation. See “[Removing the Expansion Schema](#)” on page 220.

When the new segments are initialized and the expansion schema is successfully created, the utility prints a success message and exits.

When the initialization process is complete, you can connect to Greenplum Database and view the expansion schema. The schema resides in the database you specified with `-D`, or in the database specified by the `PGDATABASE` environment variable. For more information, see “[About the Expansion Schema](#)” on page 209.

Rolling Back an Expansion Setup

You can roll back an expansion setup operation using the command `gpexpand -r | --rollback`. However, this command is only allowed before you begin the redistribution of tables. Once you have begun redistribution, the expansion is committed, and you cannot roll back.

To roll back an expansion setup, use the following command, specifying the database that contains the expansion schema:

```
gpexpand --rollback -D database_name
```

Redistributing Tables

After successfully creating an expansion schema, you can bring Greenplum Database back online and redistribute tables across the entire array. You can redistribute tables with `gpexpand` at specified intervals, targeting low-use hours when the utility’s CPU usage and table locks will have the least impact on database operations. Also, you can rank tables to ensure that the largest or most critical tables are redistributed in your preferred order.

While the redistribution of tables is underway:

- Any new tables or partitions created will be distributed across all segments.
- Queries will use all segments, even though the relevant data may not have yet been redistributed to the tables on the new segments.
- The table or partition currently being redistributed will be unavailable for read or write operations. When its redistribution is completed, normal operations resume.

Ranking Tables for Redistribution

For large systems, you may prefer to control the order in which tables are redistributed by adjusting their *rank* values in the expansion schema. This allows you to redistribute heavily-used tables first and minimize the performance hit on the system.

To rank tables for redistribution by updating *rank* values in `gpexpand.status_detail`, connect to Greenplum Database using `psql` or another supported client. Update `gpexpand.status_detail` with commands like the following:

```
=> UPDATE gpexpand.status_detail SET rank= 10;
UPDATE gpexpand.status_detail SET rank=1 WHERE fq_name =
'public.lineitem';
UPDATE gpexpand.status_detail SET rank=2 WHERE fq_name =
'public.orders';
```

These commands first lower the priority of all tables to 10, and then assign a rank of 1 to *lineitem* and then a rank of 2 to *orders*. When table redistribution begins, *lineitem* will be redistributed first, followed by *orders* and then all other tables in *gpexpand.status_detail*.



Note: For any table that you do not want to redistribute, you must remove the corresponding entry from *gpexpand.status_detail*.

Redistributing Tables Using gpexpand

To redistribute tables with gpexpand

1. Log in on the master host as the user who will be running your Greenplum Database system (for example, *gpadmin*).
2. Run the `gpexpand` utility. Optionally, you can use either the `-d` or `-e` option to define the time period for the expansion session. For example, to run the utility for a maximum of 60 consecutive hours:

```
$ gpexpand -d 60:00:00
```

The utility redistributes tables until the last table in the schema is successfully marked completed, or until the specified duration or end time is reached. Each time a session is started or finished, the utility updates the status and updated time in *gpexpand.status*.

Monitoring Table Redistribution

At any time during the process of redistributing tables, you can query the expansion schema. The view *gpexpand.expansion_progress* provides a summary of the current progress, including calculations of the estimated rate of table redistribution and estimated time to completion. The table *gpexpand.status_detail* can be queried for per-table status information.

Viewing Expansion Status

Because the estimates in *gpexpand.expansion_progress* are based on the rates achieved for each table, the view cannot calculate an accurate estimate until the first table has completed. Calculations are restarted each time you re-run `gpexpand` to start a new table redistribution session.

To monitor progress by querying *gpexpand.expansion_progress*, connect to Greenplum Database using `psql` or another supported client. Query *gpexpand.expansion_progress* with a command like the following:

```
=# select * from gpexpand.expansion_progress;
      name      | value
-----+-----
Bytes Left     | 5534842880
Bytes Done     | 142475264
Estimated Expansion Rate | 680.75667095996092 MB/s
```

```

Estimated Time to Completion | 00:01:01.008047
Tables Expanded              | 4
Tables Left                  | 4
(6 rows)

```

Viewing Table Status

The table `gpexpand.status_detail` stores status, last updated time, and other useful information about each table in the schema. To monitor the status of a particular table by querying `gpexpand.status_detail`, connect to Greenplum Database using `psql` or another supported client. Query `gpexpand.status_detail` with a command like the following:

```

=> SELECT status, expansion_started, source_bytes FROM
gpexpand.status_detail WHERE fq_name = 'public.sales';
status      |      expansion_started      | source_bytes
-----+-----+-----
COMPLETED | 2009-02-20 10:54:10.043869 | 4929748992
(1 row)

```

Removing the Expansion Schema

The expansion schema can safely be removed after the expansion operation is completed and verified. To run another expansion operation on a Greenplum system, you must first remove the existing expansion schema.

To remove the expansion schema

1. Log in on the master host as the user who will be running your Greenplum Database system (for example, `gpadmin`).
2. Run the `gpexpand` utility with the `-c` option. For example:

```
$ gpexpand -c
```

23. Monitoring a Greenplum System

As the Greenplum administrator, it is often necessary to know what is going on in the Greenplum Database system in order to plan workflow and troubleshoot problems. This chapter discusses the various tools available for monitoring database performance and activity.

- [Monitoring Database Activity and Performance](#)
- [Monitoring System State](#)
- [Viewing the Database Server Log Files](#)
- [Using the Jetpack Administrative Interface](#)

Also, once one has identified a poorly-performing query, further investigation may be needed using the `EXPLAIN` command. See “[Query Profiling](#)” on page 148 for more information.

Monitoring Database Activity and Performance

Greenplum provides an optional performance monitoring feature that administrators can install and enable with Greenplum Database 3.3.7. The Greenplum Performance Monitor is shipped separately from the Greenplum Database 3.3.7 installation. You can download the Greenplum Performance Monitor package and documentation from [Greenplum Network](#). See the *Greenplum Database Performance Monitor Guide* for more information.

Monitoring System State

In the current release of Greenplum Database, monitoring the system state is a proactive process. The administrator must actively check the system status to see if there are any problems (such as a segment being down or running out of disk space). In some cases, client error messages may provide some clue that there is a problem. See “[Knowing When a Segment is Down](#)” on page 189.

Checking System Status and Configuration

A Greenplum Database system is comprised of multiple PostgreSQL instances (the master and segments) spanning multiple machines. To monitor a Greenplum Database system, you need to know information about the system as a whole, as well as status information of the individual instances. The `gpstate` utility provides status information about a Greenplum Database system.

You can also use the `SHOW SQL` command to view the current server configuration settings.

Viewing Master and Segment Status and Configuration

The default behavior of `gpstate` is to check the segment instances and show a brief status of the valid and failed segments. For example, to see a quick status of the state of your Greenplum Database system:

```
$ gpstate
```

To see more detailed information about your Greenplum array configuration, use `gpstate` with the `-s` option:

```
$ gpstate -s
```

Viewing Your Mirroring Configuration

If you are using mirroring for data redundancy, you may want to see the list of mirror segment instances in the system, and the mirror to primary mapping. For example, to see the mirror segments in the system and their status:

```
$ gpstate -m
```

To see the primary to mirror segment mappings:

```
$ gpstate -c
```

To see the status of the standby master mirror:

```
$ gpstate -f
```

Viewing Settings of Server Configuration Parameters

The `SHOW` SQL command allows you to see the settings of the server configuration parameters used by the Greenplum Database system. For example, to see the settings for all parameters:

```
$ psql -c 'SHOW ALL;'
```

Running `SHOW` will show the settings for the master instance only. However, in most cases the settings on the master will be the same as on the segments or will take precedence over the segment settings. For exceptions, see “[Server Configuration Parameters](#)” on page 755.

Checking Disk Space Usage

The most important monitoring task of a database administrator is to make sure that the file systems where the master and segment data directories reside do not grow to more than 70 percent full. A filled data disk will not result in data corruption, but it may prevent normal database activity from occurring. If the disk grows too full, database server panic and consequent shutdown may occur.

You can use the `gpssh` utility to run the `df` (disk free) system command on all Greenplum hosts at once. This will show you the free disk space on the file systems in 1K blocks. For example (where `all_hosts_file` is a file containing the name of all your Greenplum hosts):

```
$ gpssh -f all_hosts_file df -k
```

Checking Sizing of Distributed Databases and Tables

The `gpsizecalc` utility can be used to determine the disk space usage for a distributed Greenplum database, table, or index. It calculates the total size of an object across all segments.

Viewing Disk Space Usage for a Database

To see the total size of a database, use `gpsizecalc` with the `-x` option. The `-s m` option shows the size in megabytes (rather than kilobytes, which is the default). For example:

```
$ gpsizecalc -x mydatabase -s m
```

If you want to see the database size on a per-segment basis, add the `-f` option. This will show sizing information for all active segments. For example:

```
$ gpsizecalc -x mydatabase -s m -f
```

Viewing Disk Space Usage for a Table

To see the total size of a table, use `-s m` with the `-t` option. The `-s m` option shows the size in megabytes (rather than kilobytes, which is the default). For example:

```
$ gpsizecalc -t myschema.mytable -s m
```

If you want to see the table size on a per-segment basis, add the `-f` option. This will show sizing information for all active segments. For example:

```
$ gpsizecalc -t myschema.mytable -s m -f
```

Viewing Disk Space Usage for Indexes

To see the total size of the index(es) on a table, use `gpsizecalc` with the `-t` option to specify the table and the `-i` option to show size information for the index(es) on that table. For example:

```
$ gpsizecalc -t myschema.mytable -s m -i
```

Checking for Data Distribution Skew

All tables in Greenplum Database are distributed, meaning their data is divided across all of the segments in the system. If the data is not distributed evenly, then query processing performance may suffer. A table's distribution policy is determined at table creation time. See [“Choosing the Table Distribution Policy”](#) on page 100 for more information.

The `gpskew` utility can be used to determine if table data is equally distributed across all of the active segments. `gpskew` reports the following information:

- The total number of records in the specified table.
- The number of records on each segment.
- The variance of records between segments.
- The segments with the fastest and slowest response times.

Viewing a Table's Distribution Key

To see the columns used as the data distribution key for a table, use `gpskew` with the `-t` option to specify the table and the `-c` option to show the distribution columns. For example:

```
$ gpskew -t myschema.mytable -c
```

Viewing Data Distribution and Segment Response Times

To see the data distribution and segment response times for a table, use `gpskew` with the `-t` option to specify the table and the `-f` option to provide full output including segment response times. For example:

```
$ gpskew -t myschema.mytable -f
```

Checking for Query Processing Skew

When a query is being processed, you want all of the segments to handle an equal amount of the query workload to get the best possible performance. In some cases, query processing workload can be skewed if the table's data distribution policy and the query predicates are not well matched. To check for processing skew, use `gpskew` with the `-t` option to specify the table and the `-w` option to test a particular query predicate (`WHERE` condition). For example:

```
$ gpskew -t schema.table -w "date between '2007-01-01' and
'2007-01-12'"
```

Viewing the Database Server Log Files

Every database instance in Greenplum Database (master and segments) is running a PostgreSQL database server with its own server log file. Daily log files are created in the `pg_log` directory of the master and each segment data directory.

Log File Format

The server log files are written in comma-separated values (CSV) format. Not all log entries will have values for all of the log fields. For example, only log entries associated with a query worker process will have the `slice_id` populated. Related log entries of a particular query can be identified by its session identifier (`gp_session_id`) and command identifier (`gp_command_count`).

The following fields are written to the log:

Table 23.1 Greenplum Database Server Log Format

#	Field Name	Data Type	Description
1	event_time	timestamp with time zone	Time that the log entry was written to the log
2	user_name	varchar(100)	The database user name
3	database_name	varchar(100)	The database name
4	process_id	varchar(10)	The system process id (prefixed with "p")
5	thread_id	varchar(50)	The thread count (prefixed with "th")

Table 23.1 Greenplum Database Server Log Format

#	Field Name	Data Type	Description
6	remote_host	varchar(100)	The segment or master host name
7	remote_port	varchar(10)	The segment or master port number
8	session_start_time	timestamp with time zone	Time session connection was opened
9	transaction_id	int	Global transaction ID
10	gp_session_id	text	Session identifier number (prefixed with "con")
11	gp_command_count	text	The command number within a session (prefixed with "cmd")
12	gp_segment	text	The segment content identifier (prefixed with "seg" for primaries or "mir" for mirrors). The master always has a content id of -1.
13	slice_id	text	The slice id (portion of the query plan being executed)
14	distr_tranx_id	text	Distributed transaction ID
15	local_tranx_id	text	Local transaction ID
16	sub_tranx_id	text	Subtransaction ID
17	event_severity	varchar(10)	Values include: LOG, ERROR, FATAL, PANIC, DEBUG1, DEBUG2
18	sql_state_code	varchar(10)	SQL state code associated with the log message
19	event_message	text	Log or error message text
20	event_detail	text	Detail message text associated with an error or warning message
21	event_hint	text	Hint message text associated with an error or warning message
22	internal_query	text	The internally-generated query text
23	internal_query_pos	int	The cursor index into the internally-generated query text
24	event_context	text	The context in which this message gets generated
25	debug_query_string	text	Query string with full detail for debugging
26	error_cursor_pos	int	The cursor index into the query string
27	func_name	text	The function in which this message is generated
28	file_name	text	The internal code file where the message originated
29	file_line	int	The line of the code file where the message originated
30	stack_trace	text	Stack trace text associated with this message

Searching the Greenplum Database Server Log Files

Greenplum provides a utility called `gplogfilter` that can be used to search through a Greenplum Database log file for entries matching the specified criteria. By default, this utility searches through the Greenplum master log file in the default logging location. For example, to display the last three lines of the master log file:

```
$ gplogfilter -n 3
```

You can also use `gplogfilter` to search through all segment log files at once by running it through the `gpssh` utility. For example, to display the last three lines of each segment log file:

```
$ gpssh -f seg_host_file
=> source /usr/local/greenplum-db/greenplum_path.sh
=> gplogfilter -n 3 /gpdata/gp*/pg_log/gpdb*.log
```

Using the Jetpack Administrative Interface

Greenplum provides an optional administrative schema called *Jetpack* that can be used to view and query the system log files and other system metrics. The Jetpack schema contains a number of external tables, views and functions that an administrator can access using SQL commands.

You must be a database superuser to install Jetpack, however once it is installed, public access is granted to other database users. Refer to the `JETPACK_README` file in `$GPHOME/lib/jetpack` for a description of the administrative views and functions provided.

To install the Jetpack schema

1. Execute the `jetpack.sql` file as a database superuser, where `gpadmin` is the superuser name and `mydatabase#` are the databases in which you want to install the Jetpack schema and objects. If your Greenplum system has multiple databases, you must install the Jetpack schema into each database. For example:

```
psql -f $GPHOME/lib/jetpack/jetpack.sql -U gpadmin -d
mydatabase1
```

```
psql -f $GPHOME/lib/jetpack/jetpack.sql -U gpadmin -d
mydatabase2
```

2. For convenience, add the `gp_jetpack` schema to your schema search path. For example (where `myschema1` and `myschema2` are names of your user-created schemas):

```
psql -c "ALTER DATABASE mydatabase1 SET search_path TO
myschema1,myschema2,gp_jetpack,pg_catalog" -U gpadmin -d
templatel
```

24. Routine System Maintenance Tasks

Greenplum Database, like any database software, requires that certain tasks be performed regularly to achieve optimum performance. The tasks discussed here are required, but they are repetitive in nature and can easily be automated using standard Unix tools such as `cron` scripts. But it is the database administrator's responsibility to set up appropriate scripts, and to check that they execute successfully.

- [Routine Vacuum and Analyze](#)
- [Routine Reindexing](#)
- [Managing Greenplum Database Log Files](#)

Routine Vacuum and Analyze

Because of the MVCC transaction concurrency model used in Greenplum Database, data rows that are deleted or updated still occupy physical space on disk even though they are not visible to any new transactions. If you have a database with lots of updates and deletes, you will generate a lot of expired rows. The `VACUUM` command also collects table-level statistics such as number of rows and pages, so it is also necessary to vacuum append-only tables. Vacuuming append-only tables should be instantaneous since there will be no space to reclaim. See [“Vacuuming the Database”](#) on page 130.

Transaction ID Management

Greenplum's MVCC transaction semantics depend on being able to compare transaction ID (XID) numbers to determine visibility to other transactions. But since transaction IDs have limited size, a Greenplum system that runs for a long time (more than 4 billion transactions) would suffer transaction ID wraparound: the XID counter wraps around to zero, and all of a sudden transactions that were in the past appear to be in the future — which means their outputs become invisible. To avoid this, it is necessary to `VACUUM` every table in every database at least once every two billion transactions.

System Catalog Maintenance

Numerous database updates with `CREATE` and `DROP` commands can cause growth in the size of the system catalog that affects system performance. For example, after a large number of `DROP TABLE` statements, the overall performance of the system begins to degrade due to excessive data scanning during metadata operations on the catalog tables. Depending on your system, the performance loss may occur between thousands to tens of thousands of `DROP TABLE` statements.

Greenplum recommends that you regularly run a system catalog maintenance procedure to reclaim the space occupied by deleted objects. If a regular procedure has not been run for a long time, you may need to run a more intensive procedure to clear the system catalog. Both types of procedures are described in this section.

Regular System Catalog Maintenance

Greenplum recommends that you periodically run `VACUUM` on the system catalog to clear the space occupied by deleted objects. If numerous `DROP` statements are a part of regular database operations, it is safe and appropriate to run a system catalog maintenance procedure with `VACUUM` daily at off-peak hours. This can be done while the system is running and available.

The following example script performs a `VACUUM` of the Greenplum Database system catalog:

```
#!/bin/bash
DBNAME="<database_name>"
VCOMMAND="VACUUM ANALYZE"
psql -tc "select '$VCOMMAND' || ' pg_catalog.' || relname || ';'
from pg_class a,pg_namespace b where a.relnamespace=b.oid and
b.nspname= 'pg_catalog' and a.relkind='r'" $DBNAME | psql -a
$DBNAME
```

Intensive System Catalog Maintenance

If a system catalog maintenance procedure has not been performed in a long time, the catalog may become bloated with dead space, causing excessively long wait times for simple metadata operations. A wait of more than one or two seconds to list user tables, such as with the `\d` metacommand from within `psql`, is an indication of catalog bloat.

If you see indications of system catalog bloat, an intensive system catalog maintenance procedure with `VACUUM FULL` must be performed during a scheduled downtime period. During this period you must stop all catalog activity on the system due to the exclusive locks taken against the system catalog by the `FULL` system catalog maintenance procedure.

Running regular system catalog maintenance procedures can prevent the need for the more costly intensive procedure.

Vacuum and Analyze for Query Optimization

Greenplum Database uses a cost-based query planner that relies on database statistics. Accurate statistics allow the query planner to better estimate selectivity and the number of rows retrieved by a query operation in order to choose the most efficient query plan. The `ANALYZE` command collects column-level statistics needed by the query planner.

Both `VACUUM` and `ANALYZE` operations can be run in the same command. For example:

```
=# VACUUM ANALYZE mytable;
```

Routine Reindexing

For B-tree indexes, a freshly-constructed index is somewhat faster to access than one that has been updated many times, because logically adjacent pages are usually also physically adjacent in a newly built index. It might be worthwhile to reindex periodically to improve access speed. Also, if all but a few index keys on a page have

been deleted, there will be wasted space on the index page. A reindex will reclaim that wasted space. In Greenplum Database it is often faster to drop an index (`DROP INDEX`) and then recreate it (`CREATE INDEX`) than it is to use the `REINDEX` command.

Bitmap indexes are not updated when changes are made to the indexed column(s). If you have updated a table that has a bitmap index, you must drop and recreate the index for it to remain current.

Managing Greenplum Database Log Files

- [Database Server Log Files](#)
- [Management Utility Log Files](#)

Database Server Log Files

Greenplum Database log output tends to be voluminous (especially at higher debug levels) and you do not need to save it indefinitely. Administrators need to rotate the log files periodically so that new log files are started and old ones are removed after a reasonable period of time.

Greenplum Database has log file rotation enabled on the master and all segment instances. Daily log files are created in `pg_log` of the master and each segment data directory using the naming convention of: `gpdb-YYYY-MM-DD.log`. Although log files are rolled over daily, they are not automatically truncated or deleted. Administrators will need to implement some script or program to periodically clean up old log files in the `pg_log` directory of the master and each segment instance.

See also, “[Viewing the Database Server Log Files](#)” on page 224.

Management Utility Log Files

Log files for the Greenplum Database management utilities are written to `~/gpAdminLogs` by default. The naming convention for management log files is:

```
<script_name>_<date>.log
```

The log entry format is:

```
<timestamp>:<utility>:<host>:<user>: [INFO|WARN|FATAL]:<message>
```

The log file for a particular utility execution is appended to its daily log file each time that utility is run.

Section VI: Performance Tuning

This section describes the different performance tuning opportunities of Greenplum Database and how to troubleshoot certain performance problems. It contains the following topics:

- [Defining Database Performance](#)
- [Common Causes of Performance Issues](#)
- [Investigating a Performance Problem](#)

25. Defining Database Performance

For a data warehouse database system, such as Greenplum Database, database performance can be defined as the rate at which the database management system (DBMS) supplies information to those requesting it.

- [Understanding the Performance Factors](#)
- [Determining Acceptable Performance](#)

Understanding the Performance Factors

There are several factors that influence database performance. Understanding the key performance factors can help avoid performance problems or identify performance opportunities:

- [System Resources](#)
- [Workload](#)
- [Throughput](#)
- [Contention](#)
- [Optimization](#)

System Resources

Database performance relies heavily on disk I/O and memory usage. Knowing the baseline performance of the hardware on which your DBMS is deployed is essential in setting performance expectations. Performance of hardware components such as CPUs, hard disks, disk controllers, RAM, and network interfaces (and the interaction of these resources) will have a profound effect on how fast your database performs.

Workload

Your workload equals the total demand from the DBMS. The total workload is a combination of ad-hoc user queries, applications, batch jobs, transactions, and system commands directed through the DBMS at any given time. Workload, or demand, can change over time. For example, it may increase when month-end reports need to be run, or decrease on weekends when most users are out of the office. Workload is a major influence on database performance. Knowing your workload and peak demand times will help you plan for the most efficient use of your system resources, and enable the largest possible workload to be processed.

Throughput

A system's throughput defines its overall capability to process data. DBMS throughput can be measured in queries per second, transactions per second, or average response times. DBMS throughput is closely related to the processing capacity of the

underlying systems (disk I/O, CPU speed, memory bandwidth, and so on), so it is important to know the throughput capacity of your hardware when setting DBMS throughput goals.

Contention

Contention is the condition in which two or more components of the workload are attempting to use the system in a conflicting way — for example, trying to update the same piece of data at the same time, or running multiple large workloads at once that compete with each other for system resources. As contention increases, throughput decreases.

Optimization

Optimizations that you make to your DBMS can affect the overall performance of your system. SQL formulation, database configuration parameters, table design, data distribution, and so on can enable the database query planner and optimizer to create the most efficient access plans.

Determining Acceptable Performance

When approaching a performance tuning initiative, it is important to know your system's expected level of performance and to define measurable performance requirements. Without setting an acceptable threshold for database performance, you will end up chasing a carrot always out of reach. Consider the following when setting performance goals:

- [Baseline Hardware Performance](#)
- [Performance Benchmarks](#)

Baseline Hardware Performance

The majority of database performance problems are caused not by the database itself, but by the underlying systems on which the database is running. I/O bottlenecks, memory problems, and network problems can significantly degrade database performance. Therefore, it is important to know the baseline capabilities of your hardware and operating system (OS). This will help you to identify and troubleshoot hardware-related problems before undertaking database-level or query-level tuning initiatives. See “[Validating Hardware Performance](#)” on page 37 for more information.

Performance Benchmarks

In order to maintain good performance or improve upon performance issues, you need to know the capabilities of your DBMS on a defined workload. A benchmark is a predefined workload that produces a known result set, which can then be used for comparison purposes. Periodically running the same benchmark tests can help identify system-related performance degradation over time. Benchmarks can also be used as a comparison to other workloads in an effort to identify queries or applications in need of optimization.

There are many third-party organizations which provide benchmark tools for the database industry, one of those being the Transaction Processing Performance Council (TPC). TPC-H is the ad-hoc, decision support benchmark. This benchmark illustrates decision support systems that examine large volumes of data, execute queries with a high degree of complexity, and give answers to critical business questions. For more information about TPC-H, go to:

<http://www.tpc.org/tpch>

26. Common Causes of Performance Issues

This chapter describes the common causes of performance issues and potential solutions to these issues. The following issues are the most common cause of performance problems in Greenplum Database:

- Hardware failures and invalid segments
- Multiple workloads competing for system resources
- Contention between concurrent transactions
- Inaccurate database statistics
- Unbalanced data distribution across the segments
- Unoptimized database design

Identifying Hardware and Segment Failures

As with any database system, the performance of Greenplum Database is dependant upon the hardware and IT infrastructure on which it is running. Greenplum Database is comprised several servers (or hosts) acting together as one cohesive system. Greenplum Database's performance will be as fast as the slowest host in the array. Problems with CPU utilization, memory management, I/O processing, or network load will affect performance. Common hardware-related issues are:

- **Disk Failure** – Although a single disk failure should not dramatically effect database performance if you are using RAID, there is some impact caused by disk resynching consuming resources on the host with failed disks. The `gpcheckperf` utility can help identify segment hosts that have disk I/O issues.
- **Host Failure** – When a host is offline the segments on that host are out of operation. This means that other hosts in the array are doing double duty as they are running both the primary segments and a number of mirrors. If mirrors are not enabled – service is interrupted. There is also a temporary interruption of service to recover failed segments. The `gpstate` utility can help identify failed segments.
- **Network Failure** – Failure of a network interface card, a switch, or DNS server can bring down segments. If host names or IP addresses cannot be resolved within your Greenplum array, these manifest themselves as interconnect errors in Greenplum Database. The `gpchecknet` utility can help identify segment hosts that have network issues.
- **Disk Capacity** – Disk capacity on your segment hosts should never exceed 70 percent full. Greenplum needs some free space for runtime processing. You can reclaim disk space occupied by deleted rows by running `VACUUM` after loads or updates. The `gpsizecalc` utility can be used to check sizing of your distributed tables and databases across the entire system.

Managing Workload

A database system has a limited capacity of CPU, memory, and disk I/O resources. When multiple workloads compete for access to these resources, database performance suffers. Workload management can be used to maximize system throughput while still meeting varied business requirements. Greenplum Database workload management limits the number of active queries in the system at any given time in order to avoid exhausting system resources. This is accomplished by creating role-based resource queues. A resource queue has attributes that limit the size and/or total number of queries that can be executed by the users (or roles) in that queue. By assigning all of your database roles to the appropriate resource queue, administrators can control concurrent user queries and prevent the system from being overloaded. See [Chapter 13, “Managing Workload and Resources”](#) for more information about setting up resource queues.

As the Greenplum Database administrator, run maintenance workloads such as data loads and `VACUUM ANALYZE` operations after regular business hours. Do not compete with database users for system resources by performing administrative tasks at peak usage times.

Avoiding Contention

Contention arises when two or more users or workloads try to use the system in a conflicting way. For example, if two transactions are trying to update the same table at once. A transaction seeking either a table-level or row-level lock will wait indefinitely for conflicting locks to be released. This means it is a bad idea for applications to hold transactions open for long periods of time (e.g., while waiting for user input).

Maintaining Database Statistics

Greenplum Database uses a cost-based query planner that relies on database statistics. Accurate statistics allow the query planner to better estimate the number of rows retrieved by a query in order to choose the most efficient query plan. Without database statistics, the query planner can not estimate how many records might be returned, and therefore cannot assume it has sufficient memory to perform certain operations such as aggregations. In this case, the planner always takes the safe route and does aggregations by reading/writing from disk, which is significantly slower than doing them in memory. The `ANALYZE` command collects statistics about the database needed by the query planner.

Identifying Statistics Problems in Query Plans

When looking at the query plan for a query using `EXPLAIN` or `EXPLAIN ANALYZE`, it helps to know your data in order to identify possible statistics problems. Check the plan for the following indicators of inaccurate statistics:

- **Are the planner’s estimates close to reality?** Run an `EXPLAIN ANALYZE` and see if the number of rows estimated by the planner is close to the number of rows actually returned by the query operation.

- **Are selective predicates applied early in the plan?** The most selective filters should be applied early in the plan so that less rows move up the plan tree.
- **Is the planner choosing the best join order?** When you have a query that joins multiple tables, make sure that the planner is choosing the most selective join order. Joins that eliminate the largest number of rows should be done earlier in the plan so that less rows move up the plan tree.

See also, “[Query Profiling](#)” on page 148 for more information on reading query plans.

Tuning Statistics Collection

The following configuration parameters control the amount of data sampled for statistics collection:

- `default_statistics_target`
- `gp_analyze_relative_error`

These parameters control statistics sampling at the system level. It is probably better to only sample increased statistics for the columns used most frequently in query predicates. You can adjust statistics for a particular column using the

`ALTER TABLE...SET STATISTICS` command. For example:

```
ALTER TABLE sales ALTER COLUMN region SET STATISTICS 50;
```

This is equivalent to increasing `default_statistics_target` for a particular column. Subsequent `ANALYZE` operations will then gather more statistics data for that column, and hopefully produce better query plans as a result.

Optimizing Data Distribution

When you create a table in Greenplum Database, it is important to declare a distribution key that allows for even data distribution across all segments in the system. Because the segments work on a query in parallel, Greenplum Database will always be as fast as the slowest segment. If the data is unbalanced, the segments that have more data will return their results slower.

Optimizing Your Database Design

Many performance issues can be improved by database design. Examine your database design and ask yourself the following:

- Does the schema reflect the way the data is accessed?
- Can larger tables be broken down into partitions?
- Are you using the smallest data type possible to store column values?
- Are columns used to join tables of the same datatype?
- Are your indexes being used?

Greenplum Database Maximum Limits

Knowing these maximum limits supported by Greenplum Database can help you optimize database design:

Table 26.1 Maximum Limits of Greenplum Database

Dimension	Limit
Database Size	Unlimited
Table Size	128 TB per partition per segment
Row Size	1 GB
Field Size	1 GB
Rows per Table	2 ⁴⁸ (2 to the power of 48)
Columns per Table/View	1600
Indexes per Table	Unlimited
Columns per Index	32
Table-level Constraints per Table	Unlimited
Table Name Length	64 Bytes (Limited by <i>name</i> data type)

Dimensions listed as unlimited are not intrinsically limited by Greenplum Database. However, they are limited in practice to available disk space and memory/swap space. Performance may suffer when these values get unusually large.



Note: Limits for all database objects are limited by the number of available OIDs. There are 4 GB of OIDs in total, and all objects consume OIDs from a single namespace per database.

27. Investigating a Performance Problem

Many of the performance management steps taken by database administrators are reactive — a business user calls with a response time problem, a batch job fails, the system suddenly becomes unavailable. This section lists some steps an administrator can take to help identify the cause of a performance problem. If the problem is related to a particular workload or query, then you can focus your efforts on tuning that particular workload. If the performance problem is system-wide, then hardware problems, system failures, or resource contention may be the cause.

Checking System State

The `gpstate` utility can be used to identify failed segments. A Greenplum Database system will incur performance degradation when it has segment instances down because it requires some hosts to pick up the processing responsibilities of the downed segments.

Failed segments can be an indicator of some type of hardware failure, such as a failed disk drive or network card. Greenplum Database provides the hardware verification tools `gpcheckperf` and `gpchecknet` to help identify segment hosts that have hardware issues.

Checking Database Activity

- [Checking for Active Sessions \(Workload\)](#)
- [Checking for Locks \(Contention\)](#)
- [Checking Query Status and System Utilization](#)

Checking for Active Sessions (Workload)

The `pg_stat_activity` system catalog view shows one row per server process, showing database OID, database name, process ID, user OID, user name, current query, time at which the current query began execution, time at which the process was started, and client address and port number. Querying this view can provide more information about the current workload on the system. For example:

```
SELECT * FROM pg_stat_activity;
```

This view should be queried as the database superuser to obtain the most information possible. Also note that the information does not update instantaneously.

Checking for Locks (Contention)

If a transaction is holding a lock on an object, there may be other queries that are waiting for that lock to be released before they can continue. This may appear to the user as if their query is hanging. The `pg_locks` system catalog view allows you to view information about outstanding locks. Examining `pg_locks` for ungranted locks can

help identify contention between database client sessions. *pg_locks* provides a global view of all locks in the database system, not only those relevant to the current database. Although its relation column can be joined against *pg_class.oid* to identify locked relations (such as tables), this will only work correctly for relations in the current database. The *pid* column can be joined to the *pg_stat_activity.procpid* to get more information on the session holding or waiting to hold a lock. For example:

```
SELECT locktype, database, c.relname, l.relation,
       l.transactionid, l.transaction, l.pid, l.mode, l.granted,
       a.current_query
   FROM pg_locks l, pg_class c, pg_stat_activity a
  WHERE l.relation=c.oid AND l.pid=a.procpid
  ORDER BY c.relname;
```

If you are using resource queues for workload management, queries that are waiting in a queue will also show in *pg_locks*. To see how many queries are waiting to run from a particular resource queue, use the *pg_resqueue_status* system catalog view. For example:

```
SELECT * FROM pg_resqueue_status;
```

Checking Query Status and System Utilization

System monitoring utilities such as *ps*, *top*, *iostat*, *vmstat*, *netstat* and so on can be used to monitor database activity on the hosts in your Greenplum Database array. These tools can be used to help identify Greenplum Database processes (*postgres* processes) currently running on the system and the most resource intensive tasks with regards to CPU, memory, disk I/O, or network activity. Looking at these system statistics can help identify queries that are overloading the system by consuming excessive resources and thereby degrading database performance. Greenplum Database comes with an management tool called *gpssh*, which allows you to run these system monitoring commands on several hosts at once.

The Greenplum Performance Monitor also collects query and system utilization metrics. See the *Greenplum Database Performance Monitor Guide* for information on enabling Greenplum Performance Monitor.

Troubleshooting Problem Queries

If a query is performing poorly, looking at its query plan can help identify problem areas. You can use the *EXPLAIN* command to see the query plan for a given query. See “[Query Profiling](#)” on page 148 for more information on reading query plans and identifying problems with a plan.

Investigating Error Messages

Greenplum Database log messages are written to files in the *pg_log* directory within the master’s or segment’s data directory. Log files are rolled over daily and are named using the naming convention: *gpdb-YYYY-MM-DD*. For example, to locate the log files on the master host:

```
$ cd $MASTER_DATA_DIRECTORY/pg_log
```

The master log file contains the most information and should always be checked first. Log lines have the format of:

```
timestamp | user | database | statement_id | con# cmd#
|:-LOG_LEVEL: log_message
```

You may want to focus your search for WARNING, ERROR, FATAL or PANIC log level messages. The Greenplum utility, `gplogfilter`, can be used to search through Greenplum Database log files. For example, the following command (when run on the master host) will check for problem log messages in the standard logging locations:

```
$ gplogfilter -t
```

To search for any related log entries in the segment log files, you can run `gplogfilter` on the segment hosts using `gpssh`. One way to identify corresponding log entries is by the `statement_id` or `con#` (session identifier). For example, to search for log messages in the segment log files containing the string `con6` and save output to a file:

```
gpssh -f seg_hosts_file -e 'source
/usr/local/greenplum-db/greenplum_path.sh ; gplogfilter -f
con6 /gpdata/*/pg_log/gpdb*.log' > seglog.out
```

Gathering Information for Greenplum Support

The `gpdetective` utility collects information from a running Greenplum Database system and creates a bzip2-compressed tar output file. This output file can then be sent to Greenplum Customer Support to help with the diagnosis of Greenplum Database errors or system failures. Run `gpdetective` on your master host, for example:

```
$ gpdetective -f /var/data/my043008gp.tar
```

Section VII: Extending Greenplum Database

This section describes how to extend the functionality of Greenplum Database by developing your own functions and programs.

- [Using Greenplum MapReduce](#)

28 Using Greenplum MapReduce

- [About Greenplum MapReduce](#)
- [Programming Greenplum MapReduce](#)
- [Submitting MapReduce Jobs for Execution](#)
- [Troubleshooting Problems with MapReduce Jobs](#)

About Greenplum MapReduce

[MapReduce](#) is a programming model developed by Google for processing and generating large data sets on an array of commodity servers. Greenplum MapReduce allows programmers who are familiar with the MapReduce paradigm to write map and reduce functions and submit them to the Greenplum Database parallel data flow engine for processing. The Greenplum Database system takes care of the details of distributing the input data, executing the program across a set of machines, handling machine failures, and managing the required inter-machine communication.

The Basics of MapReduce

In general, MapReduce is a simple dataflow programming model that utilizes user-defined functions to pass data items from one stage of processing to the next. MapReduce programs typically start with a large data file that is broken down into smaller, contiguous pieces called *splits*, which are akin to database rows. Typically each split is parsed into (*key, value*) pairs that are then sent to a Map module. The Map module invokes a user-defined Map function on each pair and produces new (*key, output_list*) pairs. Each new (*key, output_list*) pair is then passed to a Reduce module. The Reduce module gathers these pairs together, grouping them by key, and then calls a user-defined Reduce function to produce one reduced output list for each distinct input key. Both the Map and Reduce modules utilize parallelism to enable many Map tasks and Reduce tasks to be worked on at the same time on a number of machines.

The MapReduce engine acts as an abstraction allowing programmers to focus on their desired data computations while hiding the details of parallelism, distribution, load balancing and fault tolerance.

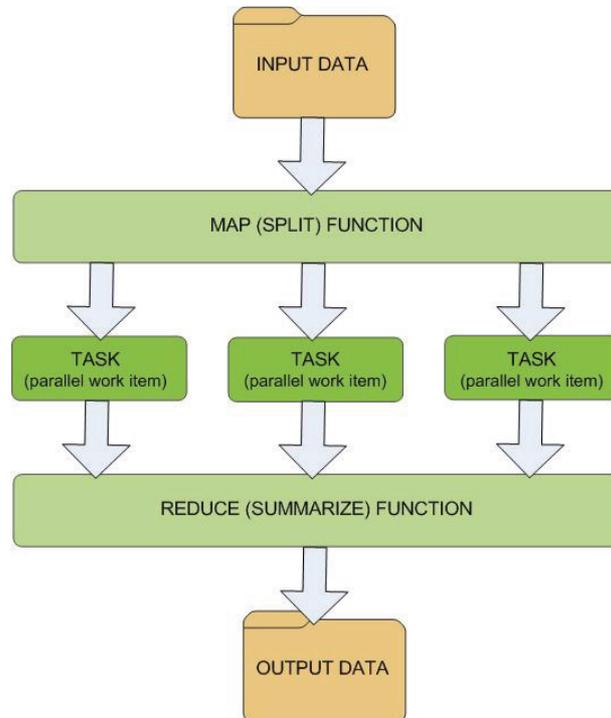


Figure 28.1 Basic MapReduce Data Flow

How Greenplum MapReduce Works

In order for Greenplum to be able to process the Map and Reduce functions written by a user, the functions need to be defined in a specially formatted Greenplum MapReduce document, which is then passed to the Greenplum MapReduce program, `gpmapreduce`, for execution by the Greenplum Database parallel engine.

The Greenplum MapReduce document defines the parts that comprise a complete MapReduce job:

- **Input Data** - Input data can come from a number of sources either inside or outside the database. Greenplum Database supports a number of file formats for external data as well as SQL for data already stored in the database.
- **Map Function** - Users provide their own map function(s) written in Python or PERL.
- **Reduce Function** - Users provide their own reduce function(s) written in Python or PERL, or use one of the built-in reduce functions.
- **Output Data** - Output can be persistently stored in the database or directed to standard output or an external file.

Once you have defined your MapReduce job in the specification document, you then use the `gpmapper` client program to submit your job to the Greenplum Database parallel data flow engine for execution.

Programming Greenplum MapReduce

In order to submit a MapReduce job to Greenplum Database for execution, you must define your MapReduce job in a special Greenplum MapReduce specification document. Greenplum MapReduce uses the YAML document framework and implements its own YAML schema. See “[Greenplum MapReduce Specification](#)” on page 791.

This section walks through the stages of the Greenplum MapReduce specification and provides examples for each stage:

- [Defining Inputs](#)
- [Defining Map Functions](#)
- [Defining Reduce Functions](#)
- [Defining Outputs](#)
- [Putting Together a Complete MapReduce Specification](#)

These stages are specified in the `DEFINE` section of a Greenplum MapReduce specification document.

Defining Inputs

Every MapReduce job requires at least one `INPUT` data source. A data source can be a single file, files served by the Greenplum parallel file distribution program (`gpfdist`), a table in the database, an SQL `SELECT` statement, or an operating system command that outputs data.

External File Inputs

A `FILE` input describes a single file located on a machine that is a Greenplum segment host. The file must be in either text-delimited or comma-separated values (CSV) format. See “[Formatting of Input Data](#)” on page 155 for more information about the format requirements of input data files. If `COLUMNS` (delimited fields in the file) are not

specified, the entire file is treated as one big text column named `value` by default. You must be a Greenplum Database superuser to run MapReduce jobs with a `FILE` input.

```
- INPUT:
  NAME: my_file_input
  FILE: seghostname:/var/data/gpfiles/employees.txt
  COLUMNS
    - first_name text
    - last_name text
    - dept text
    - hire_date text
  FORMAT: TEXT
  DELIMITER: |
```

A `GPFDIST` input is similar to `FILE`, except that the file is served by `gpfdist`, rather than the file system on a single segment host. See [“Using the Greenplum File Server \(gpfdist\)”](#) on page 156 for instructions on setting up a `gpfdist` file server. One advantage of using the `gpfdist` file server (as opposed to the `FILE` input) is that it ensures that all of the segments in your Greenplum Database system are fully utilized when reading the external data file(s). You must be a Greenplum Database superuser to run MapReduce jobs with `GPFDIST` input unless the server configuration parameter `gp_external_grant_privileges` is set to on.

```
- INPUT:
  NAME: my_distributed_input

  # specifies the host, port and the desired files served
  # by gpfdist. /* denotes all files on the gpfdist server
  GPFDIST:
    - gpfdisthost:8080/*
  COLUMNS
    - first_name text
    - last_name text
    - dept text
    - hire_date text
  FORMAT: TEXT
  DELIMITER: |
```

Database Inputs

You can use data already stored in Greenplum Database as your MapReduce data source. A `TABLE` input simply takes all data from the table name you specify. The columns and data types are already defined by the table definition.

```
- INPUT:
    NAME: my_table_input
    TABLE: sales
```

Similarly, a `QUERY` input specifies a `SELECT` statement that returns selected data from one or more tables. The columns and data types are already defined by the sourced table columns.

```
- INPUT:
    NAME: my_query_input
    QUERY: SELECT vendor, amt FROM sales WHERE region='usa';
```

OS Executable Inputs

An `EXEC` input allows you to specify a shell command or script that will be executed by all Greenplum segments. The combined output of all the segment processes comprise your data source. The command is executed by all active segment instances on all segment hosts. For example, if each segment host has four primary segment instances running, the command will be executed four times per segment host. Data is comprised of the output of the command at the time the MapReduce job is executed on each segment instance. All segment instances execute the command in parallel. If your command calls a script or program, that executable must reside on all Greenplum segment hosts.

```
- INPUT:
    NAME: my_query_input
    EXEC: /var/load_scripts/get_log_data.sh
    COLUMNS
      - url text
      - date timestamp
    FORMAT: TEXT
    DELIMITER: |
```

If you use environment variables in external web table commands (such as `$PATH`), keep in mind that the command is executed from within the database and not from a login shell. Therefore the `.bashrc` or `.profile` of the current user will not be sourced. However, you can set desired environment variables from within the `EXEC` definition itself, for example:

```
EXEC 'export PATH=/var/myscripts:$PATH; get_log_data.sh;'
```

In order to use this input type, you must be a Greenplum Database superuser and have the server configuration parameter `gp_external_enable_exec` set to `on` in your `master postgresql.conf` file.

Defining Map Functions

To borrow from database terminology, a Map function takes as input a single *row* (a set of values assigned to parameters), and produces zero or more rows of output. By default, the input and output are both defined to have two parameters of type `text`, called `key` and `value`. However, Greenplum MapReduce allows arbitrary parameter lists to be defined for both the input and the output, in the style of SQL table definitions. The input format is specified in the `PARAMETERS` definition in the `MAP` specification; the output format is specified in the `RETURNS` definition. The `RETURNS` definition requires each output parameter to be assigned a SQL data type for use in subsequent steps involving table outputs or SQL query inputs. When in doubt, SQL's `text` data type will usually work fine, since both PERL and Python will interpret text strings appropriately.

The `MAP` specification also includes a `FUNCTION` definition that provides the code for the function, in a scripting language specified via `LANGUAGE`. Supported languages currently include `PERL` and `PYTHON`.

A typical Map function definition uses the values in the `PARAMETERS` in some way to generate appropriate output values matching the format of the `RETURNS` declaration. So the main issue for defining a Map function is to know how to access the `PARAMETERS` from within the scripting language, and how to prepare the output needed for `RETURNS`.

Map Functions in PERL

Following PERL style, the `PARAMETERS` to a Map function are available in the usual `@_` parameters list. A typical first step in a PERL function is to extract the parameters into local variables via an assignment statement.

The output of a Map function must be a PERL hash, with a hash-key for each parameter in the `RETURNS` definition. Output is typically returned to the MapReduce runtime engine via a special PERL function called `return_next`. This function behaves like a normal `return`, except that when the map code is re-invoked to generate another output row, it will pick up processing on the line after the last `return_next` that was executed (analogous to Python's `yield` statement). This programming style makes it possible to take a single row as input, and return multiple outputs (each being passed back via `return_next` from within a PERL loop). When there are no more results to pass back, a standard PERL `return undef` call will tell the MapReduce harness to proceed with the next row of input, starting at the top of the Map function.

If you know that your Map function will only return one row of output for every input, you can specify the definition `MODE: SINGLE` in your `MAP` specification, and pass back a hash using a standard PERL `return` call, rather than `return_next`.

The following simple Map example converts a row containing a comma-separated value into multiple rows, one per value. Note the vertical bar (the YAML ‘literal’ marker) after the `FUNCTION:` declaration, indicating that the subsequent indented lines are to be considered a single literal string.

```
- MAP:
  NAME: perl_splitter
  LANGUAGE: PERL
  PARAMETERS: [key, value]
  RETURNS: [key text, value text]
  FUNCTION: |
    my ($key, $value) = @_;
    my @list = split(/,/ , $value);
    for my $item(@list) {
      return_next({"key" => $key, "value" => $item});
    }
    return undef;
```

Map Functions in Python

In Python, the `PARAMETERS` specified for a Map function are available as local Python variables. No PERL-style parameter interpretation is necessary.

The output of a Map function must be a (Python) dictionary, with a key for each parameter in the `RETURNS` definition. Output is typically returned to the MapReduce harness via the Python `yield` construct - but when the Map code is re-invoked to generate another output row, it will pick up processing on the line after the last `yield` that was executed. This programming style makes it possible to take a single row as input, and return multiple outputs (each being passed back via `yield` from within a Python loop). When there are no more results to pass back, the Python code should simply ‘drop through’ to the end of the script. This tells the MapReduce harness to proceed with the next row of input, starting at the top of the Map function.

If you know that your Map function will only return one row of output for every input, you can specify the definition `MODE: SINGLE` in your `MAP` specification, and pass back a hash using a standard Python return call, rather than `yield`.

The following simple Map example converts a row containing a comma-separated value into multiple rows, one per value. Note the vertical bar - a YAML ‘literal’ marker - after the `FUNCTION:` declaration, indicating that the subsequent indented lines are to be considered a single literal string.

```
- MAP:
  NAME: py_splitter
  LANGUAGE: PYTHON
  PARAMETERS: [key, value]
  RETURNS: [key text, value text]
  FUNCTION: |
    list = value.split(',')
    for item in list:
      yield {'key': key, 'value': item}
```

Defining Reduce Functions

Reduce functions handle a set of input rows that have matching values in a particular attribute (or set of attributes), and produce a single ‘reduced’ row.

Greenplum Database provides several predefined `REDUCE` functions you can execute, which all operate over a column named `value`:

- `IDENTITY` - returns (key, value) pairs unchanged
- `SUM` - calculates the sum of numeric data
- `AVG` - calculates the average of numeric data
- `COUNT` - calculates the count of input data
- `MIN` - calculates minimum value of numeric data
- `MAX` - calculates maximum value of numeric data

To use one of the predefined `REDUCE` jobs, you can simply declare it by name in the `EXECUTE` portion of your MapReduce specification document. For example:

```
EXECUTE
- RUN
  SOURCE: input_or_task_name
  MAP: map_function_name
  REDUCE: IDENTITY
```

Writing custom Reduce functions is a bit more involved than writing Map functions, because the Reduce has to be defined to work through a *set* of input rows, not just a single row. To achieve this, you must define a `TRANSITION` function associated with the `REDUCE`, which is called once for each input row. In order to allow you to ‘remember’ information between calls of the transition function, it takes as its *first* input parameter a variable called `state`. Before a set of tuples is to be Reduced, the `state` variable is initialized to the value specified in your `INITIALIZE` definition.

This value must be an SQL data type. For example, a (single-quoted) SQL text string. During the processing of a set, the `state` variable records the most recent return value of the `TRANSITION` function. After the last row in the set is processed by the `TRANSITION` function, the `state` variable is passed to the `FINALIZE` function, which can return multiple rows (via PERL's `return_next` or Python's `yield`). Each row returned must be a hash representing the reduced output row.

By default, the parameters to a Reduce are (`key`, `value`) pairs. However, for custom Reduce functions, an arbitrary list of columns can be passed in. The `KEYS` definition defines the column or columns used to partition the input into subsets to be Reduced; the default value of the `KEYS` definition is the column called `key`. In the absence of a `KEYS` definition, the `key` is defined to be the set of parameters not mentioned in the `TRANSITION` function's `PARAMETERS` list.

As a performance optimization, you can optionally define a `CONSOLIDATE` function, which consolidates multiple `state` variables into a single `state` variable. This allows Greenplum Database to send a `state` variable between machines in lieu of a set of input tuples, substantially lowering the amount of network traffic over the Greenplum interconnect. `CONSOLIDATE` is similar to `TRANSITION` in its structure, taking two `state` variables at each invocation and returning a single `state`.

Below is a complete PERL Reduce function definition for computing the average of all positive values:

```
- REDUCE:
  NAME: perl_pos_avg
  TRANSITION: perl_pos_avg_trans
  CONSOLIDATE: perl_pos_avg_cons
  FINALIZE: perl_pos_avg_final
  INITIALIZE: '0,0'
  KEYS: [key]
```

```
- TRANSITION:
  NAME: perl_pos_avg_trans
  PARAMETERS: [state, value]
  RETURNS: [state text]
  LANGUAGE: perl
  FUNCTION: |
    my ($state, $value) = @_;
    my ($count, $sum) = split(/,/ , $state);
    if ($value > 0) {
      $sum += $value;
      $count++;
      $state = $count . "," . $sum;
    }
    return $state;
```

```

- CONSOLIDATE:
  NAME: perl_pos_avg_cons
  PARAMETERS: [state, value]
  RETURNS: [state text]
  LANGUAGE: perl
  FUNCTION: |
    my ($state, $value) = @_;
    my ($scount, $ssum) = split(/,/, $state);
    my ($vcount, $vsum) = split(/,/, $value);
    my $count = $scount + $vcount;
    my $sum = $ssum + $vsum;
    return ($count . "," . $sum);

```

```

- FINALIZE:
  NAME: perl_pos_avg_final
  PARAMETERS: [state]
  RETURNS: [value float]
  LANGUAGE: perl
  FUNCTION: |
    my ($state) = @_;
    my ($count, $sum) = split(/,/, $state);
    return_next ($count*1.0/$sum);
    return undef;

```

Defining Outputs

Defining an `OUTPUT` specification is optional. If no output is defined, the default is to send the final results to standard output of the Greenplum MapReduce client. You can also direct output to a file on the Greenplum MapReduce client or to a table in the database by defining an `OUTPUT` specification.

Table Outputs

A `TABLE` output defines a table in the database where the final output of your MapReduce job will be stored. By default, a table of the given `TABLE` name will be created in the database if it does not already exist. If the named table does exist in the database, you must declare a `MODE` to specify if you want to add the output to the table

(APPEND) or drop and recreate the table (REPLACE). By default, the table will be distributed by the REDUCE keys or you can optionally declare a distribution column using the KEYS specification.

```
- OUTPUT:
  NAME: gpmr_output
  TABLE: wordcount_out
  KEYS:
    - value
  MODE: REPLACE
```

File Outputs

A FILE output defines a file location on the Greenplum MapReduce client where the output data will be written to. The named file will be created when the MapReduce job is run. The file cannot already exist or else the job will return an error.

```
- OUTPUT:
  NAME: gpmr_output
  FILE: /var/data/mapreduce/wordcount.out
```

Defining Tasks

A TASK specification is optional, but can be useful in multi-stage MapReduce jobs. A task defines a complete end-to-end INPUT/MAP/REDUCE stage within a complete Greenplum MapReduce job pipeline. Once defined, a TASK object can be called as input to further processing stages.

For example, suppose you have defined a table INPUT called `documents` and another called `keywords`. Each respective table input is processed by its own MAP function `document_map` and `keyword_map`. If you wanted to use the results of these processing stages as input to further stages in your MapReduce job, you could define two tasks as follows:

```
- TASK:
  NAME: document_prep
  SOURCE: documents
  MAP: document_map

- TASK:
  NAME: keyword_prep
  SOURCE: keywords
  MAP: keyword_map
```

These named tasks can then be called as input in a later processing stage. In this example, we are defining an SQL `QUERY` input that joins the results of the two tasks we defined earlier (`document_prep` and `keyword_prep`).

```
- INPUT:
  NAME: term_join
  QUERY: |
    SELECT doc.doc_id, kw.keyword_id, kw.term, kw.nterms,
           doc.positions as doc_positions,
           kw.positions as kw_positions
    FROM document_prep doc INNER JOIN keyword_prep kw
    ON (doc.term = kw.term)
```

Putting Together a Complete MapReduce Specification

Once you have defined all of the stages of your MapReduce job in the `DEFINE` section of your [Greenplum MapReduce Specification](#) document, you must define an `EXECUTE` section to specify the final `INPUT/MAP/REDUCE` stage. All of the objects named in the `EXECUTE` section are defined earlier in the Greenplum MapReduce specification `DEFINE` section.

```
EXECUTE:
- RUN:
  SOURCE: input_or_task_name
  TARGET: output_name
  MAP: map_function_name
  REDUCE: reduce_function_name
```

Submitting MapReduce Jobs for Execution

Once you have defined your MapReduce program in a [Greenplum MapReduce Specification](#) document, you can submit the document to Greenplum Database for execution using the `gpmareduce` client program. This is a database client program similar to `psql` in that it requires you to supply connection information such as the database you are connecting to, the database user you are connecting as, the host and port of the Greenplum master, and so on. You can either specify the connection options on the command-line or use the environment variable settings `$PGDATABASE`, `$PGUSER`, `$PGHOST` and `$PGPORT` (if set). For example:

```
gpmareduce -h gpmasterhost -p 54321 -f my_gpmr_spec.yml mydatabase
```

Creating the Languages in the Database

During execution of your map and reduce functions, Greenplum MapReduce makes use of the procedural languages built in to the database. Greenplum requires these languages to be created in the database prior to executing your MapReduce jobs. Use

the `CREATE LANGUAGE` command to create the language in the database you will be using to execute your MapReduce jobs. You must be a database super user to create a language. For example to create the PERL procedural language:

```
$ psql -c 'CREATE LANGUAGE plperl;'
```

And to create the Python procedural language:

```
$ psql -c 'CREATE LANGUAGE plpythonu;'
```

Troubleshooting Problems with MapReduce Jobs

This section describes some common errors encountered when executing Greenplum MapReduce jobs and how to resolve them.

Language Does Not Exist

Symptom:

```
ERROR: language "pl*" does not exist
HINT: Use CREATE LANGUAGE to load the language into the database.
```

Explanation:

During execution of your map and reduce functions, Greenplum MapReduce makes use of the procedural languages built in to the database. Greenplum requires these languages to be created in the database prior to executing your MapReduce jobs.

Solution:

If the language you are trying to use has not been created in the database you are connecting to, you (or your Greenplum administrator) will need to create the language for you before you can proceed. For example:

```
psql database_name -c 'CREATE LANGUAGE plperl;'
```

See `CREATE LANGUAGE` for more information. Some languages may require database superuser privileges to use.

Generic Python Iterator Error

Symptom:

```
ERROR: plpython: function "function_name" error fetching next item
from iterator
DETAIL:
<type 'exceptions.IOError'>: [Errno 2] No such file or directory:
'/tmp/file/doesnt/exist' (, line 39)
```

Explanation:

This is an error returned from Python that occurs whenever there was an error in an iterator function (such as a function which makes use of `yield`). This usually indicates a bug in your Python code.

Solution:

The easiest way to debug the problem is to wrap your function with additional exception handling. For example:

```
FUNCTION: |
  try:
    ...
    user code
    ...
  except Exception, e:
    plpy.warning('my function name:' + str(e))
```

This will generate a warning message that is more useful for debugging.

Function Defined Using Wrong MODE

Symptom:

```
ERROR: set-returning PERL function must return reference to array
or use return_next
```

```
-----
ERROR: composite-returning PERL function must return reference
to hash
```

```
-----
ERROR: returned sequence's length must be same as tuple's length
```

```
-----
ERROR: no attribute named "key"
HINT: to return null in specific column, let returned object to
have attribute named after column with value None
```

Explanation:

There are two primary modes that a Greenplum MapReduce function can be in:

- **MODE: SINGLE** returns precisely one row for every row received.
- **MODE: MULTI** may return any number of rows from 0-*N* for every row received.

TRANSITION and **CONSOLIDATE** functions only support **SINGLE** mode since they are effectively finite state functions and must return the next state.

MAP and **FINALIZE** functions support both modes defaulting to **MULTI** mode since it is more general.

Solution:

In a **SINGLE** mode function you must use the language's `return` method to return a single row of data.

In `MULTI` mode it is best to write the function as a generator function. In Python this is done by using `yield`. in PERL this is done using `return_next`. For example:

```
- MAP:
  NAME:      perl_single
  MODE:      SINGLE
  PARAMETERS: [key text, value text]
  RETURNS:   [key text, value text]
  LANGUAGE:  perl
  FUNCTION: |
    my ($key, $value) = @_;
    return {'key' => $key, 'value' => $value}
```

```
- MAP:
  NAME:      perl_multi
  MODE:      MULTI
  PARAMETERS: [key text, value text]
  RETURNS:   [key text, value text]
  LANGUAGE:  perl
  FUNCTION: |
    my ($key, $value) = @_;
    for my $i (0..10) {
      return_next {'key' => $key, 'value' => $value}
    }
    return undef
```

```
- MAP:
  NAME:      python_single
  MODE:      SINGLE
  PARAMETERS: [key text, value text]
  RETURNS:   [key text, value text]
  LANGUAGE:  python
  FUNCTION: |
    return {'key': key, 'value': value}
```

```
- MAP:
  NAME:      python_multi
  MODE:      MULTI
  PARAMETERS: [key text, value text]
  RETURNS:   [key text, value text]
  LANGUAGE:  python
  FUNCTION: |
    try:
      for i in range(0,10):
        yield {'key': key, 'value': value}
    except Exception, e:
      plpy.warning('python_multi: ' +str(e))
```

The most common occurrence of the ‘returned sequence’s length ...’ error occurs when you try to use `return` from a `MULTI` mode function. In this context, the single row is interpreted not as a single row with columns, but as a set of rows each one column wide. Each column is too small to be a full row, so the returned sequence’s length does not match and an error is given.

A less common scenario that can generate the same error is when the number of columns returned does not match the declared number. For example:

```
- MAP:
  NAME:      python_error
  MODE:      SINGLE
  PARAMETERS: [key text, value text]
  RETURNS:   [key text, value text]
  LANGUAGE:  python
  FUNCTION: |
            return {'key': key}
```

Because the function declared that it returns a `key` and a `value` but actually only returns a `key`, this can also cause the ‘returned sequence’s length ...’ error.

Section VIII: References

This section contains the following references:

- [SQL Command Reference](#)
- [Management Utility Reference](#)
- [Client Utility Reference](#)
- [Server Configuration Parameters](#)
- [Initialization Configuration File Reference](#)
- [Greenplum MapReduce Specification](#)
- [Greenplum Environment Variables](#)
- [Greenplum Database Data Types](#)
- [System Catalog Reference](#)
- [Glossary](#)

A. SQL Command Reference

This appendix provides references for the SQL commands available in Greenplum Database:

- ABORT
- ALTER AGGREGATE
- ALTER CONVERSION
- ALTER DATABASE
- ALTER DOMAIN
- ALTER FUNCTION
- ALTER GROUP
- ALTER INDEX
- ALTER LANGUAGE
- ALTER OPERATOR
- ALTER OPERATOR CLASS
- ALTER RESOURCE QUEUE
- ALTER ROLE
- ALTER SCHEMA
- ALTER SEQUENCE
- ALTER TABLE
- ALTER TABLESPACE
- ALTER TRIGGER
- ALTER TYPE
- ALTER USER
- ANALYZE
- BEGIN
- CHECKPOINT
- CLOSE
- CLUSTER
- COMMENT
- COMMIT
- COPY
- CREATE AGGREGATE
- CREATE CAST
- CREATE CONVERSION
- CREATE DATABASE
- CREATE DOMAIN
- CREATE EXTERNAL TABLE
- CREATE FUNCTION
- CREATE GROUP
- CREATE INDEX
- CREATE LANGUAGE
- CREATE OPERATOR
- CREATE OPERATOR CLASS
- CREATE RESOURCE QUEUE
- CREATE ROLE
- CREATE RULE
- CREATE SCHEMA
- CREATE SEQUENCE
- CREATE TABLE
- CREATE TABLE AS
- CREATE TABLESPACE
- CREATE TRIGGER
- CREATE TYPE
- CREATE USER
- CREATE VIEW
- DEALLOCATE
- DECLARE
- DELETE
- DROP AGGREGATE
- DROP CAST
- DROP CONVERSION
- DROP DATABASE
- DROP DOMAIN

- DROP EXTERNAL TABLE
- DROP FUNCTION
- DROP GROUP
- DROP INDEX
- DROP LANGUAGE
- DROP OPERATOR
- DROP OPERATOR CLASS
- DROP OWNED
- DROP RESOURCE QUEUE
- DROP ROLE
- DROP RULE
- DROP SCHEMA
- DROP SEQUENCE
- DROP TABLE
- DROP TABLESPACE
- DROP TRIGGER
- DROP TYPE
- DROP USER
- DROP VIEW
- END
- EXECUTE
- EXPLAIN
- FETCH
- GRANT
- INSERT
- LOAD
- LOCK
- MOVE
- PREPARE
- REASSIGN OWNED
- REINDEX
- RELEASE SAVEPOINT
- RESET
- REVOKE
- ROLLBACK
- ROLLBACK TO SAVEPOINT
- SAVEPOINT
- SELECT
- SELECT INTO
- SET
- SET ROLE
- SET SESSION AUTHORIZATION
- SET TRANSACTION
- SHOW
- START TRANSACTION
- TRUNCATE
- UPDATE
- VACUUM
- VALUES

SQL Syntax Summary

ABORT

Aborts the current transaction.

```
ABORT [WORK | TRANSACTION]
```

ALTER AGGREGATE

Changes the definition of an aggregate function

```
ALTER AGGREGATE name ( type [ , ... ] ) RENAME TO new_name
ALTER AGGREGATE name ( type [ , ... ] ) OWNER TO new_owner
ALTER AGGREGATE name ( type [ , ... ] ) SET SCHEMA new_schema
```

ALTER CONVERSION

Changes the definition of a conversion.

```
ALTER CONVERSION name RENAME TO newname
ALTER CONVERSION name OWNER TO newowner
```

ALTER DATABASE

Changes the attributes of a database.

```
ALTER DATABASE name [ WITH CONNECTION LIMIT conlimit ]
ALTER DATABASE name SET parameter { TO | = } { value | DEFAULT }
ALTER DATABASE name RESET parameter
ALTER DATABASE name RENAME TO newname
ALTER DATABASE name OWNER TO new_owner
```

ALTER DOMAIN

Changes the definition of a domain.

```
ALTER DOMAIN name { SET DEFAULT expression | DROP DEFAULT }
ALTER DOMAIN name { SET | DROP } NOT NULL
ALTER DOMAIN name ADD domain_constraint
ALTER DOMAIN name DROP CONSTRAINT constraint_name [RESTRICT | CASCADE]
ALTER DOMAIN name OWNER TO new_owner
ALTER DOMAIN name SET SCHEMA new_schema
```

ALTER FUNCTION

Changes the definition of a function.

```
ALTER FUNCTION name ( [ [argmode] [argname] argtype [, ...] ] ) action [, ... ]
[RESTRICT]
ALTER FUNCTION name ( [ [argmode] [argname] argtype [, ...] ] ) RENAME TO new_name
ALTER FUNCTION name ( [ [argmode] [argname] argtype [, ...] ] ) OWNER TO new_owner
ALTER FUNCTION name ( [ [argmode] [argname] argtype [, ...] ] ) SET SCHEMA
```

new_schema

where *action* is one of:

```
{CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT}
{IMMUTABLE | STABLE | VOLATILE}
{[EXTERNAL] SECURITY INVOKER | [EXTERNAL] SECURITY DEFINER}
```

ALTER GROUP

Changes a role name or membership.

```
ALTER GROUP groupname ADD USER username [, ... ]
ALTER GROUP groupname DROP USER username [, ... ]
ALTER GROUP groupname RENAME TO newname
```

ALTER INDEX

Changes the definition of an index.

```
ALTER INDEX name RENAME TO new_name
ALTER INDEX name SET TABLESPACE tablespace_name
ALTER INDEX name SET ( FILLFACTOR = value )
ALTER INDEX name RESET ( FILLFACTOR )
```

ALTER LANGUAGE

Changes the name of a procedural language.

```
ALTER LANGUAGE name RENAME TO newname
```

ALTER OPERATOR

Changes the definition of an operator.

```
ALTER OPERATOR name ( {lefttype | NONE} , {righttype | NONE} ) OWNER TO newowner
```

ALTER OPERATOR CLASS

Changes the definition of an operator class.

```
ALTER OPERATOR CLASS name USING index_method RENAME TO newname
ALTER OPERATOR CLASS name USING index_method OWNER TO newowner
```

ALTER RESOURCE QUEUE

Changes the limits of a resource queue.

```
ALTER RESOURCE QUEUE name ACTIVE THRESHOLD integer
[COST THRESHOLD float [OVERCOMMIT | NOOVERCOMMIT]
[IGNORE THRESHOLD float]]
ALTER RESOURCE QUEUE name COST THRESHOLD float
[OVERCOMMIT | NOOVERCOMMIT] [IGNORE THRESHOLD float]
[ACTIVE THRESHOLD integer]
```

ALTER ROLE

Changes a database role (user or group).

```
ALTER ROLE name RENAME TO newname
ALTER ROLE name SET config_parameter {TO | =} {value | DEFAULT}
ALTER ROLE name RESET config_parameter
ALTER ROLE name RESOURCE QUEUE {queue_name | NONE}
ALTER ROLE name [ [WITH] option [ ... ] ]
```

where *option* can be:

```
    SUPERUSER | NOSUPERUSER
    | CREATEDB | NOCREATEDB
    | CREATEROLE | NOCREATEROLE
    | INHERIT | NOINHERIT
    | LOGIN | NOLOGIN
    | CONNECTION LIMIT conlimit
    | [ENCRYPTED | UNENCRYPTED] PASSWORD 'password'
    | VALID UNTIL 'timestamp'
```

ALTER SCHEMA

Changes the definition of a schema.

```
ALTER SCHEMA name RENAME TO newname
ALTER SCHEMA name OWNER TO newowner
```

ALTER SEQUENCE

Changes the definition of a sequence generator.

```
ALTER SEQUENCE name [INCREMENT [ BY ] increment]
    [MINVALUE minvalue | NO MINVALUE]
    [MAXVALUE maxvalue | NO MAXVALUE]
    [RESTART [ WITH ] start]
    [CACHE cache] [[ NO ] CYCLE]
    [OWNED BY {table.column | NONE}]
ALTER SEQUENCE name SET SCHEMA new_schema
```

ALTER TABLE

Changes the definition of a table.

```
ALTER TABLE [ONLY] name RENAME [COLUMN] column TO new_column
ALTER TABLE name RENAME TO new_name
ALTER TABLE name SET SCHEMA new_schema
ALTER TABLE [ONLY] name SET
    DISTRIBUTED BY (column, [ ... ] )
    | DISTRIBUTED RANDOMLY
    | WITH (REORGANIZE=true|false)
ALTER TABLE [ONLY] name action [, ... ]
ALTER TABLE name
    [ ALTER PARTITION { partition_name | FOR (RANK(number)) | FOR (value) }
```

```
partition_action [...] ]
  partition_action
```

where *action* is one of:

```
ADD [COLUMN] column_name type [column_constraint [ ... ] ]
DROP [COLUMN] column [RESTRICT | CASCADE]
ALTER [COLUMN] column TYPE type [USING expression]
ALTER [COLUMN] column SET DEFAULT expression
ALTER [COLUMN] column DROP DEFAULT
ALTER [COLUMN] column { SET | DROP } NOT NULL
ALTER [COLUMN] column SET STATISTICS integer
ADD table_constraint
DROP CONSTRAINT constraint_name [RESTRICT | CASCADE]
DISABLE TRIGGER [trigger_name | ALL | USER]
ENABLE TRIGGER [trigger_name | ALL | USER]
CLUSTER ON index_name
SET WITHOUT CLUSTER
SET WITHOUT OIDS
SET (FILLFACTOR = value)
RESET (FILLFACTOR)
INHERIT parent_table
NO INHERIT parent_table
OWNER TO new_owner
SET TABLESPACE new_tablespace
ALTER DEFAULT PARTITION
DROP DEFAULT PARTITION [IF EXISTS]
DROP PARTITION [IF EXISTS] { partition_name |
  FOR (RANK(number)) | FOR (value) } [CASCADE]
TRUNCATE DEFAULT PARTITION
TRUNCATE PARTITION { partition_name | FOR (RANK(number)) |
  FOR (value) }
RENAME DEFAULT PARTITION TO new_partition_name
RENAME PARTITION { partition_name | FOR (RANK(number)) |
  FOR (value) } TO new_partition_name
ADD DEFAULT PARTITION name [ ( subpartition_spec ) ]
ADD PARTITION [name] partition_element
  [ ( subpartition_spec ) ]
EXCHANGE PARTITION { partition_name | FOR (RANK(number)) |
  FOR (value) } WITH TABLE table_name
  [ WITH | WITHOUT VALIDATION ]
EXCHANGE DEFAULT PARTITION WITH TABLE table_name
  [ WITH | WITHOUT VALIDATION ]
SET SUBPARTITION TEMPLATE (subpartition_spec)
SPLIT DEFAULT PARTITION AT (list_value)
  START([datatype] range_value) [INCLUSIVE | EXCLUSIVE]
  END([datatype] range_value) [INCLUSIVE | EXCLUSIVE]
  [ INTO ( PARTITION new_partition_name,
    PARTITION default_partition_name ) ]
SPLIT PARTITION { partition_name | FOR (RANK(number)) |
  FOR (value) } AT (value)
  [ INTO (PARTITION partition_name, PARTITION partition_name)]
```

where *partition_element* is:

```
VALUES (list_value [, ... ] )
| START ([datatype] 'start_value') [INCLUSIVE | EXCLUSIVE]
  [ END ([datatype] 'end_value') [INCLUSIVE | EXCLUSIVE] ]
| END ([datatype] 'end_value') [INCLUSIVE | EXCLUSIVE]
[ WITH ( partition_storage_parameter=value [, ... ] ) ]
```

```
[ TABLESPACE tablespace ]
where subpartition_spec is:
subpartition_element [, ...]
and subpartition_element is:
  DEFAULT SUBPARTITION name
  | [SUBPARTITION name] VALUES (list_value [,... ] )
  | [SUBPARTITION name]
    START ([datatype] 'start_value') [INCLUSIVE | EXCLUSIVE]
    [ END ([datatype] 'end_value') [INCLUSIVE | EXCLUSIVE] ]
    [ EVERY ( [number | datatype] 'interval_value') ]
  | [SUBPARTITION name]
    END ([datatype] 'end_value') [INCLUSIVE | EXCLUSIVE]
    [ EVERY ( [number | datatype] 'interval_value') ]
[ WITH ( partition_storage_parameter=value [, ... ] ) ]
[ TABLESPACE tablespace ]
```

where *storage_parameter* is:

```
APPENDONLY={TRUE|FALSE}
COMPRESSTYPE={ZLIB|QUICKLZ}
COMPRESSLEVEL={0-9|1}
ORIENTATION={COLUMN|ROW}
FILLFACTOR={10-100}
```

ALTER TABLESPACE

Changes the definition of a tablespace.

```
ALTER TABLESPACE name RENAME TO newname
ALTER TABLESPACE name OWNER TO newowner
```

ALTER TRIGGER

Changes the definition of a trigger.

```
ALTER TRIGGER name ON table RENAME TO newname
```

ALTER TYPE

Changes the definition of a data type.

```
ALTER TYPE name OWNER TO new_owner
ALTER TYPE name SET SCHEMA new_schema
```

ALTER USER

Changes the definition of a database role (user).

```
ALTER USER name RENAME TO newname
ALTER USER name SET config_parameter {TO | =} {value | DEFAULT}
ALTER USER name RESET config_parameter
ALTER USER name [ [WITH] option [ ... ] ]
```

where *option* can be:

```
    SUPERUSER | NOSUPERUSER
    | CREATEDB | NOCREATEDB
    | CREATEROLE | NOCREATEROLE
    | CREATEUSER | NOCREATEUSER
    | INHERIT | NOINHERIT
    | LOGIN | NOLOGIN
    | [ ENCRYPTED | UNENCRYPTED ] PASSWORD 'password'
    | VALID UNTIL 'timestamp'
```

ANALYZE

Collects statistics about a database.

```
ANALYZE [VERBOSE] [table [ (column [, ...] ) ]]
```

BEGIN

Starts a transaction block.

```
BEGIN [WORK | TRANSACTION] [SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ
UNCOMMITTED] [READ WRITE | READ ONLY]
```

CHECKPOINT

Forces a transaction log checkpoint.

```
CHECKPOINT
```

CLOSE

Closes a cursor.

```
CLOSE cursor_name
```

CLUSTER

Physically reorders a table on disk according to an index. Not a recommended operation in Greenplum Database.

```
CLUSTER indexname ON tablename
CLUSTER tablename
CLUSTER
```

COMMENT

Defines or change the comment of an object.

```
COMMENT ON
{ TABLE object_name |
  COLUMN table_name.column_name |
  AGGREGATE agg_name (agg_type [, ...]) |
  CAST (sourcetype AS targettype) |
  CONSTRAINT constraint_name ON table_name |
  CONVERSION object_name |
  DATABASE object_name |
  DOMAIN object_name |
  FUNCTION func_name ([[argmode] [argname] argtype [, ...]]) |
  INDEX object_name |
  LARGE OBJECT large_object_oid |
  OPERATOR op (leftoperand_type, rightoperand_type) |
  OPERATOR CLASS object_name USING index_method |
  [PROCEDURAL] LANGUAGE object_name |
  ROLE object_name |
  RULE rule_name ON table_name |
  SCHEMA object_name |
  SEQUENCE object_name |
  TABLESPACE object_name |
  TRIGGER trigger_name ON table_name |
  TYPE object_name |
  VIEW object_name }
IS 'text'
```

COMMIT

Commits the current transaction.

```
COMMIT [WORK | TRANSACTION]
```

COPY

Copies data between a file and a table.

```
COPY table [(column [, ...])] FROM {'file' | STDIN}
  [ [WITH]
    [BINARY]
    [OIDS]
    [DELIMITER [ AS ] 'delimiter']
    [NULL [ AS ] 'null string']
    [ESCAPE [ AS ] 'escape' | 'OFF']
    [NEWLINE [ AS ] 'LF' | 'CR' | 'CRLF']
    [CSV [HEADER] [QUOTE [ AS ] 'quote']
      [FORCE NOT NULL column [, ...]]
    [FILL MISSING FIELDS]
  [ [LOG ERRORS INTO error_table] [KEEP]
    SEGMENT REJECT LIMIT count [ROWS | PERCENT] ]
COPY {table [(column [, ...])] | (query)} TO {'file' | STDOUT}
  [ [WITH]
    [BINARY]
    [OIDS]
    [DELIMITER [ AS ] 'delimiter']
    [NULL [ AS ] 'null string']
    [ESCAPE [ AS ] 'escape' | 'OFF']
    [NEWLINE [ AS ] 'LF' | 'CR' | 'CRLF']
    [CSV [HEADER] [QUOTE [ AS ] 'quote']
      [FORCE QUOTE column [, ...]] ]
```

CREATE AGGREGATE

Defines a new aggregate function.

```
CREATE AGGREGATE name (input_data_type [ , ... ])
  ( SFUNC = sfunc,
    STYPE = state_data_type
    [, PREFUNC = pfunc]
    [, FINALFUNC = ffunc]
    [, INITCOND = initial_condition]
    [, SORTOP = sort_operator] )
```

CREATE CAST

Defines a new cast.

```
CREATE CAST (sourcetype AS targettype)
  WITH FUNCTION funcname (argtypes)
  [AS ASSIGNMENT | AS IMPLICIT]
CREATE CAST (sourcetype AS targettype) WITHOUT FUNCTION
  [AS ASSIGNMENT | AS IMPLICIT]
```

CREATE CONVERSION

Defines a new encoding conversion.

```
CREATE [DEFAULT] CONVERSION name FOR source_encoding TO dest_encoding FROM funcname
```

CREATE DATABASE

Creates a new database.

```
CREATE DATABASE name [ [WITH] [OWNER [=] dbowner]  
                    [TEMPLATE [=] template]  
                    [ENCODING [=] encoding]  
                    [TABLESPACE [=] tablespace]  
                    [CONNECTION LIMIT [=] connlimit ] ]
```

CREATE DOMAIN

Defines a new domain.

```
CREATE DOMAIN name [AS] data_type [DEFAULT expression]  
    [CONSTRAINT constraint_name  
    | NOT NULL | NULL  
    | CHECK (expression) [...]]
```

CREATE EXTERNAL TABLE

Defines a new external table or web table.

```
CREATE EXTERNAL TABLE table_name
  ( column_name data_type [, ...] | LIKE other_table )
  LOCATION ('file://seghost[:port]/path/file' [, ...])
    | ('gpfdist://filehost[:port]/file_pattern' [, ...])
  FORMAT 'TEXT'
    [( [DELIMITER [AS] 'delimiter' | 'OFF']
    [NULL [AS] 'null string']
    [ESCAPE [AS] 'escape' | 'OFF']
    [NEWLINE [ AS ] 'LF' | 'CR' | 'CRLF']
    [FILL MISSING FIELDS] )]
  | 'CSV'
    [( [HEADER]
    [QUOTE [AS] 'quote']
    [DELIMITER [AS] 'delimiter']
    [NULL [AS] 'null string']
    [FORCE NOT NULL column [, ...]]
    [ESCAPE [AS] 'escape']
    [NEWLINE [ AS ] 'LF' | 'CR' | 'CRLF']
    [FILL MISSING FIELDS] )]
  [ ENCODING 'encoding' ]
  [ [LOG ERRORS INTO error_table] SEGMENT REJECT LIMIT count
  [ROWS | PERCENT] ]

CREATE EXTERNAL WEB TABLE table_name
  ( column_name data_type [, ...] | LIKE other_table )
  LOCATION ('http://webhost[:port]/path/file' [, ...])
  | EXECUTE 'command' [ON ALL
    | MASTER
    | number_of_segments
    | HOST ['segment_hostname']
    | SEGMENT segment_id ]
  FORMAT 'TEXT'
    [( [DELIMITER [AS] 'delimiter' | 'OFF']
    [NULL [AS] 'null string']
    [ESCAPE [AS] 'escape' | 'OFF']
    [NEWLINE [ AS ] 'LF' | 'CR' | 'CRLF']
    [FILL MISSING FIELDS] )]
  | 'CSV'
    [( [HEADER]
    [QUOTE [AS] 'quote']
    [DELIMITER [AS] 'delimiter']
    [NULL [AS] 'null string']
    [FORCE NOT NULL column [, ...]]
    [ESCAPE [AS] 'escape']
    [NEWLINE [ AS ] 'LF' | 'CR' | 'CRLF']
    [FILL MISSING FIELDS] )]
  [ ENCODING 'encoding' ]
  [ [LOG ERRORS INTO error_table] SEGMENT REJECT LIMIT count
  [ROWS | PERCENT] ]
```

CREATE FUNCTION

Defines a new function.

```
CREATE [OR REPLACE] FUNCTION name
  ( [ argmode] [argname] argtype [, ...] )
  [RETURNS rettype]
  { LANGUAGE langname
  | IMMUTABLE | STABLE | VOLATILE
  | CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT
  | [EXTERNAL] SECURITY INVOKER | [EXTERNAL] SECURITY DEFINER
  | AS 'definition'
  | AS 'obj_file', 'link_symbol' } ...
```

CREATE GROUP

Defines a new database role.

```
CREATE GROUP name [ [WITH] option [ ... ] ]
```

where *option* can be:

```
  SUPERUSER | NOSUPERUSER
  | CREATEDB | NOCREATEDB
  | CREATEROLE | NOCREATEROLE
  | CREATEUSER | NOCREATEUSER
  | INHERIT | NOINHERIT
  | LOGIN | NOLOGIN
  | [ ENCRYPTED | UNENCRYPTED ] PASSWORD 'password'
  | VALID UNTIL 'timestamp'
  | IN ROLE rolename [, ...]
  | IN GROUP rolename [, ...]
  | ROLE rolename [, ...]
  | ADMIN rolename [, ...]
  | USER rolename [, ...]
  | SYSID uid
```

CREATE INDEX

Defines a new index.

```
CREATE [UNIQUE] INDEX [CONCURRENTLY] name ON table
  [USING method]
  ( {column | (expression)} [opclass] [, ...] )
  [ WITH ( FILLFACTOR = value ) ]
  [TABLESPACE tablespace]
  [WHERE predicate]
```

CREATE LANGUAGE

Defines a new procedural language.

```
CREATE [PROCEDURAL] LANGUAGE name
CREATE [TRUSTED] [PROCEDURAL] LANGUAGE name
  HANDLER call_handler [VALIDATOR valfunction]
```

CREATE OPERATOR

Defines a new operator.

```
CREATE OPERATOR name (
    PROCEDURE = funcname
    [, LEFTARG = lefttype] [, RIGHTARG = righttype]
    [, COMMUTATOR = com_op] [, NEGATOR = neg_op]
    [, RESTRICT = res_proc] [, JOIN = join_proc]
    [, HASHES] [, MERGES]
    [, SORT1 = left_sort_op] [, SORT2 = right_sort_op]
    [, LTCMP = less_than_op] [, GTCMP = greater_than_op] )
```

CREATE OPERATOR CLASS

Defines a new operator class.

```
CREATE OPERATOR CLASS name [DEFAULT] FOR TYPE data_type
    USING index_method AS
    {
        OPERATOR strategy_number op_name [(op_type, op_type)] [RECHECK]
        | FUNCTION support_number funcname (argument_type [, ...] )
        | STORAGE storage_type
    } [, ... ]
```

CREATE RESOURCE QUEUE

Defines a new resource queue.

```
CREATE RESOURCE QUEUE name ACTIVE THRESHOLD integer
[COST THRESHOLD float [OVERCOMMIT | NOOVERCOMMIT]
    [IGNORE THRESHOLD float]]

CREATE RESOURCE QUEUE name COST THRESHOLD float
[OVERCOMMIT | NOOVERCOMMIT] [IGNORE THRESHOLD float]
[ACTIVE THRESHOLD integer]
```

CREATE ROLE

Defines a new database role (user or group).

```
CREATE ROLE name [[WITH] option [ ... ]]
```

where *option* can be:

```
    SUPERUSER | NOSUPERUSER
    | CREATEDB | NOCREATEDB
    | CREATEROLE | NOCREATEROLE
    | INHERIT | NOINHERIT
    | LOGIN | NOLOGIN
    | CONNECTION LIMIT conlimit
    | [ ENCRYPTED | UNENCRYPTED ] PASSWORD 'password'
    | VALID UNTIL 'timestamp'
    | IN ROLE rolename [, ...]
    | ROLE rolename [, ...]
    | ADMIN rolename [, ...]
    | RESOURCE QUEUE queue_name
```

CREATE RULE

Defines a new rewrite rule.

```
CREATE [OR REPLACE] RULE name AS ON event
    TO table [WHERE condition]
    DO [ALSO | INSTEAD] { NOTHING | command | (command; command ...) }
```

CREATE SCHEMA

Defines a new schema.

```
CREATE SCHEMA schema_name [AUTHORIZATION username] [schema_element [ ... ]]  
CREATE SCHEMA AUTHORIZATION rolename [schema_element [ ... ]]
```

CREATE SEQUENCE

Defines a new sequence generator.

```
CREATE [TEMPORARY | TEMP] SEQUENCE name  
  [INCREMENT [BY] value]  
  [MINVALUE minvalue | NO MINVALUE]  
  [MAXVALUE maxvalue | NO MAXVALUE]  
  [START [ WITH ] start]  
  [CACHE cache]  
  [[NO] CYCLE]  
  [OWNED BY { table.column | NONE }]
```

CREATE TABLE

Defines a new table.

```

CREATE [[GLOBAL | LOCAL] {TEMPORARY | TEMP}] TABLE table_name (
    [ { column_name data_type [DEFAULT default_expr]
      [column_constraint [ ... ]]
      | table_constraint
      | LIKE other_table [{INCLUDING | EXCLUDING}
                          {DEFAULTS | CONSTRAINTS}] ...}
      [, ... ] ] )
[ INHERITS ( parent_table [, ... ] ) ]
[ WITH ( storage_parameter=value [, ... ] ) ]
[ ON COMMIT {PRESERVE ROWS | DELETE ROWS | DROP} ]
[ TABLESPACE tablespace ]
[ DISTRIBUTED BY (column, [ ... ] ) | DISTRIBUTED RANDOMLY ]
[ PARTITION BY partition_type (column)
  [ SUBPARTITION BY partition_type (column) ]
  [ SUBPARTITION TEMPLATE ( template_spec ) ]
  [...]
  ( partition_spec )
  | [ SUBPARTITION BY partition_type (column) ]
  [...]
  ( partition_spec
    [ ( subpartition_spec
      [ (...)]
    ) ]
  ) ]
)

```

where *storage_parameter* is:

```

APPENDONLY={TRUE|FALSE}
ORIENTATION={COLUMN|ROW}
COMPRESSTYPE={ZLIB|QUICKLZ}
COMPRESSLEVEL={0-9 | 1}
FILLFACTOR={10-100}
OIDS [=TRUE|FALSE]

```

where *column_constraint* is:

```

[CONSTRAINT constraint_name]
NOT NULL | NULL
| UNIQUE [USING INDEX TABLESPACE tablespace]
          [WITH ( FILLFACTOR = value )]
| PRIMARY KEY [USING INDEX TABLESPACE tablespace]
              [WITH ( FILLFACTOR = value )]
| CHECK ( expression )

```

and *table_constraint* is:

```

[CONSTRAINT constraint_name]
UNIQUE ( column_name [, ... ] )
        [USING INDEX TABLESPACE tablespace]
        [WITH ( FILLFACTOR=value )]
| PRIMARY KEY ( column_name [, ... ] )
              [USING INDEX TABLESPACE tablespace]
              [WITH ( FILLFACTOR=value )]
| CHECK ( expression )

```

where *partition_type* is:

```

LIST
| RANGE

```

where *partition_specification* is:

```

partition_element [, ...]

```

and *partition_element* is:

```

DEFAULT PARTITION name
| [PARTITION name] VALUES (list_value [,... ] )
| [PARTITION name]
  START ([datatype] 'start_value') [INCLUSIVE | EXCLUSIVE]
  [ END ([datatype] 'end_value') [INCLUSIVE | EXCLUSIVE] ]
  [ EVERY ([datatype] [number | INTERVAL] 'interval_value') ]
| [PARTITION name]
  END ([datatype] 'end_value') [INCLUSIVE | EXCLUSIVE]
  [ EVERY ([datatype] [number | INTERVAL] 'interval_value') ]
[ WITH ( partition_storage_parameter=value [, ... ] ) ]
[ TABLESPACE tablespace ]

```

where *subpartition_spec* or *template_spec* is:

subpartition_element [, ...]

and *subpartition_element* is:

```

DEFAULT SUBPARTITION name
| [SUBPARTITION name] VALUES (list_value [,... ] )
| [SUBPARTITION name]
  START ([datatype] 'start_value') [INCLUSIVE | EXCLUSIVE]
  [ END ([datatype] 'end_value') [INCLUSIVE | EXCLUSIVE] ]
  [ EVERY ([datatype] [number | INTERVAL] 'interval_value') ]
| [SUBPARTITION name]
  END ([datatype] 'end_value') [INCLUSIVE | EXCLUSIVE]
  [ EVERY ([datatype] [number | INTERVAL] 'interval_value') ]
[ WITH ( partition_storage_parameter=value [, ... ] ) ]
[ TABLESPACE tablespace ]

```

CREATE TABLE AS

Defines a new table from the results of a query.

```

CREATE [ [GLOBAL | LOCAL] {TEMPORARY | TEMP} ] TABLE table_name
  [(column_name [, ... ] )]
  [ WITH ( storage_parameter=value [, ... ] ) ]
  [ON COMMIT {PRESERVE ROWS | DELETE ROWS | DROP}]
  [TABLESPACE tablespace]
  AS query
  [DISTRIBUTED BY (column, [ ... ] ) | DISTRIBUTED RANDOMLY]

```

where *storage_parameter* is:

```

APPENDONLY={TRUE|FALSE}
ORIENTATION={COLUMN|ROW}
COMPRESSTYPE={ZLIB|QUICKLZ}
COMPRESSLEVEL={0-9 | 1}
FILLFACTOR={10-100}
oids [=TRUE|FALSE]

```

CREATE TABLESPACE

Defines a new tablespace.

```

CREATE TABLESPACE tablespacename [OWNER username]
  LOCATION 'segcontent0dir', 'segcontent1dir', ...
  [MIRROR LOCATION 'segcontent0dir', 'segcontent1dir', ...]
  [MASTER LOCATION 'mastercontentdir']

```

CREATE TRIGGER

Defines a new trigger.

```
CREATE TRIGGER name {BEFORE | AFTER} {event [OR ...]}
    ON table [ FOR [EACH] {ROW | STATEMENT} ]
    EXECUTE PROCEDURE funcname ( arguments )
```

CREATE TYPE

Defines a new data type.

```
CREATE TYPE name AS ( attribute_name data_type [, ... ] )
CREATE TYPE name (
    INPUT = input_function,
    OUTPUT = output_function
    [, RECEIVE = receive_function]
    [, SEND = send_function]
    [, ANALYZE = analyze_function]
    [, INTERNALLENGTH = {internallength | VARIABLE}]
    [, PASSEDBYVALUE]
    [, ALIGNMENT = alignment]
    [, STORAGE = storage]
    [, DEFAULT = default]
    [, ELEMENT = element]
    [, DELIMITER = delimiter]
)
CREATE TYPE name
```

CREATE USER

Defines a new database role with the LOGIN privilege by default.

```
CREATE USER name [ [WITH] option [ ... ] ]
```

where *option* can be:

```
    SUPERUSER | NOSUPERUSER
    | CREATEDB | NOCREATEDB
    | CREATEROLE | NOCREATEROLE
    | CREATEUSER | NOCREATEUSER
    | INHERIT | NOINHERIT
    | LOGIN | NOLOGIN
    | [ ENCRYPTED | UNENCRYPTED ] PASSWORD 'password'
    | VALID UNTIL 'timestamp'
    | IN ROLE rolename [, ...]
    | IN GROUP rolename [, ...]
    | ROLE rolename [, ...]
    | ADMIN rolename [, ...]
    | USER rolename [, ...]
    | SYSID uid
```

CREATE VIEW

Defines a new view.

```
CREATE [OR REPLACE] [TEMP | TEMPORARY] VIEW name
    [ ( column_name [, ...] ) ]
    AS query
```

DEALLOCATE

Deallocates a prepared statement.

```
DEALLOCATE [PREPARE] name
```

DECLARE

Defines a cursor.

```
DECLARE name [BINARY] [INSENSITIVE] [NO SCROLL] CURSOR
        [{WITH | WITHOUT} HOLD]
        FOR query [FOR READ ONLY]
```

DELETE

Deletes rows from a table.

```
DELETE FROM [ONLY] table [[AS] alias]
        [USING usinglist]
        [WHERE condition]
```

DROP AGGREGATE

Removes an aggregate function.

```
DROP AGGREGATE [IF EXISTS] name ( type [, ...] ) [CASCADE | RESTRICT]
```

DROP CAST

Removes a cast.

```
DROP CAST [IF EXISTS] (sourcetype AS targettype) [CASCADE | RESTRICT]
```

DROP CONVERSION

Removes a conversion.

```
DROP CONVERSION [IF EXISTS] name [CASCADE | RESTRICT]
```

DROP DATABASE

Removes a database.

```
DROP DATABASE [IF EXISTS] name
```

DROP DOMAIN

Removes a domain.

```
DROP DOMAIN [IF EXISTS] name [, ...] [CASCADE | RESTRICT]
```

DROP EXTERNAL TABLE

Removes an external or web table definition.

```
DROP EXTERNAL [WEB] TABLE [IF EXISTS] name [CASCADE | RESTRICT]
```

DROP FUNCTION

Removes a function.

```
DROP FUNCTION [IF EXISTS] name ( [ argmode] [argname] argtype [, ...] ) [CASCADE
| RESTRICT]
```

DROP GROUP

Removes a database role.

```
DROP GROUP [IF EXISTS] name [, ...]
```

DROP INDEX

Removes an index.

```
DROP INDEX [IF EXISTS] name [, ...] [CASCADE | RESTRICT]
```

DROP LANGUAGE

Removes a procedural language.

```
DROP [PROCEDURAL] LANGUAGE [IF EXISTS] name [CASCADE | RESTRICT]
```

DROP OPERATOR

Removes an operator.

```
DROP OPERATOR [IF EXISTS] name ( {lefttype | NONE} , {righttype | NONE} ) [CASCADE | RESTRICT]
```

DROP OPERATOR CLASS

Removes an operator class.

```
DROP OPERATOR CLASS [IF EXISTS] name USING index_method [CASCADE | RESTRICT]
```

DROP OWNED

Removes database objects owned by a database role.

```
DROP OWNED BY name [, ...] [CASCADE | RESTRICT]
```

DROP RESOURCE QUEUE

Removes a resource queue.

```
DROP RESOURCE QUEUE queue_name
```

DROP ROLE

Removes a database role.

```
DROP ROLE [IF EXISTS] name [, ...]
```

DROP RULE

Removes a rewrite rule.

```
DROP RULE [IF EXISTS] name ON relation [CASCADE | RESTRICT]
```

DROP SCHEMA

Removes a schema.

```
DROP SCHEMA [IF EXISTS] name [, ...] [CASCADE | RESTRICT]
```

DROP SEQUENCE

Removes a sequence.

```
DROP SEQUENCE [IF EXISTS] name [, ...] [CASCADE | RESTRICT]
```

DROP TABLE

Removes a table.

```
DROP TABLE [IF EXISTS] name [, ...] [CASCADE | RESTRICT]
```

DROP TABLESPACE

Removes a tablespace.

```
DROP TABLESPACE [IF EXISTS] tablespacename
```

DROP TRIGGER

Removes a trigger.

```
DROP TRIGGER [IF EXISTS] name ON table [CASCADE | RESTRICT]
```

DROP TYPE

Removes a data type.

```
DROP TYPE [IF EXISTS] name [, ...] [CASCADE | RESTRICT]
```

DROP USER

Removes a database role.

```
DROP USER [IF EXISTS] name [, ...]
```

DROP VIEW

Removes a view.

```
DROP VIEW [IF EXISTS] name [, ...] [CASCADE | RESTRICT]
```

END

Commits the current transaction.

```
END [WORK | TRANSACTION]
```

EXECUTE

Executes a prepared SQL statement.

```
EXECUTE name [ (parameter [, ...] ) ]
```

EXPLAIN

Shows the query plan of a statement.

```
EXPLAIN [ANALYZE] [VERBOSE] statement
```

FETCH

Retrieves rows from a query using a cursor.

```
FETCH [ forward_direction { FROM | IN } ] cursorname
```

where forward_direction can be empty or one of:

```

NEXT
FIRST
LAST
ABSOLUTE count
RELATIVE count
count
ALL
FORWARD
FORWARD count
FORWARD ALL
```

GRANT

Defines access privileges.

```
GRANT { {SELECT | INSERT | UPDATE | DELETE | REFERENCES | TRIGGER} [, ...] | ALL [PRIV-
ILEGES] }
    ON [TABLE] tablename [, ...]
    TO {rolename | PUBLIC} [, ...] [WITH GRANT OPTION]

GRANT { {USAGE | SELECT | UPDATE} [, ...] | ALL [PRIVILEGES] }
    ON SEQUENCE sequencename [, ...]
    TO { rolename | PUBLIC } [, ...] [WITH GRANT OPTION]

GRANT { {CREATE | CONNECT | TEMPORARY | TEMP} [, ...] | ALL [PRIVILEGES] }
    ON DATABASE dbname [, ...]
    TO {rolename | PUBLIC} [, ...] [WITH GRANT OPTION]

GRANT { EXECUTE | ALL [PRIVILEGES] }
    ON FUNCTION funcname ( [ [argmode] [argname] argtype [, ...] ] ) [, ...]
    TO {rolename | PUBLIC} [, ...] [WITH GRANT OPTION]

GRANT { USAGE | ALL [PRIVILEGES] }
    ON LANGUAGE langname [, ...]
    TO {rolename | PUBLIC} [, ...] [WITH GRANT OPTION]

GRANT { {CREATE | USAGE} [, ...] | ALL [PRIVILEGES] }
    ON SCHEMA schemaname [, ...]
    TO {rolename | PUBLIC} [, ...] [WITH GRANT OPTION]

GRANT { CREATE | ALL [PRIVILEGES] }
    ON TABLESPACE tablespacename [, ...]
    TO {rolename | PUBLIC} [, ...] [WITH GRANT OPTION]

GRANT parent_role [, ...]
    TO member_role [, ...] [WITH ADMIN OPTION]
```

INSERT

Creates new rows in a table.

```
INSERT INTO table [( column [, ...] )]
    {DEFAULT VALUES | VALUES ( {expression | DEFAULT} [, ...] ) [, ...] | query}
```

LOAD

Loads or reloads a shared library file.

```
LOAD 'filename'
```

LOCK

Locks a table.

```
LOCK [TABLE] name [, ...] [IN lockmode MODE] [NOWAIT]
```

where *lockmode* is one of:

```
ACCESS SHARE | ROW SHARE | ROW EXCLUSIVE | SHARE UPDATE EXCLUSIVE | SHARE | SHARE ROW
EXCLUSIVE | EXCLUSIVE | ACCESS EXCLUSIVE
```

MOVE

Positions a cursor.

```
MOVE [ forward_direction {FROM | IN} ] cursorname
```

where *direction* can be empty or one of:

```
NEXT
FIRST
LAST
ABSOLUTE count
RELATIVE count
count
ALL
FORWARD
FORWARD count
FORWARD ALL
```

PREPARE

Prepare a statement for execution.

```
PREPARE name [ (datatype [, ...] ) ] AS statement
```

REASSIGN OWNED

Changes the ownership of database objects owned by a database role.

```
REASSIGN OWNED BY old_role [, ...] TO new_role
```

REINDEX

Rebuilds indexes.

```
REINDEX {INDEX | TABLE | DATABASE | SYSTEM} name
```

RELEASE SAVEPOINT

Destroys a previously defined savepoint.

```
RELEASE [SAVEPOINT] savepoint_name
```

RESET

Restores the value of a system configuration parameter to the default value.

```
RESET configuration_parameter
```

```
RESET ALL
```

REVOKE

Removes access privileges.

```

REVOKE [GRANT OPTION FOR] { {SELECT | INSERT | UPDATE | DELETE
| REFERENCES | TRIGGER} [,...] | ALL [PRIVILEGES] }
ON [TABLE] tablename [, ...]
FROM {rolename | PUBLIC} [, ...]
[CASCADE | RESTRICT]

REVOKE [GRANT OPTION FOR] { {USAGE | SELECT | UPDATE} [,...]
| ALL [PRIVILEGES] }
ON SEQUENCE sequencename [, ...]
FROM { rolename | PUBLIC } [, ...]
[CASCADE | RESTRICT]

REVOKE [GRANT OPTION FOR] { {CREATE | CONNECT
| TEMPORARY | TEMP} [,...] | ALL [PRIVILEGES] }
ON DATABASE dbname [, ...]
FROM {rolename | PUBLIC} [, ...]
[CASCADE | RESTRICT]

REVOKE [GRANT OPTION FOR] {EXECUTE | ALL [PRIVILEGES]}
ON FUNCTION funcname ( [[argmode] [argname] argtype
[, ...]] ) [, ...]
FROM {rolename | PUBLIC} [, ...]
[CASCADE | RESTRICT]

REVOKE [GRANT OPTION FOR] {USAGE | ALL [PRIVILEGES]}
ON LANGUAGE langname [, ...]
FROM {rolename | PUBLIC} [, ...]
[ CASCADE | RESTRICT ]

REVOKE [GRANT OPTION FOR] { {CREATE | USAGE} [,...]
| ALL [PRIVILEGES] }
ON SCHEMA schemaname [, ...]
FROM {rolename | PUBLIC} [, ...]
[CASCADE | RESTRICT]

REVOKE [GRANT OPTION FOR] { CREATE | ALL [PRIVILEGES] }
ON TABLESPACE tablespacename [, ...]
FROM { rolename | PUBLIC } [, ...]
[CASCADE | RESTRICT]

REVOKE [ADMIN OPTION FOR] parent_role [, ...]
FROM member_role [, ...]
[CASCADE | RESTRICT]

```

ROLLBACK

Aborts the current transaction.

```
ROLLBACK [WORK | TRANSACTION]
```

ROLLBACK TO SAVEPOINT

Rolls back the current transaction to a savepoint.

```
ROLLBACK [WORK | TRANSACTION] TO [SAVEPOINT] savepoint_name
```

SAVEPOINT

Defines a new savepoint within the current transaction.

```
SAVEPOINT savepoint_name
```

SELECT

Retrieves rows from a table or view.

```
SELECT [ALL | DISTINCT [ON (expression [, ...])]]
* | expression [[AS] output_name] [, ...]
[FROM from_item [, ...]]
[WHERE condition]
[GROUP BY grouping_element [, ...]]
[HAVING condition [, ...]]
[WINDOW window_name AS (window_specification)]
[{UNION | INTERSECT | EXCEPT} [ALL] select]
[ORDER BY expression [ASC | DESC | USING operator] [, ...]]
[LIMIT {count | ALL}]
[OFFSET start]
[FOR {UPDATE | SHARE} [OF table_name [, ...]] [NOWAIT] [...]]
```

where *grouping_element* can be one of:

```
()
expression
ROLLUP (expression [,...])
CUBE (expression [,...])
GROUPING SETS ((grouping_element [, ...]))
```

where *window_specification* can be:

```
[window_name]
[PARTITION BY expression [, ...]]
[ORDER BY expression [ASC | DESC | USING operator] [, ...]
  [{RANGE | ROWS}
   { UNBOUNDED PRECEDING
     | expression PRECEDING
     | CURRENT ROW
     | BETWEEN window_frame_bound AND window_frame_bound }]]
```

where *window_frame_bound* can be one of:

```
UNBOUNDED PRECEDING
expression PRECEDING
CURRENT ROW
expression FOLLOWING
UNBOUNDED FOLLOWING
```

where *from_item* can be one of:

```
[ONLY] table_name [[AS] alias [( column_alias [, ...] )]]
(select) [AS] alias [( column_alias [, ...] )]
function_name ( [argument [, ...]] ) [AS] alias
  [( column_alias [, ...]
    | column_definition [, ...] )]
function_name ( [argument [, ...]] ) AS
  ( column_definition [, ...] )
from_item [NATURAL] join_type from_item
  [ON join_condition | USING ( join_column [, ...] )]
```

SELECT INTO

Defines a new table from the results of a query.

```
SELECT [ALL | DISTINCT [ON ( expression [, ...] )]]
      * | expression [AS output_name] [, ...]
INTO [TEMPORARY | TEMP] [TABLE] new_table
[FROM from_item [, ...]]
[WHERE condition]
[GROUP BY expression [, ...]]
[HAVING condition [, ...]]
[{UNION | INTERSECT | EXCEPT} [ALL] select]
[ORDER BY expression [ASC | DESC | USING operator] [, ...]]
[LIMIT {count | ALL}]
[OFFSET start]
[FOR {UPDATE | SHARE} [OF table_name [, ...]] [NOWAIT] [...]]
```

SET

Changes the value of a Greenplum Database configuration parameter.

```
SET [SESSION | LOCAL] configuration_parameter {TO | =} value | 'value' | DEFAULT}
SET [SESSION | LOCAL] TIME ZONE {timezone | LOCAL | DEFAULT}
```

SET ROLE

Sets the current role identifier of the current session.

```
SET [SESSION | LOCAL] ROLE rolename
SET [SESSION | LOCAL] ROLE NONE
RESET ROLE
```

SET SESSION AUTHORIZATION

Sets the session role identifier and the current role identifier of the current session.

```
SET [SESSION | LOCAL] SESSION AUTHORIZATION rolename
SET [SESSION | LOCAL] SESSION AUTHORIZATION DEFAULT
RESET SESSION AUTHORIZATION
```

SET TRANSACTION

Sets the characteristics of the current transaction.

```
SET TRANSACTION transaction_mode [, ...]
SET SESSION CHARACTERISTICS AS TRANSACTION transaction_mode [, ...]
```

where *transaction_mode* is one of:

```
ISOLATION LEVEL {SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ UNCOMMITTED}
READ WRITE | READ ONLY
```

SHOW

Shows the value of a system configuration parameter.

```
SHOW configuration_parameter
SHOW ALL
```

START TRANSACTION

Starts a transaction block.

```
START TRANSACTION [SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ UNCOMMITTED]
[READ WRITE | READ ONLY]
```

TRUNCATE

Empties a table of all rows.

```
TRUNCATE [TABLE] name [, ...] [CASCADE | RESTRICT]
```

UPDATE

Updates rows of a table.

```
UPDATE [ONLY] table [[AS] alias]
  SET {column = {expression | DEFAULT} |
      (column [, ...] = ({expression | DEFAULT} [, ...]))} [, ...]
  [FROM fromlist]
  [WHERE condition]
```

VACUUM

Garbage-collects and optionally analyzes a database.

```
VACUUM [FULL] [FREEZE] [VERBOSE] [table]
VACUUM [FULL] [FREEZE] [VERBOSE] ANALYZE
      [table [(column [, ...] )]]
```

VALUES

Computes a set of rows.

```
VALUES ( expression [, ...] ) [, ...]
[ORDER BY sort_expression [ASC | DESC | USING operator] [, ...]]
[LIMIT {count | ALL}] [OFFSET start]
```

ABORT

Aborts the current transaction.

Synopsis

```
ABORT [WORK | TRANSACTION]
```

Description

`ABORT` rolls back the current transaction and causes all the updates made by the transaction to be discarded. This command is identical in behavior to the standard SQL command `ROLLBACK`, and is present only for historical reasons.

Parameters

WORK
TRANSACTION

Optional key words. They have no effect.

Notes

Use `COMMIT` to successfully terminate a transaction.

Issuing `ABORT` when not inside a transaction does no harm, but it will provoke a warning message.

Compatibility

This command is a Greenplum Database extension present for historical reasons. `ROLLBACK` is the equivalent standard SQL command.

See Also

[BEGIN](#), [COMMIT](#), [ROLLBACK](#)

ALTER AGGREGATE

Changes the definition of an aggregate function

Synopsis

```
ALTER AGGREGATE name ( type [ , ... ] ) RENAME TO new_name
ALTER AGGREGATE name ( type [ , ... ] ) OWNER TO new_owner
ALTER AGGREGATE name ( type [ , ... ] ) SET SCHEMA new_schema
```

Description

`ALTER AGGREGATE` changes the definition of an aggregate function.

You must own the aggregate function to use `ALTER AGGREGATE`. To change the schema of an aggregate function, you must also have `CREATE` privilege on the new schema. To alter the owner, you must also be a direct or indirect member of the new owning role, and that role must have `CREATE` privilege on the aggregate function's schema. (These restrictions enforce that altering the owner does not do anything you could not do by dropping and recreating the aggregate function. However, a superuser can alter ownership of any aggregate function anyway.)

Parameters

name

The name (optionally schema-qualified) of an existing aggregate function.

type

An input data type on which the aggregate function operates. To reference a zero-argument aggregate function, write `*` in place of the list of input data types.

new_name

The new name of the aggregate function.

new_owner

The new owner of the aggregate function.

new_schema

The new schema for the aggregate function.

Examples

To rename the aggregate function *myavg* for type *integer* to *my_average*:

```
ALTER AGGREGATE myavg(integer) RENAME TO my_average;
```

To change the owner of the aggregate function *myavg* for type *integer* to *joe*:

```
ALTER AGGREGATE myavg(integer) OWNER TO joe;
```

To move the aggregate function *myavg* for type *integer* into schema *myschema*:

```
ALTER AGGREGATE myavg(integer) SET SCHEMA myschema;
```

Compatibility

There is no `ALTER AGGREGATE` statement in the SQL standard.

See Also

[CREATE AGGREGATE](#), [DROP AGGREGATE](#)

ALTER CONVERSION

Changes the definition of a conversion.

Synopsis

```
ALTER CONVERSION name RENAME TO newname
```

```
ALTER CONVERSION name OWNER TO newowner
```

Description

ALTER CONVERSION changes the definition of a conversion.

You must own the conversion to use ALTER CONVERSION. To alter the owner, you must also be a direct or indirect member of the new owning role, and that role must have CREATE privilege on the conversion's schema. (These restrictions enforce that altering the owner does not do anything you could not do by dropping and recreating the conversion. However, a superuser can alter ownership of any conversion anyway.)

Parameters

name

The name (optionally schema-qualified) of an existing conversion.

newname

The new name of the conversion.

newowner

The new owner of the conversion.

Examples

To rename the conversion *iso_8859_1_to_utf8* to *latin1_to_unicode*:

```
ALTER CONVERSION iso_8859_1_to_utf8 RENAME TO
latin1_to_unicode;
```

To change the owner of the conversion *iso_8859_1_to_utf8* to *joe*:

```
ALTER CONVERSION iso_8859_1_to_utf8 OWNER TO joe;
```

Compatibility

There is no ALTER CONVERSION statement in the SQL standard.

See Also

CREATE CONVERSION, DROP CONVERSION

ALTER DATABASE

Changes the attributes of a database.

Synopsis

```
ALTER DATABASE name [ WITH CONNECTION LIMIT connlimit ]
ALTER DATABASE name SET parameter { TO | = } { value | DEFAULT }
ALTER DATABASE name RESET parameter
ALTER DATABASE name RENAME TO newname
ALTER DATABASE name OWNER TO new_owner
```

Description

ALTER DATABASE changes the attributes of a database.

The first form changes the allowed connection limit for a database. Only the database owner or a superuser can change this setting.

The second and third forms change the session default for a configuration parameter for a Greenplum database. Whenever a new session is subsequently started in that database, the specified value becomes the session default value. The database-specific default overrides whatever setting is present in the server configuration file (`postgresql.conf`). Only the database owner or a superuser can change the session defaults for a database. Certain parameters cannot be set this way, or can only be set by a superuser.

The fourth form changes the name of the database. Only the database owner or a superuser can rename a database; non-superuser owners must also have the `CREATEDB` privilege. You cannot rename the current database. Connect to a different database first.

The fifth form changes the owner of the database. To alter the owner, you must own the database and also be a direct or indirect member of the new owning role, and you must have the `CREATEDB` privilege. (Note that superusers have all these privileges automatically.)

Parameters

name

The name of the database whose attributes are to be altered.

connlimit

Warning: Setting a connection limit at the database level cannot be enforced in Greenplum Database and may cause queries to fail. Leave this set at the default of -1 (no limit). Connection limits may only be set at the system level in Greenplum Database. See [“Limiting Concurrent Connections”](#) on page 75 for more information.

***parameter
value***

Set this database's session default for the specified configuration parameter to the given value. If value is `DEFAULT` or, equivalently, `RESET` is used, the database-specific setting is removed, so the system-wide default setting will be inherited in new sessions. Use `RESET ALL` to clear all database-specific settings. See “[Server Configuration Parameters](#)” on page 755 for information about all user-settable configuration parameters.

newname

The new name of the database.

new_owner

The new owner of the database.

Notes

It is also possible to set a configuration parameter session default for a specific role (user) rather than to a database. Role-specific settings override database-specific ones if there is a conflict. See `ALTER ROLE`.

Examples

To set the default schema search path for the *mydatabase* database:

```
ALTER DATABASE mydatabase SET search_path TO myschema,  
public, pg_catalog;
```

Compatibility

The `ALTER DATABASE` statement is a Greenplum Database extension.

See Also

[CREATE DATABASE](#), [DROP DATABASE](#), [SET](#)

ALTER DOMAIN

Changes the definition of a domain.

Synopsis

```
ALTER DOMAIN name { SET DEFAULT expression | DROP DEFAULT }
ALTER DOMAIN name { SET | DROP } NOT NULL
ALTER DOMAIN name ADD domain_constraint
ALTER DOMAIN name DROP CONSTRAINT constraint_name [RESTRICT |
CASCADE]
ALTER DOMAIN name OWNER TO new_owner
ALTER DOMAIN name SET SCHEMA new_schema
```

Description

`ALTER DOMAIN` changes the definition of an existing domain. There are several sub-forms:

- **SET/DROP DEFAULT** — These forms set or remove the default value for a domain. Note that defaults only apply to subsequent `INSERT` commands. They do not affect rows already in a table using the domain.
- **SET/DROP NOT NULL** — These forms change whether a domain is marked to allow `NULL` values or to reject `NULL` values. You may only `SET NOT NULL` when the columns using the domain contain no null values.
- **ADD domain_constraint** — This form adds a new constraint to a domain using the same syntax as `CREATE DOMAIN`. This will only succeed if all columns using the domain satisfy the new constraint.
- **DROP CONSTRAINT** — This form drops constraints on a domain.
- **OWNER** — This form changes the owner of the domain to the specified user.
- **SET SCHEMA** — This form changes the schema of the domain. Any constraints associated with the domain are moved into the new schema as well.

You must own the domain to use `ALTER DOMAIN`. To change the schema of a domain, you must also have `CREATE` privilege on the new schema. To alter the owner, you must also be a direct or indirect member of the new owning role, and that role must have `CREATE` privilege on the domain's schema. (These restrictions enforce that altering the owner does not do anything you could not do by dropping and recreating the domain. However, a superuser can alter ownership of any domain anyway.)

Parameters

name

The name (optionally schema-qualified) of an existing domain to alter.

domain_constraint

New domain constraint for the domain.

constraint_name

Name of an existing constraint to drop.

CASCADE

Automatically drop objects that depend on the constraint.

RESTRICT

Refuse to drop the constraint if there are any dependent objects. This is the default behavior.

new_owner

The user name of the new owner of the domain.

new_schema

The new schema for the domain.

Examples

To add a NOT NULL constraint to a domain:

```
ALTER DOMAIN zipcode SET NOT NULL;
```

To remove a NOT NULL constraint from a domain:

```
ALTER DOMAIN zipcode DROP NOT NULL;
```

To add a check constraint to a domain:

```
ALTER DOMAIN zipcode ADD CONSTRAINT zipchk CHECK
(char_length(VALUE) = 5);
```

To remove a check constraint from a domain:

```
ALTER DOMAIN zipcode DROP CONSTRAINT zipchk;
```

To move the domain into a different schema:

```
ALTER DOMAIN zipcode SET SCHEMA customers;
```

Compatibility

ALTER DOMAIN conforms to the SQL standard, except for the OWNER and SET SCHEMA variants, which are Greenplum Database extensions.

See Also

[CREATE DOMAIN](#), [DROP DOMAIN](#)

ALTER FUNCTION

Changes the definition of a function.

Synopsis

```
ALTER FUNCTION name ( [ [argmode] [argname] argtype [, ...] ] )
action [, ... ] [RESTRICT]
```

```
ALTER FUNCTION name ( [ [argmode] [argname] argtype [, ...] ] )
RENAME TO new_name
```

```
ALTER FUNCTION name ( [ [argmode] [argname] argtype [, ...] ] )
OWNER TO new_owner
```

```
ALTER FUNCTION name ( [ [argmode] [argname] argtype [, ...] ] )
SET SCHEMA new_schema
```

where *action* is one of:

```
{CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT}
{IMMUTABLE | STABLE | VOLATILE}
{[EXTERNAL] SECURITY INVOKER | [EXTERNAL] SECURITY DEFINER}
```

Description

ALTER FUNCTION changes the definition of a function.

You must own the function to use ALTER FUNCTION. To change a function's schema, you must also have CREATE privilege on the new schema. To alter the owner, you must also be a direct or indirect member of the new owning role, and that role must have CREATE privilege on the function's schema. (These restrictions enforce that altering the owner does not do anything you could not do by dropping and recreating the function. However, a superuser can alter ownership of any function anyway.)

Parameters

name

The name (optionally schema-qualified) of an existing function.

argmode

The mode of an argument: either IN, OUT, or INOUT. If omitted, the default is IN.

Note that ALTER FUNCTION does not actually pay any attention to OUT arguments, since only the input arguments are needed to determine the function's identity. So it is sufficient to list the IN and INOUT arguments.

argname

The name of an argument. Note that ALTER FUNCTION does not actually pay any attention to argument names, since only the argument data types are needed to determine the function's identity.

argtype

The data type(s) of the function's arguments (optionally schema-qualified), if any.

new_name

The new name of the function.

new_owner

The new owner of the function. Note that if the function is marked `SECURITY DEFINER`, it will subsequently execute as the new owner.

new_schema

The new schema for the function.

CALLED ON NULL INPUT
RETURNS NULL ON NULL INPUT
STRICT

`CALLED ON NULL INPUT` changes the function so that it will be invoked when some or all of its arguments are null. `RETURNS NULL ON NULL INPUT` or `STRICT` changes the function so that it is not invoked if any of its arguments are null; instead, a null result is assumed automatically. See `CREATE FUNCTION` for more information.

IMMUTABLE
STABLE
VOLATILE

Change the volatility of the function to the specified setting. See `CREATE FUNCTION` for details.

[**EXTERNAL**] **SECURITY INVOKER**
 [**EXTERNAL**] **SECURITY DEFINER**

Change whether the function is a security definer or not. The key word `EXTERNAL` is ignored for SQL conformance. See `CREATE FUNCTION` for more information about this capability.

RESTRICT

Ignored for conformance with the SQL standard.

Notes

Greenplum Database has limitations on the use of functions defined as `STABLE` or `VOLATILE`. See `CREATE FUNCTION` for more information.

Examples

To rename the function `sqrt` for type `integer` to `square_root`:

```
ALTER FUNCTION sqrt(integer) RENAME TO square_root;
```

To change the owner of the function `sqrt` for type `integer` to `joe`:

```
ALTER FUNCTION sqrt(integer) OWNER TO joe;
```

To change the *schema* of the function *sqrt* for type *integer* to *math*:

```
ALTER FUNCTION sqrt(integer) SET SCHEMA math;
```

Compatibility

This statement is partially compatible with the `ALTER FUNCTION` statement in the SQL standard. The standard allows more properties of a function to be modified, but does not provide the ability to rename a function, make a function a security definer, or change the owner, schema, or volatility of a function. The standard also requires the `RESTRICT` key word, which is optional in Greenplum Database.

See Also

[CREATE FUNCTION](#), [DROP FUNCTION](#)

ALTER GROUP

Changes a role name or membership.

Synopsis

```
ALTER GROUP groupname ADD USER username [, ... ]
```

```
ALTER GROUP groupname DROP USER username [, ... ]
```

```
ALTER GROUP groupname RENAME TO newname
```

Description

`ALTER GROUP` is an obsolete command, though still accepted for backwards compatibility. Groups (and users) have been superseded by the more general concept of roles. See [ALTER ROLE](#) for more information.

Parameters

groupname

The name of the group (role) to modify.

username

Users (roles) that are to be added to or removed from the group. The users (roles) must already exist.

newname

The new name of the group (role).

Examples

To add users to a group:

```
ALTER GROUP staff ADD USER karl, john;
```

To remove a user from a group:

```
ALTER GROUP workers DROP USER beth;
```

Compatibility

There is no `ALTER GROUP` statement in the SQL standard.

See Also

[ALTER ROLE](#), [GRANT](#), [REVOKE](#)

ALTER INDEX

Changes the definition of an index.

Synopsis

```
ALTER INDEX name RENAME TO new_name
ALTER INDEX name SET TABLESPACE tablespace_name
ALTER INDEX name SET ( FILLFACTOR = value )
ALTER INDEX name RESET ( FILLFACTOR )
```

Description

ALTER INDEX changes the definition of an existing index. There are several subforms:

- **RENAME** — Changes the name of the index. There is no effect on the stored data.
- **SET TABLESPACE** — Changes the index's tablespace to the specified tablespace and moves the data file(s) associated with the index to the new tablespace. See also CREATE TABLESPACE.
- **SET FILLFACTOR** — Changes the index-method-specific storage parameters for the index. The built-in index methods all accept a single parameter: FILLFACTOR. The fillfactor for an index is a percentage that determines how full the index method will try to pack index pages. Index contents will not be modified immediately by this command. Use REINDEX to rebuild the index to get the desired effects.
- **RESET FILLFACTOR** — Resets FILLFACTOR to the default. As with SET, a REINDEX may be needed to update the index entirely.

Parameters

name

The name (optionally schema-qualified) of an existing index to alter.

new_name

New name for the index.

tablespace_name

The tablespace to which the index will be moved.

FILLFACTOR

The fillfactor for an index is a percentage that determines how full the index method will try to pack index pages. For B-trees, leaf pages are filled to this percentage during initial index build, and also when extending the index at the right (largest key values). If pages subsequently become completely full, they will be split, leading to gradual degradation in the index's efficiency.

B-trees use a default fillfactor of 90, but any value from 10 to 100 can be selected. If the table is static then fillfactor 100 is best to minimize the index's physical size, but for heavily updated tables a smaller fillfactor is better to minimize the need for page splits. The other index methods use fillfactor in different but roughly analogous ways; the default fillfactor varies between methods.

Notes

These operations are also possible using `ALTER TABLE`.

Changing any part of a system catalog index is not permitted.

Examples

To rename an existing index:

```
ALTER INDEX distributors RENAME TO suppliers;
```

To move an index to a different tablespace:

```
ALTER INDEX distributors SET TABLESPACE fasttablespace;
```

To change an index's fill factor (assuming that the index method supports it):

```
ALTER INDEX distributors SET (fillfactor = 75);  
REINDEX INDEX distributors;
```

Compatibility

`ALTER INDEX` is a Greenplum Database extension.

See Also

[CREATE INDEX](#), [REINDEX](#), [ALTER TABLE](#)

ALTER LANGUAGE

Changes the name of a procedural language.

Synopsis

```
ALTER LANGUAGE name RENAME TO newname
```

Description

`ALTER LANGUAGE` changes the name of a procedural language. Only a superuser can rename languages.

Parameters

name

Name of a language.

newname

The new name of the language.

Compatibility

There is no `ALTER LANGUAGE` statement in the SQL standard.

See Also

[CREATE LANGUAGE](#), [DROP LANGUAGE](#)

ALTER OPERATOR

Changes the definition of an operator.

Synopsis

```
ALTER OPERATOR name ( {lefttype | NONE} , {righttype | NONE} )  
OWNER TO newowner
```

Description

`ALTER OPERATOR` changes the definition of an operator. The only currently available functionality is to change the owner of the operator.

You must own the operator to use `ALTER OPERATOR`. To alter the owner, you must also be a direct or indirect member of the new owning role, and that role must have `CREATE` privilege on the operator's schema. (These restrictions enforce that altering the owner does not do anything you could not do by dropping and recreating the operator. However, a superuser can alter ownership of any operator anyway.)

Parameters

name

The name (optionally schema-qualified) of an existing operator.

lefttype

The data type of the operator's left operand; write `NONE` if the operator has no left operand.

righttype

The data type of the operator's right operand; write `NONE` if the operator has no right operand.

newowner

The new owner of the operator.

Examples

Change the owner of a custom operator `a @@ b` for type `text`:

```
ALTER OPERATOR @@ (text, text) OWNER TO joe;
```

Compatibility

There is no `ALTER OPERATOR` statement in the SQL standard.

See Also

[CREATE OPERATOR](#), [DROP OPERATOR](#)

ALTER OPERATOR CLASS

Changes the definition of an operator class.

Synopsis

```
ALTER OPERATOR CLASS name USING index_method RENAME TO newname  
ALTER OPERATOR CLASS name USING index_method OWNER TO newowner
```

Description

ALTER OPERATOR CLASS changes the definition of an operator class.

You must own the operator class to use ALTER OPERATOR CLASS. To alter the owner, you must also be a direct or indirect member of the new owning role, and that role must have CREATE privilege on the operator class's schema. (These restrictions enforce that altering the owner does not do anything you could not do by dropping and recreating the operator class. However, a superuser can alter ownership of any operator class anyway.)

Parameters

name

The name (optionally schema-qualified) of an existing operator class.

index_method

The name of the index method this operator class is for.

newname

The new name of the operator class.

newowner

The new owner of the operator class

Compatibility

There is no ALTER OPERATOR CLASS statement in the SQL standard.

See Also

[CREATE OPERATOR CLASS](#), [DROP OPERATOR CLASS](#)

ALTER RESOURCE QUEUE

Changes the limits of a resource queue.

Synopsis

```
ALTER RESOURCE QUEUE name ACTIVE THRESHOLD integer
[COST THRESHOLD float [OVERCOMMIT | NOOVERCOMMIT]
                        [IGNORE THRESHOLD float]]

ALTER RESOURCE QUEUE name COST THRESHOLD float
[OVERCOMMIT | NOOVERCOMMIT] [IGNORE THRESHOLD float]
[ACTIVE THRESHOLD integer]
```

Description

`ALTER RESOURCE QUEUE` changes the limits of a resource queue. Only a superuser can alter a resource queue. A resource queue must have either an `ACTIVE THRESHOLD` or a `COST THRESHOLD` value (or it can have both). See [CREATE RESOURCE QUEUE](#) for more information.

Parameters

name

The name of the resource queue whose limits are to be altered.

ACTIVE THRESHOLD *integer*

The number of active statements submitted from users in this resource queue allowed on the system at any one time. The value for `ACTIVE THRESHOLD` should be an integer greater than 0. To reset an `ACTIVE THRESHOLD` to have no limit, enter a value of -1.

COST THRESHOLD *float*

The total query planner cost of statements submitted from users in this resource queue allowed on the system at any one time. The value for `COST THRESHOLD` is specified as a floating point number (for example 100.0) or can also be specified as an exponent (for example 1e+2). To reset a `COST THRESHOLD` to have no limit, enter a value of -1.0.

OVERCOMMIT | NOOVERCOMMIT

If a resource queue is limited based on a cost threshold, then the administrator can allow `OVERCOMMIT` (the default). This means that a query that exceeds the allowed cost threshold will be allowed to run but only when the queue is idle. If `NOOVERCOMMIT` is specified, queries that exceed the cost limit will always be rejected and never allowed to run.

IGNORE THRESHOLD *float*

Queries with a cost under this limit will not be queued and run immediately. Cost is measured in units of disk page fetches; 1.0 equals one sequential disk page read. The value for `IGNORE THRESHOLD` is specified as a floating point number (for example 100.0) or can also be specified as an exponent (for example 1e+2). To reset `IGNORE THRESHOLD` to have no limit, enter a value of -1.0.

Notes

Use `CREATE ROLE` or `ALTER ROLE` to add a role (user) to a resource queue.

Examples

Change the active query limit for a resource queue:

```
ALTER RESOURCE QUEUE myqueue ACTIVE THRESHOLD 20;
```

Reset the query and ignore cost limit for a resource queue to no limit:

```
ALTER RESOURCE QUEUE myqueue COST THRESHOLD -1.0 IGNORE
THRESHOLD -1.0;
```

Reset the query cost limit for a resource queue to 3¹⁰ (or 30000000000.0) and do not allow overcommit:

```
ALTER RESOURCE QUEUE myqueue COST THRESHOLD 3e+10
NOOVERCOMMIT;
```

Compatibility

The `ALTER RESOURCE QUEUE` statement is a Greenplum Database extension. This command does not exist in standard PostgreSQL.

See Also

`CREATE RESOURCE QUEUE`, `DROP RESOURCE QUEUE`, `CREATE ROLE`, `ALTER ROLE`

ALTER ROLE

Changes a database role (user or group).

Synopsis

```
ALTER ROLE name RENAME TO newname
ALTER ROLE name SET config_parameter {TO | =} {value | DEFAULT}
ALTER ROLE name RESET config_parameter
ALTER ROLE name RESOURCE QUEUE {queue_name | NONE}
ALTER ROLE name [ [WITH] option [ ... ] ]
```

where *option* can be:

```
    SUPERUSER | NOSUPERUSER
    | CREATEDB | NOCREATEDB
    | CREATEROLE | NOCREATEROLE
    | INHERIT | NOINHERIT
    | LOGIN | NOLOGIN
    | CONNECTION LIMIT connlimit
    | [ENCRYPTED | UNENCRYPTED] PASSWORD 'password'
    | VALID UNTIL 'timestamp'
```

Description

ALTER ROLE changes the attributes of a Greenplum Database role. There are several variants of this command:

- **RENAME** — Changes the name of the role. Database superusers can rename any role. Roles having CREATEROLE privilege can rename non-superuser roles. The current session user cannot be renamed (connect as a different user to rename a role). Because MD5-encrypted passwords use the role name as cryptographic salt, renaming a role clears its password if the password is MD5-encrypted.
- **SET | RESET** — changes a role’s session default for a specified configuration parameter. Whenever the role subsequently starts a new session, the specified value becomes the session default, overriding whatever setting is present in server configuration file (`postgresql.conf`). For a role without LOGIN privilege, session defaults have no effect. Ordinary roles can change their own session defaults. Superusers can change anyone’s session defaults. Roles having CREATEROLE privilege can change defaults for non-superuser roles. See “[Server Configuration Parameters](#)” on page 755 for more information on all user-settable configuration parameters.
- **RESOURCE QUEUE** — Assigns the role to a workload management resource queue. The role would then be subject to the limits assigned to the resource queue when issuing queries. Specify NONE to exempt the role from resource scheduling. A role can only belong to one resource queue. For a role without LOGIN privilege, resource queues have no effect. See [CREATE RESOURCE QUEUE](#) for more information.

- **WITH option** — Changes many of the role attributes that can be specified in `CREATE ROLE`. Attributes not mentioned in the command retain their previous settings. Database superusers can change any of these settings for any role. Roles having `CREATEROLE` privilege can change any of these settings, but only for non-superuser roles. Ordinary roles can only change their own password.

Parameters

name

The name of the role whose attributes are to be altered.

newname

The new name of the role.

config_parameter value

Set this role's session default for the specified configuration parameter to the given value. If value is `DEFAULT` or if `RESET` is used, the role-specific variable setting is removed, so the role will inherit the system-wide default setting in new sessions. Use `RESET ALL` to clear all role-specific settings. See `SET` and “[Server Configuration Parameters](#)” on page 755 for more information on user-settable configuration parameters.

queue_name

The name of the resource queue to which the user-level role is to be assigned. Only roles with `LOGIN` privilege can be assigned to a resource queue. To unassign a role from a resource queue, specify `NONE`. A role can only belong to one resource queue.

`SUPERUSER | NOSUPERUSER`
`CREATEDB | NOCREATEDB`
`CREATEROLE | NOCREATEROLE`
`INHERIT | NOINHERIT`
`LOGIN | NOLOGIN`
`CONNECTION LIMIT connlimit`
`PASSWORD password`
`ENCRYPTED | UNENCRYPTED`
`VALID UNTIL 'timestamp'`

These clauses alter role attributes originally set by `CREATE ROLE`.

Warning: Setting a connection limit at the role level cannot be enforced in Greenplum Database and may cause queries to fail. Leave this set at the default of -1 (no limit). Connection limits may only be set at the system level in Greenplum Database. See “[Limiting Concurrent Connections](#)” on page 75 for more information.

Notes

Use `GRANT` and `REVOKE` for adding and removing role memberships.

Caution must be exercised when specifying an unencrypted password with this command. The password will be transmitted to the server in clear text, and it might also be logged in the client's command history or the server log. The `psql` command-line client contains a meta-command `\password` that can be used to safely change a role's password.

It is also possible to tie a session default to a specific database rather than to a role. Role-specific settings override database-specific ones if there is a conflict. See [ALTER DATABASE](#).

Examples

Change a role's password:

```
ALTER ROLE daria WITH PASSWORD 'passwd123';
```

Change a password expiration date:

```
ALTER ROLE scott VALID UNTIL 'May 4 12:00:00 2015 +1';
```

Make a password valid forever:

```
ALTER ROLE luke VALID UNTIL 'infinity';
```

Give a role the ability to create other roles and new databases:

```
ALTER ROLE joelle CREATEROLE CREATEDB;
```

Give a role a non-default setting of the *maintenance_work_mem* parameter:

```
ALTER ROLE admin SET maintenance_work_mem = 100000;
```

Assign a role to a resource queue:

```
ALTER ROLE sammy RESOURCE QUEUE poweruser;
```

Compatibility

The `ALTER ROLE` statement is a Greenplum Database extension.

See Also

[CREATE ROLE](#), [DROP ROLE](#), [SET](#), [CREATE RESOURCE QUEUE](#), [GRANT](#), [REVOKE](#)

ALTER SCHEMA

Changes the definition of a schema.

Synopsis

```
ALTER SCHEMA name RENAME TO newname
```

```
ALTER SCHEMA name OWNER TO newowner
```

Description

ALTER SCHEMA changes the definition of a schema.

You must own the schema to use ALTER SCHEMA. To rename a schema you must also have the CREATE privilege for the database. To alter the owner, you must also be a direct or indirect member of the new owning role, and you must have the CREATE privilege for the database. Note that superusers have all these privileges automatically.

Parameters

name

The name of an existing schema.

newname

The new name of the schema. The new name cannot begin with pg_, as such names are reserved for system schemas.

newowner

The new owner of the schema.

Compatibility

There is no ALTER SCHEMA statement in the SQL standard.

See Also

[CREATE SCHEMA](#), [DROP SCHEMA](#)

ALTER SEQUENCE

Changes the definition of a sequence generator.

Synopsis

```
ALTER SEQUENCE name [INCREMENT [ BY ] increment]
    [MINVALUE minvalue | NO MINVALUE]
    [MAXVALUE maxvalue | NO MAXVALUE]
    [RESTART [ WITH ] start]
    [CACHE cache] [[ NO ] CYCLE]
    [OWNED BY {table.column | NONE}]

ALTER SEQUENCE name SET SCHEMA new_schema
```

Description

`ALTER SEQUENCE` changes the parameters of an existing sequence generator. Any parameters not specifically set in the `ALTER SEQUENCE` command retain their prior settings.

You must own the sequence to use `ALTER SEQUENCE`. To change a sequence's schema, you must also have `CREATE` privilege on the new schema. Note that superusers have all these privileges automatically.

Parameters

name

The name (optionally schema-qualified) of a sequence to be altered.

increment

The clause `INCREMENT BY increment` is optional. A positive value will make an ascending sequence, a negative one a descending sequence. If unspecified, the old increment value will be maintained.

minvalue NO MINVALUE

The optional clause `MINVALUE minvalue` determines the minimum value a sequence can generate. If `NO MINVALUE` is specified, the defaults of 1 and -263-1 for ascending and descending sequences, respectively, will be used. If neither option is specified, the current minimum value will be maintained.

maxvalue NO MAXVALUE

The optional clause `MAXVALUE maxvalue` determines the maximum value for the sequence. If `NO MAXVALUE` is specified, the defaults are 263-1 and -1 for ascending and descending sequences, respectively, will be used. If neither option is specified, the current maximum value will be maintained.

start

The optional clause `RESTART WITH start` changes the current value of the sequence.

cache

The clause `CACHE cache` enables sequence numbers to be preallocated and stored in memory for faster access. The minimum value is 1 (only one value can be generated at a time, i.e., no cache). If unspecified, the old cache value will be maintained.

CYCLE

The optional `CYCLE` key word may be used to enable the sequence to wrap around when the *maxvalue* or *minvalue* has been reached by an ascending or descending sequence. If the limit is reached, the next number generated will be the respective *minvalue* or *maxvalue*.

NO CYCLE

If the optional `NO CYCLE` key word is specified, any calls to `nextval` after the sequence has reached its maximum value will return an error. If neither `CYCLE` or `NO CYCLE` are specified, the old cycle behavior will be maintained.

OWNED BY *table.column***OWNED BY NONE**

The `OWNED BY` option causes the sequence to be associated with a specific table column, such that if that column (or its whole table) is dropped, the sequence will be automatically dropped as well. If specified, this association replaces any previously specified association for the sequence. The specified table must have the same owner and be in the same schema as the sequence. Specifying `OWNED BY NONE` removes any existing table column association.

new_schema

The new schema for the sequence.

Notes

To avoid blocking of concurrent transactions that obtain numbers from the same sequence, `ALTER SEQUENCE`'s effects on the sequence generation parameters are never rolled back; those changes take effect immediately and are not reversible. However, the `OWNED BY` and `SET SCHEMA` clauses are ordinary catalog updates and can be rolled back.

`ALTER SEQUENCE` will not immediately affect `nextval` results in sessions, other than the current one, that have preallocated (cached) sequence values. They will use up all cached values prior to noticing the changed sequence generation parameters. The current session will be affected immediately.

Some variants of `ALTER TABLE` can be used with sequences as well. For example, to rename a sequence use `ALTER TABLE RENAME`.

Examples

Restart a sequence called *serial*, at 105:

```
ALTER SEQUENCE serial RESTART WITH 105;
```

Compatibility

`ALTER SEQUENCE` conforms to the SQL standard, except for the `OWNED BY` and `SET SCHEMA` clauses, which are Greenplum Database extensions.

See Also

[CREATE SEQUENCE](#), [DROP SEQUENCE](#), [ALTER TABLE](#)

ALTER TABLE

Changes the definition of a table.

Synopsis

```
ALTER TABLE [ONLY] name RENAME [COLUMN] column TO new_column
ALTER TABLE name RENAME TO new_name
ALTER TABLE name SET SCHEMA new_schema
ALTER TABLE [ONLY] name SET
    DISTRIBUTED BY (column, [ ... ] )
    | DISTRIBUTED RANDOMLY
    | WITH (REORGANIZE=true|false)
ALTER TABLE [ONLY] name action [, ... ]
ALTER TABLE name
    [ ALTER PARTITION { partition_name | FOR (RANK(number)) | FOR
    (value) } partition_action [...] ]
    partition_action
```

where *action* is one of:

```
ADD [COLUMN] column_name type [column_constraint [ ... ]]
DROP [COLUMN] column [RESTRICT | CASCADE]
ALTER [COLUMN] column TYPE type [USING expression]
ALTER [COLUMN] column SET DEFAULT expression
ALTER [COLUMN] column DROP DEFAULT
ALTER [COLUMN] column { SET | DROP } NOT NULL
ALTER [COLUMN] column SET STATISTICS integer
ADD table_constraint
DROP CONSTRAINT constraint_name [RESTRICT | CASCADE]
DISABLE TRIGGER [trigger_name | ALL | USER]
ENABLE TRIGGER [trigger_name | ALL | USER]
CLUSTER ON index_name
SET WITHOUT CLUSTER
SET WITHOUT OIDS
SET (FILLFACTOR = value)
RESET (FILLFACTOR)
INHERIT parent_table
NO INHERIT parent_table
OWNER TO new_owner
SET TABLESPACE new_tablespace
```

where *partition_action* is one of:

```
ALTER DEFAULT PARTITION
DROP DEFAULT PARTITION [IF EXISTS]
DROP PARTITION [IF EXISTS] { partition_name |
    FOR (RANK(number)) | FOR (value) } [CASCADE]
TRUNCATE DEFAULT PARTITION
```

```

TRUNCATE PARTITION { partition_name | FOR (RANK(number)) |
FOR (value) }
RENAME DEFAULT PARTITION TO new_partition_name
RENAME PARTITION { partition_name | FOR (RANK(number)) |
FOR (value) } TO new_partition_name
ADD DEFAULT PARTITION name [ ( subpartition_spec ) ]
ADD PARTITION [name] partition_element
[ ( subpartition_spec ) ]
EXCHANGE PARTITION { partition_name | FOR (RANK(number)) |
FOR (value) } WITH TABLE table_name
[ WITH | WITHOUT VALIDATION ]
EXCHANGE DEFAULT PARTITION WITH TABLE table_name
[ WITH | WITHOUT VALIDATION ]
SET SUBPARTITION TEMPLATE (subpartition_spec)
SPLIT DEFAULT PARTITION AT (list_value)
START([datatype] range_value) [INCLUSIVE | EXCLUSIVE]
END([datatype] range_value) [INCLUSIVE | EXCLUSIVE]
[ INTO ( PARTITION new_partition_name,
PARTITION default_partition_name ) ]
SPLIT PARTITION { partition_name | FOR (RANK(number)) |
FOR (value) } AT (value)
[ INTO (PARTITION partition_name, PARTITION
partition_name)]

```

where *partition_element* is:

```

VALUES (list_value [, ...] )
| START ([datatype] 'start_value') [INCLUSIVE | EXCLUSIVE]
[ END ([datatype] 'end_value') [INCLUSIVE | EXCLUSIVE] ]
| END ([datatype] 'end_value') [INCLUSIVE | EXCLUSIVE]
[ WITH ( partition_storage_parameter=value [, ...] ) ]
[ TABLESPACE tablespace ]

```

where *subpartition_spec* is:

```
subpartition_element [, ...]
```

and *subpartition_element* is:

```

DEFAULT SUBPARTITION name
| [SUBPARTITION name] VALUES (list_value [, ...] )
| [SUBPARTITION name]
START ([datatype] 'start_value') [INCLUSIVE | EXCLUSIVE]
[ END ([datatype] 'end_value') [INCLUSIVE | EXCLUSIVE] ]
[ EVERY ( [number | datatype] 'interval_value') ]
| [SUBPARTITION name]
END ([datatype] 'end_value') [INCLUSIVE | EXCLUSIVE]
[ EVERY ( [number | datatype] 'interval_value') ]
[ WITH ( partition_storage_parameter=value [, ...] ) ]
[ TABLESPACE tablespace ]

```

where *storage_parameter* is:

```
APPENDONLY={ TRUE | FALSE }
COMPRESSTYPE={ ZLIB | QUICKLZ }
COMPRESSLEVEL={ 0-9 | 1 }
ORIENTATION={ COLUMN | ROW }
FILLFACTOR={ 10-100 }
```

Description

`ALTER TABLE` changes the definition of an existing table. There are several subforms:

- **ADD COLUMN** — Adds a new column to the table, using the same syntax as `CREATE TABLE`.
- **DROP COLUMN** — Drops a column from a table. Note that if you drop table columns that are being used as the Greenplum Database distribution key, the distribution policy for the table will be changed to `DISTRIBUTED RANDOMLY`. Indexes and table constraints involving the column will be automatically dropped as well. You will need to say `CASCADE` if anything outside the table depends on the column (such as views).
- **ALTER COLUMN TYPE** — Changes the data type of a column of a table. Note that you cannot alter column data types that are being used as the Greenplum Database distribution key. Indexes and simple table constraints involving the column will be automatically converted to use the new column type by reparsing the originally supplied expression. The optional `USING` clause specifies how to compute the new column value from the old. If omitted, the default conversion is the same as an assignment cast from old data type to new. A `USING` clause must be provided if there is no implicit or assignment cast from old to new type.
- **SET/DROP DEFAULT** — Sets or removes the default value for a column. The default values only apply to subsequent `INSERT` commands. They do not cause rows already in the table to change. Defaults may also be created for views, in which case they are inserted into statements on the view before the view's `ON INSERT` rule is applied.
- **SET/DROP NOT NULL** — Changes whether a column is marked to allow null values or to reject null values. You can only use `SET NOT NULL` when the column contains no null values.
- **SET STATISTICS** — Sets the per-column statistics-gathering target for subsequent `ANALYZE` operations. The target can be set in the range 0 to 1000, or set to -1 to revert to using the system default statistics target (`default_statistics_target`).
- **ADD table_constraint** — Adds a new constraint to a table using the same syntax as `CREATE TABLE`.
- **DROP CONSTRAINT** — Drops the specified constraint on a table.

- **DISABLE/ENABLE TRIGGER** — Disables or enables trigger(s) belonging to the table. A disabled trigger is still known to the system, but is not executed when its triggering event occurs. For a deferred trigger, the enable status is checked when the event occurs, not when the trigger function is actually executed. One may disable or enable a single trigger specified by name, or all triggers on the table, or only user-created triggers. Disabling or enabling constraint triggers requires superuser privileges. Note that foreign key constraint triggers are not currently supported in Greenplum Database, and triggers in general have very limited functionality due to the parallelism of Greenplum Database. See `CREATE TRIGGER` for more information.
- **CLUSTER/SET WITHOUT CLUSTER** — Selects or removes the default index for future `CLUSTER` operations. It does not actually re-cluster the table. Note that `CLUSTER` is not the recommended way to physically reorder a table in Greenplum Database because it takes so long. It is better to recreate the table with `CREATE TABLE AS` and order it by the index column(s).
- **SET WITHOUT OIDS** — Removes the OID system column from the table. Note that there is no variant of `ALTER TABLE` that allows OIDs to be restored to a table once they have been removed.
- **SET (FILLFACTOR = value) / RESET (FILLFACTOR)** — Changes the fillfactor for the table. The fillfactor for a table is a percentage between 10 and 100. 100 (complete packing) is the default. When a smaller fillfactor is specified, `INSERT` operations pack table pages only to the indicated percentage; the remaining space on each page is reserved for updating rows on that page. This gives `UPDATE` a chance to place the updated copy of a row on the same page as the original, which is more efficient than placing it on a different page. For a table whose entries are never updated, complete packing is the best choice, but in heavily updated tables smaller fillfactors are appropriate. Note that the table contents will not be modified immediately by this command. You will need to rewrite the table to get the desired effects.
- **SET DISTRIBUTED** — Changes the distribution policy of a table. Changes to a hash distribution policy will cause the table data to be physically redistributed on disk, which can be resource intensive.
- **INHERIT parent_table / NO INHERIT parent_table** — Adds or removes the target table as a child of the specified parent table. Queries against the parent will include records of its child table. To be added as a child, the target table must already contain all the same columns as the parent (it could have additional columns, too). The columns must have matching data types, and if they have `NOT NULL` constraints in the parent then they must also have `NOT NULL` constraints in the child. There must also be matching child-table constraints for all `CHECK` constraints of the parent.
- **OWNER** — Changes the owner of the table, sequence, or view to the specified user.
- **SET TABLESPACE** — Changes the table's tablespace to the specified tablespace and moves the data file(s) associated with the table to the new tablespace. Indexes on the table, if any, are not moved; but they can be moved separately with additional `SET TABLESPACE` commands. See also `CREATE TABLESPACE`.
- **RENAME** — Changes the name of a table (or an index, sequence, or view) or the name of an individual column in a table. There is no effect on the stored data. Note that Greenplum Database distribution key columns cannot be renamed.

- **SET SCHEMA** — Moves the table into another schema. Associated indexes, constraints, and sequences owned by table columns are moved as well.
- **ALTER PARTITION | DROP PARTITION | RENAME PARTITION | TRUNCATE PARTITION | ADD PARTITION | SPLIT PARTITION | EXCHANGE PARTITION | SET SUBPARTITION TEMPLATE** — Changes the structure of a partitioned table. In most cases, you must go through the parent table to alter one of its child table partitions.

You must own the table to use `ALTER TABLE`. To change the schema of a table, you must also have `CREATE` privilege on the new schema. To add the table as a new child of a parent table, you must own the parent table as well. To alter the owner, you must also be a direct or indirect member of the new owning role, and that role must have `CREATE` privilege on the table's schema. A superuser has these privileges automatically.

Parameters

ONLY

Only perform the operation on the table name specified. If the `ONLY` keyword is not used, the operation will be performed on the named table and any child table partitions associated with that table.

name

The name (possibly schema-qualified) of an existing table to alter. If `ONLY` is specified, only that table is altered. If `ONLY` is not specified, the table and all its descendant tables (if any) are updated.

column

Name of a new or existing column. Note that Greenplum Database distribution key columns must be treated with special care. Altering or dropping these columns can change the distribution policy for the table.

new_column

New name for an existing column.

new_name

New name for the table.

type

Data type of the new column, or new data type for an existing column. If changing the data type of a Greenplum distribution key column, you are only allowed to change it to a compatible type (for example, `text` to `varchar` is OK, but `text` to `int` is not).

table_constraint

New table constraint for the table. Note that foreign key constraints are currently not supported in Greenplum Database. Also a table is only allowed one unique constraint and the uniqueness must be within the Greenplum Database distribution key.

constraint_name

Name of an existing constraint to drop.

CASCADE

Automatically drop objects that depend on the dropped column or constraint (for example, views referencing the column).

RESTRICT

Refuse to drop the column or constraint if there are any dependent objects. This is the default behavior.

trigger_name

Name of a single trigger to disable or enable. Note that Greenplum Database has limited support of triggers. See `CREATE TRIGGER` for more information.

ALL

Disable or enable all triggers belonging to the table including constraint related triggers. This requires superuser privilege.

USER

Disable or enable all user-created triggers belonging to the table.

index_name

The index name on which the table should be marked for clustering. Note that `CLUSTER` is not the recommended way to physically reorder a table in Greenplum Database because it takes so long. It is better to recreate the table with `CREATE TABLE AS` and order it by the index column(s).

FILLFACTOR

Set the fillfactor percentage for a table.

value

The new value for the `FILLFACTOR` parameter, which is a percentage between 10 and 100. 100 is the default.

DISTRIBUTED BY (*column*) | DISTRIBUTED RANDOMLY

Specifies the distribution policy for a table. Changing a hash distribution policy will cause the table data to be physically redistributed on disk, which can be resource intensive. If you declare the same hash distribution policy or change from hash to random distribution, data will not be redistributed unless you declare `SET WITH (REORGANIZE=true)`.

REORGANIZE=true|false

Use `REORGANIZE=true` when the hash distribution policy has not changed or when you have changed from a hash to a random distribution, and you want to redistribute the data anyways.

parent_table

A parent table to associate or de-associate with this table.

new_owner

The role name of the new owner of the table.

new_tablespace

The name of the tablespace to which the table will be moved.

new_schema

The name of the schema to which the table will be moved.

parent_table_name

When altering a partitioned table, the name of the top-level parent table.

ALTER [DEFAULT] PARTITION

If altering a partition deeper than the first level of partitions, the `ALTER PARTITION` clause is used to specify which subpartition in the hierarchy you want to alter.

DROP [DEFAULT] PARTITION

Drops the specified partition. If the partition has subpartitions, the subpartitions are automatically dropped as well.

TRUNCATE [DEFAULT] PARTITION

Truncates the specified partition. If the partition has subpartitions, the subpartitions are automatically truncated as well.

RENAME [DEFAULT] PARTITION

Changes the partition name of a partition (not the relation name). Partitioned tables are created using the naming convention:

```
<parentname>_<level>_prt_<partition_name>.
```

ADD DEFAULT PARTITION

Adds a default partition to an existing partition design. When data does not match to an existing partition, it is inserted into the default partition. Partition designs that do not have a default partition will reject incoming rows that do not match to an existing partition. Default partitions must be given a name.

ADD PARTITION

partition_element - Using the existing partition type of the table (range or list), defines the boundaries of new partition you are adding.

name - A name for this new partition.

VALUES - For list partitions, defines the value(s) that the partition will contain.

START - For range partitions, defines the starting range value for the partition. By default, start values are `INCLUSIVE`. For example, if you declared a start date of `'2008-01-01'`, then the partition would contain all dates greater than or equal to `'2008-01-01'`. Typically the data type of the `START` expression is the same type as the partition key column. If that is not the case, then you must explicitly cast to the intended data type.

END - For range partitions, defines the ending range value for the partition. By default, end values are `EXCLUSIVE`. For example, if you declared an end date of `'2008-02-01'`, then the partition would contain all dates less than but not equal to `'2008-02-01'`. Typically the data type of the `END` expression is the same type as the partition key column. If that is not the case, then you must explicitly cast to the intended data type.

WITH - Sets the table storage options for a partition. For example, you may want older partitions to be append-only tables and newer partitions to be regular heap tables. See “[CREATE TABLE](#)” on page 415 for a description of the storage options.

TABLESPACE - Currently not supported in Greenplum Database. The name of the tablespace in which the partition is to be created.

subpartition_spec - Only allowed on partition designs that were created without a subpartition template. Declares a subpartition specification for the new partition you are adding. If the partitioned table was originally defined using a subpartition template, then the template will be used to generate the subpartitions automatically.

EXCHANGE [DEFAULT] PARTITION

Exchanges another table into the partition hierarchy into the place of an existing partition. In a multi-level partition design, you can only exchange the lowest level partitions (those that contain data).

WITH TABLE *table_name* - The name of the table you are swapping in to the partition design.

WITH | WITHOUT VALIDATION - Validates that the data in the table matches the `CHECK` constraint of the partition you are exchanging. The default is to validate the data against the `CHECK` constraint.

SET SUBPARTITION TEMPLATE

Modifies the subpartition template for an existing partition. After a new subpartition template is set, all new partitions added will have the new subpartition design (existing partitions are not modified).

SPLIT DEFAULT PARTITION

Splits a default partition. In a multi-level partition design, you can only split the lowest level default partitions (those that contain data). Splitting a default partition creates a new partition containing the values specified and leaves the default partition containing any values that do not match to an existing partition.

AT - For list partitioned tables, specifies a single list value that should be used as the criteria for the split.

START - For range partitioned tables, specifies a starting value for the new partition.

END - For range partitioned tables, specifies an ending value for the new partition.

INTO - Allows you to specify a name for the new partition. When using the `INTO` clause to split a default partition, the second partition name specified should always be that of the existing default partition. If you do not know the name of the default partition, you can look it up using the `pg_partitions` view.

SPLIT PARTITION

Splits an existing partition into two partitions. In a multi-level partition design, you can only split the lowest level partitions (those that contain data).

AT - Specifies a single value that should be used as the criteria for the split. The partition will be divided into two new partitions with the split value specified being the starting range for the *latter* partition.

INTO - Allows you to specify names for the two new partitions created by the split.

partition_name

The given name of a partition.

FOR (RANK (number))

For range partitions, the rank of the partition in the range.

FOR ('value')

Specifies a partition by declaring a value that falls within the partition boundary specification. If the value declared with `FOR` matches to both a partition and one of its subpartitions (for example, if the value is a date and the table is partitioned by month and then by day), then `FOR` will operate on the first level where a match is found (for example, the monthly partition). If your intent is to operate on a subpartition, you must declare so as follows:

```
ALTER TABLE name ALTER PARTITION FOR ('2008-10-01') DROP
PARTITION FOR ('2008-10-01');
```

Notes

Take special care when altering or dropping columns that are part of the Greenplum Database distribution key as this can change the distribution policy for the table.

Greenplum Database does not currently support foreign key constraints. For a unique constraint to be enforced in Greenplum Database, the table must be hash-distributed (not `DISTRIBUTED RANDOMLY`), and all of the distribution key columns must be the same as the initial columns of the unique constraint columns.

Adding a `CHECK` or `NOT NULL` constraint requires scanning the table to verify that existing rows meet the constraint.

When a column is added with `ADD COLUMN`, all existing rows in the table are initialized with the column's default value (`NULL` if no `DEFAULT` clause is specified). Adding a column with a non-null default or changing the type of an existing column will require the entire table to be rewritten. This may take a significant amount of time for a large table; and it will temporarily require double the disk space.

You can specify multiple changes in a single `ALTER TABLE` command, which will be done in a single pass over the table.

The `DROP COLUMN` form does not physically remove the column, but simply makes it invisible to SQL operations. Subsequent insert and update operations in the table will store a null value for the column. Thus, dropping a column is quick but it will not immediately reduce the on-disk size of your table, as the space occupied by the dropped column is not reclaimed. The space will be reclaimed over time as existing rows are updated.

The fact that `ALTER TYPE` requires rewriting the whole table is sometimes an advantage, because the rewriting process eliminates any dead space in the table. For example, to reclaim the space occupied by a dropped column immediately, the fastest way is: `ALTER TABLE table ALTER COLUMN anycol TYPE sametype;` Where *anycol* is any remaining table column and *sametype* is the same type that column already has. This results in no semantically-visible change in the table, but the command forces rewriting, which gets rid of no-longer-useful data.

If a table is partitioned or has any descendant tables, it is not permitted to add, rename, or change the type of a column in the parent table without doing the same to the descendants. This ensures that the descendants always have columns matching the parent.

To see the structure of a partitioned table, you can use the view `pg_partitions`. This view can help identify the particular partitions you may want to alter.

A recursive `DROP COLUMN` operation will remove a descendant table's column only if the descendant does not inherit that column from any other parents and never had an independent definition of the column. A nonrecursive `DROP COLUMN (ALTER TABLE ONLY ... DROP COLUMN)` never removes any descendant columns, but instead marks them as independently defined rather than inherited.

The `TRIGGER`, `CLUSTER`, `OWNER`, and `TABLESPACE` actions never recurse to descendant tables; that is, they always act as though `ONLY` were specified. Adding a constraint can recurse only for `CHECK` constraints.

Changing any part of a system catalog table is not permitted.

Examples

Add a column to a table:

```
ALTER TABLE distributors ADD COLUMN address varchar(30);
```

Rename an existing column:

```
ALTER TABLE distributors RENAME COLUMN address TO city;
```

Rename an existing table:

```
ALTER TABLE distributors RENAME TO suppliers;
```

Add a not-null constraint to a column:

```
ALTER TABLE distributors ALTER COLUMN street SET NOT NULL;
```

Add a check constraint to a table:

```
ALTER TABLE distributors ADD CONSTRAINT zipchk CHECK
(char_length(zipcode) = 5);
```

Move a table to a different schema:

```
ALTER TABLE myschema.distributors SET SCHEMA yourschema;
```

Add a new partition to a partitioned table:

```
ALTER TABLE sales ADD PARTITION
      START (date '2009-02-01') INCLUSIVE
      END (date '2009-03-01') EXCLUSIVE;
```

Add a default partition to an existing partition design:

```
ALTER TABLE sales ADD DEFAULT PARTITION other;
```

Rename a partition:

```
ALTER TABLE sales RENAME PARTITION FOR ('2008-01-01') TO
jan08;
```

Drop the first (oldest) partition in a range sequence:

```
ALTER TABLE sales DROP PARTITION FOR (RANK(1));
```

Exchange a table into your partition design:

```
ALTER TABLE sales EXCHANGE PARTITION FOR ('2008-01-01') WITH
TABLE jan08;
```

Split the default partition (where the existing default partition's name is *'other'*) to add a new monthly partition for January 2009:

```
ALTER TABLE sales SPLIT DEFAULT PARTITION
      START ('2009-01-01') INCLUSIVE
      END ('2009-02-01') EXCLUSIVE
      INTO (PARTITION jan09, PARTITION other);
```

Split a monthly partition into two with the first partition containing dates January 1-15 and the second partition containing dates January 16-31:

```
ALTER TABLE sales SPLIT PARTITION FOR ('2008-01-01')
      AT ('2008-01-16')
      INTO (PARTITION jan081to15, PARTITION jan0816to31);
```

Compatibility

The `ADD`, `DROP`, and `SET DEFAULT` forms conform with the SQL standard. The other forms are Greenplum Database extensions of the SQL standard. Also, the ability to specify more than one manipulation in a single `ALTER TABLE` command is an extension.

`ALTER TABLE DROP COLUMN` can be used to drop the only column of a table, leaving a zero-column table. This is an extension of SQL, which disallows zero-column tables.

See Also

`CREATE TABLE`, `DROP TABLE`

ALTER TABLESPACE

Changes the definition of a tablespace.

Synopsis

```
ALTER TABLESPACE name RENAME TO newname
```

```
ALTER TABLESPACE name OWNER TO newowner
```

Description

ALTER TABLESPACE changes the definition of a tablespace.

You must own the tablespace to use ALTER TABLESPACE. To alter the owner, you must also be a direct or indirect member of the new owning role. (Note that superusers have these privileges automatically.)

Parameters

name

The name of an existing tablespace.

newname

The new name of the tablespace. The new name cannot begin with *pg_* (reserved for system tablespaces).

newowner

The new owner of the tablespace.

Examples

Rename tablespace *index_space* to *fast_raid*:

```
ALTER TABLESPACE index_space RENAME TO fast_raid;
```

Change the owner of tablespace *index_space*:

```
ALTER TABLESPACE index_space OWNER TO mary;
```

Compatibility

There is no ALTER TABLESPACE statement in the SQL standard.

See Also

CREATE TABLESPACE, DROP TABLESPACE

ALTER TRIGGER

Changes the definition of a trigger.

Synopsis

```
ALTER TRIGGER name ON table RENAME TO newname
```

Description

`ALTER TRIGGER` changes properties of an existing trigger. The `RENAME` clause changes the name of the given trigger without otherwise changing the trigger definition. You must own the table on which the trigger acts to be allowed to change its properties.

Parameters

name

The name of an existing trigger to alter.

table

The name of the table on which this trigger acts.

newname

The new name for the trigger.

Notes

The ability to temporarily enable or disable a trigger is provided by `ALTER TABLE`, not by `ALTER TRIGGER`, because `ALTER TRIGGER` has no convenient way to express the option of enabling or disabling all of a table's triggers at once.

Note that Greenplum Database has limited support of triggers in this release. See `CREATE TRIGGER` for more information.

Examples

To rename an existing trigger:

```
ALTER TRIGGER emp_stamp ON emp RENAME TO emp_track_chgs;
```

Compatibility

`ALTER TRIGGER` is a Greenplum Database extension of the SQL standard.

See Also

`ALTER TABLE`, `CREATE TRIGGER`, `DROP TRIGGER`

ALTER TYPE

Changes the definition of a data type.

Synopsis

```
ALTER TYPE name OWNER TO new_owner
ALTER TYPE name SET SCHEMA new_schema
```

Description

`ALTER TYPE` changes the definition of an existing type. The only currently available capabilities are changing the owner and schema of a type.

You must own the type to use `ALTER TYPE`. To change the schema of a type, you must also have `CREATE` privilege on the new schema. To alter the owner, you must also be a direct or indirect member of the new owning role, and that role must have `CREATE` privilege on the type's schema. (These restrictions enforce that altering the owner doesn't do anything you couldn't do by dropping and recreating the type. However, a superuser can alter ownership of any type anyway.)

Parameters

name

The name (optionally schema-qualified) of an existing type to alter.

new_owner

The user name of the new owner of the type.

new_schema

The new schema for the type.

Examples

To change the owner of the user-defined type *email* to *joe*:

```
ALTER TYPE email OWNER TO joe;
```

To change the schema of the user-defined type *email* to *customers*:

```
ALTER TYPE email SET SCHEMA customers;
```

Compatibility

There is no `ALTER TYPE` statement in the SQL standard.

See Also

[CREATE TYPE](#), [DROP TYPE](#)

ALTER USER

Changes the definition of a database role (user).

Synopsis

```
ALTER USER name RENAME TO newname
ALTER USER name SET config_parameter {TO | =} {value | DEFAULT}
ALTER USER name RESET config_parameter
ALTER USER name [ [WITH] option [ ... ] ]
```

where *option* can be:

```
    SUPERUSER | NOSUPERUSER
  | CREATEDB | NOCREATEDB
  | CREATEROLE | NOCREATEROLE
  | CREATEUSER | NOCREATEUSER
  | INHERIT | NOINHERIT
  | LOGIN | NOLOGIN
  | [ ENCRYPTED | UNENCRYPTED ] PASSWORD 'password'
  | VALID UNTIL 'timestamp'
```

Description

ALTER USER is a deprecated command but is still accepted for historical reasons. It is an alias for ALTER ROLE. See [ALTER ROLE](#) for more information.

Compatibility

The ALTER USER statement is a Greenplum Database extension. The SQL standard leaves the definition of users to the implementation.

See Also

[ALTER ROLE](#)

ANALYZE

Collects statistics about a database.

Synopsis

```
ANALYZE [VERBOSE] [table [ (column [, ...] ) ]]
```

Description

`ANALYZE` collects statistics about the contents of tables in the database, and stores the results in the system table `pg_statistic`. Subsequently, the query planner uses these statistics to help determine the most efficient execution plans for queries.

With no parameter, `ANALYZE` examines every table in the current database. With a parameter, `ANALYZE` examines only that table. It is further possible to give a list of column names, in which case only the statistics for those columns are collected.

Parameters

VERBOSE

Enables display of progress messages. When specified, `ANALYZE` emits progress messages to indicate which table is currently being processed. Various statistics about the tables are printed as well.

table

The name (possibly schema-qualified) of a specific table to analyze. Defaults to all tables in the current database.

column

The name of a specific column to analyze. Defaults to all columns.

Notes

It is a good idea to run `ANALYZE` periodically, or just after making major changes in the contents of a table. Accurate statistics will help the query planner to choose the most appropriate query plan, and thereby improve the speed of query processing. A common strategy is to run `VACUUM` and `ANALYZE` once a day during a low-usage time of day.

`ANALYZE` requires only a read lock on the target table, so it can run in parallel with other activity on the table.

The statistics collected by `ANALYZE` usually include a list of some of the most common values in each column and a histogram showing the approximate data distribution in each column. One or both of these may be omitted if `ANALYZE` deems them uninteresting (for example, in a unique-key column, there are no common values) or if the column data type does not support the appropriate operators.

For large tables, `ANALYZE` takes a random sample of the table contents, rather than examining every row. This allows even very large tables to be analyzed in a small amount of time. Note, however, that the statistics are only approximate, and will change slightly each time `ANALYZE` is run, even if the actual table contents did not change. This may result in small changes in the planner's estimated costs shown by `EXPLAIN`. In rare situations, this non-determinism will cause the query optimizer to choose a different query plan between runs of `ANALYZE`. To avoid this, raise the amount of statistics collected by `ANALYZE` by adjusting the `default_statistics_target` configuration parameter, or on a column-by-column basis by setting the per-column statistics target with `ALTER TABLE ... ALTER COLUMN ... SET STATISTICS` (see `ALTER TABLE`). The target value sets the maximum number of entries in the most-common-value list and the maximum number of bins in the histogram. The default target value is 10, but this can be adjusted up or down to trade off accuracy of planner estimates against the time taken for `ANALYZE` and the amount of space occupied in `pg_statistic`. In particular, setting the statistics target to zero disables collection of statistics for that column. It may be useful to do that for columns that are never used as part of the `WHERE`, `GROUP BY`, or `ORDER BY` clauses of queries, since the planner will have no use for statistics on such columns.

The largest statistics target among the columns being analyzed determines the number of table rows sampled to prepare the statistics. Increasing the target causes a proportional increase in the time and space needed to do `ANALYZE`.

Examples

Collect statistics for the table *mytable*:

```
ANALYZE mytable;
```

Compatibility

There is no `ANALYZE` statement in the SQL standard.

See Also

[ALTER TABLE](#), [EXPLAIN](#), [VACUUM](#)

BEGIN

Starts a transaction block.

Synopsis

```
BEGIN [WORK | TRANSACTION] [SERIALIZABLE | REPEATABLE READ |  
READ COMMITTED | READ UNCOMMITTED] [READ WRITE | READ ONLY]
```

Description

`BEGIN` initiates a transaction block, that is, all statements after a `BEGIN` command will be executed in a single transaction until an explicit `COMMIT` or `ROLLBACK` is given. By default (without `BEGIN`), Greenplum Database executes transactions in autocommit mode, that is, each statement is executed in its own transaction and a commit is implicitly performed at the end of the statement (if execution was successful, otherwise a rollback is done).

Statements are executed more quickly in a transaction block, because transaction start/commit requires significant CPU and disk activity. Execution of multiple statements inside a transaction is also useful to ensure consistency when making several related changes: other sessions will be unable to see the intermediate states wherein not all the related updates have been done.

If the isolation level or read/write mode is specified, the new transaction has those characteristics, as if `SET TRANSACTION` was executed.

Parameters

WORK
TRANSACTION

Optional key words. They have no effect.

SERIALIZABLE
REPEATABLE READ
READ COMMITTED
READ UNCOMMITTED

The SQL standard defines four transaction isolation levels: `READ COMMITTED`, `READ UNCOMMITTED`, `SERIALIZABLE`, and `REPEATABLE READ`. The default behavior is that a statement can only see rows committed before it began (`READ COMMITTED`). In Greenplum Database `READ UNCOMMITTED` is treated the same as `READ COMMITTED`. `SERIALIZABLE` is supported the same as `REPEATABLE READ` wherein all statements of the current transaction can only see rows committed before the first statement was executed in the transaction. `SERIALIZABLE` is the strictest transaction isolation. This level emulates serial transaction execution, as if transactions had been executed one after another, serially, rather than concurrently. Applications using this level must be prepared to retry transactions due to serialization failures.

READ WRITE
READ ONLY

Determines whether the transaction is read/write or read-only. Read/write is the default. When a transaction is read-only, the following SQL commands are disallowed: `INSERT`, `UPDATE`, `DELETE`, and `COPY FROM` if the table they would write to is not a temporary table; all `CREATE`, `ALTER`, and `DROP` commands; `GRANT`, `REVOKE`, `TRUNCATE`; and `EXPLAIN ANALYZE` and `EXECUTE` if the command they would execute is among those listed.

Notes

`START TRANSACTION` has the same functionality as `BEGIN`.

Use `COMMIT` or `ROLLBACK` to terminate a transaction block.

Issuing `BEGIN` when already inside a transaction block will provoke a warning message. The state of the transaction is not affected. To nest transactions within a transaction block, use savepoints (see `SAVEPOINT`).

Examples

To begin a transaction block:

```
BEGIN;
```

Compatibility

`BEGIN` is a Greenplum Database language extension. It is equivalent to the SQL-standard command `START TRANSACTION`.

Incidentally, the `BEGIN` key word is used for a different purpose in embedded SQL. You are advised to be careful about the transaction semantics when porting database applications.

See Also

`COMMIT`, `ROLLBACK`, `START TRANSACTION`, `SAVEPOINT`

CHECKPOINT

Forces a transaction log checkpoint.

Synopsis

CHECKPOINT

Description

Write-Ahead Logging (WAL) puts a checkpoint in the transaction log every so often. The automatic checkpoint interval is set per Greenplum Database segment instance by the server configuration parameters *checkpoint_segments* and *checkpoint_timeout*. The `CHECKPOINT` command forces an immediate checkpoint when the command is issued, without waiting for a scheduled checkpoint.

A checkpoint is a point in the transaction log sequence at which all data files have been updated to reflect the information in the log. All data files will be flushed to disk.

Only superusers may call `CHECKPOINT`. The command is not intended for use during normal operation.

Compatibility

The `CHECKPOINT` command is a Greenplum Database language extension.

CLOSE

Closes a cursor.

Synopsis

```
CLOSE cursor_name
```

Description

`CLOSE` frees the resources associated with an open cursor. After the cursor is closed, no subsequent operations are allowed on it. A cursor should be closed when it is no longer needed.

Every non-holdable open cursor is implicitly closed when a transaction is terminated by `COMMIT` or `ROLLBACK`. A holdable cursor is implicitly closed if the transaction that created it aborts via `ROLLBACK`. If the creating transaction successfully commits, the holdable cursor remains open until an explicit `CLOSE` is executed, or the client disconnects.

Parameters

cursor_name

The name of an open cursor to close.

Notes

Greenplum Database does not have an explicit `OPEN` cursor statement. A cursor is considered open when it is declared. Use the `DECLARE` statement to declare (and open) a cursor.

You can see all available cursors by querying the `pg_cursors` system view.

Examples

Close the cursor *portala*:

```
CLOSE portala;
```

Compatibility

`CLOSE` is fully conforming with the SQL standard.

See Also

[DECLARE](#), [FETCH](#), [MOVE](#)

CLUSTER

Physically reorders a table on disk according to an index. Not a recommended operation in Greenplum Database.

Synopsis

```
CLUSTER indexname ON tablename
```

```
CLUSTER tablename
```

```
CLUSTER
```

Description

`CLUSTER` orders a table based on an index. Clustering an index means that the records are physically ordered on disk according to the index information. If the records you need are distributed randomly on disk, then the database has to seek across the disk to get the records requested. If those records are stored more closely together, then the fetching from disk is more sequential. A good example for a clustered index is on a date column where the data is ordered sequentially by date. A query against a specific date range will result in an ordered fetch from the disk, which leverages faster sequential access.

Clustering is a one-time operation: when the table is subsequently updated, the changes are not clustered. That is, no attempt is made to store new or updated rows according to their index order. If one wishes, one can periodically recluster by issuing the command again.

When a table is clustered using this command, Greenplum Database remembers on which index it was clustered. The form `CLUSTER tablename` reclusters the table on the same index that it was clustered before. `CLUSTER` without any parameter reclusters all previously clustered tables in the current database that the calling user owns, or all tables if called by a superuser. This form of `CLUSTER` cannot be executed inside a transaction block.

When a table is being clustered, an `ACCESS EXCLUSIVE` lock is acquired on it. This prevents any other database operations (both reads and writes) from operating on the table until the `CLUSTER` is finished.

Parameters

indexname

The name of an index.

tablename

The name (optionally schema-qualified) of a table.

Notes

In cases where you are accessing single rows randomly within a table, the actual order of the data in the table is unimportant. However, if you tend to access some data more than others, and there is an index that groups them together, you will benefit from using `CLUSTER`. If you are requesting a range of indexed values from a table, or a single indexed value that has multiple rows that match, `CLUSTER` will help because once the index identifies the table page for the first row that matches, all other rows that match are probably already on the same table page, and so you save disk accesses and speed up the query.

During the cluster operation, a temporary copy of the table is created that contains the table data in the index order. Temporary copies of each index on the table are created as well. Therefore, you need free space on disk at least equal to the sum of the table size and the index sizes.

Because the query planner records statistics about the ordering of tables, it is advisable to run `ANALYZE` on the newly clustered table. Otherwise, the planner may make poor choices of query plans.

There is another way to cluster data. The `CLUSTER` command reorders the original table by scanning it using the index you specify. This can be slow on large tables because the rows are fetched from the table in index order, and if the table is disordered, the entries are on random pages, so there is one disk page retrieved for every row moved. (Greenplum Database has a cache, but the majority of a big table will not fit in the cache.) The other way to cluster a table is to use a statement such as:

```
CREATE TABLE newtable AS SELECT * FROM table ORDER BY column;
```

This uses the Greenplum Database sorting code to produce the desired order, which is usually much faster than an index scan for disordered data. Then you drop the old table, use `ALTER TABLE ... RENAME` to rename *newtable* to the old name, and recreate the table's indexes. The big disadvantage of this approach is that it does not preserve OIDs, constraints, granted privileges, and other ancillary properties of the table — all such items must be manually recreated. Another disadvantage is that this way requires a sort temporary file about the same size as the table itself, so peak disk usage is about three times the table size instead of twice the table size.

Examples

Cluster the table *employees* on the basis of its index *emp_ind*:

```
CLUSTER emp_ind ON emp;
```

Cluster a large table by recreating it and loading it in the correct index order:

```
CREATE TABLE newtable AS SELECT * FROM table ORDER BY column;
DROP table;
ALTER TABLE newtable RENAME TO table;
CREATE INDEX column_ix ON table (column);
VACUUM ANALYZE table;
```

Compatibility

There is no `CLUSTER` statement in the SQL standard.

See Also

`CREATE TABLE AS`, `CREATE INDEX`

COMMENT

Defines or change the comment of an object.

Synopsis

```
COMMENT ON
{ TABLE object_name |
  COLUMN table_name.column_name |
  AGGREGATE agg_name (agg_type [, ...]) |
  CAST (sourcetype AS targettype) |
  CONSTRAINT constraint_name ON table_name |
  CONVERSION object_name |
  DATABASE object_name |
  DOMAIN object_name |
  FUNCTION func_name ([[argmode] [argname] argtype [, ...]]) |
  INDEX object_name |
  LARGE OBJECT large_object_oid |
  OPERATOR op (leftoperand_type, rightoperand_type) |
  OPERATOR CLASS object_name USING index_method |
  [PROCEDURAL] LANGUAGE object_name |
  ROLE object_name |
  RULE rule_name ON table_name |
  SCHEMA object_name |
  SEQUENCE object_name |
  TABLESPACE object_name |
  TRIGGER trigger_name ON table_name |
  TYPE object_name |
  VIEW object_name }
IS 'text'
```

Description

COMMENT stores a comment about a database object. To modify a comment, issue a new COMMENT command for the same object. Only one comment string is stored for each object. To remove a comment, write NULL in place of the text string. Comments are automatically dropped when the object is dropped.

Comments can be easily retrieved with the psql meta-commands \dd, \d+, and \l+. Other user interfaces to retrieve comments can be built atop the same built-in functions that psql uses, namely obj_description, col_description, and shobj_description.

Parameters

object_name
table_name.column_name
agg_name
constraint_name
func_name

op
rule_name
trigger_name

The name of the object to be commented. Names of tables, aggregates, domains, functions, indexes, operators, operator classes, sequences, types, and views may be schema-qualified.

agg_type

An input data type on which the aggregate function operates. To reference a zero-argument aggregate function, write * in place of the list of input data types.

sourcetype

The name of the source data type of the cast.

targettype

The name of the target data type of the cast.

argmode

The mode of a function argument: either `IN`, `OUT`, or `INOUT`. If omitted, the default is `IN`. Note that `COMMENT ON FUNCTION` does not actually pay any attention to `OUT` arguments, since only the input arguments are needed to determine the function's identity. So it is sufficient to list the `IN` and `INOUT` arguments.

argname

The name of a function argument. Note that `COMMENT ON FUNCTION` does not actually pay any attention to argument names, since only the argument data types are needed to determine the function's identity.

argtype

The data type(s) of the function's arguments (optionally schema-qualified), if any.

large_object_oid

The OID of the large object.

PROCEDURAL

This is a noise word.

text

The new comment, written as a string literal; or `NULL` to drop the comment.

Notes

There is presently no security mechanism for comments: any user connected to a database can see all the comments for objects in that database (although only superusers can change comments for objects that they do not own). For shared objects such as databases, roles, and tablespaces comments are stored globally and any user connected to any database can see all the comments for shared objects. Therefore, do not put security-critical information in comments.

Examples

Attach a comment to the table *mytable*:

```
COMMENT ON TABLE mytable IS 'This is my table.';
```

Remove it again:

```
COMMENT ON TABLE mytable IS NULL;
```

Compatibility

There is no `COMMENT` statement in the SQL standard.

COMMIT

Commits the current transaction.

Synopsis

```
COMMIT [WORK | TRANSACTION]
```

Description

`COMMIT` commits the current transaction. All changes made by the transaction become visible to others and are guaranteed to be durable if a crash occurs.

Parameters

`WORK`
`TRANSACTION`

Optional key words. They have no effect.

Notes

Use `ROLLBACK` to abort a transaction.

Issuing `COMMIT` when not inside a transaction does no harm, but it will provoke a warning message.

Examples

To commit the current transaction and make all changes permanent:

```
COMMIT;
```

Compatibility

The SQL standard only specifies the two forms `COMMIT` and `COMMIT WORK`. Otherwise, this command is fully conforming.

See Also

`BEGIN`, `END`, `START TRANSACTION`, `ROLLBACK`

COPY

Copies data between a file and a table.

Synopsis

```

COPY table [(column [, ...])] FROM {'file' | STDIN}
    [ [WITH]
      [BINARY]
      [OIDS]
      [DELIMITER [ AS ] 'delimiter']
      [NULL [ AS ] 'null string']
      [ESCAPE [ AS ] 'escape' | 'OFF']
      [NEWLINE [ AS ] 'LF' | 'CR' | 'CRLF']
      [CSV [HEADER] [QUOTE [ AS ] 'quote']
        [FORCE NOT NULL column [, ...]]
      [FILL MISSING FIELDS]
    [ [LOG ERRORS INTO error_table] [KEEP]
      SEGMENT REJECT LIMIT count [ROWS | PERCENT] ]

COPY {table [(column [, ...])] | (query)} TO {'file' | STDOUT}
    [ [WITH]
      [BINARY]
      [OIDS]
      [DELIMITER [ AS ] 'delimiter']
      [NULL [ AS ] 'null string']
      [ESCAPE [ AS ] 'escape' | 'OFF']
      [NEWLINE [ AS ] 'LF' | 'CR' | 'CRLF']
      [CSV [HEADER] [QUOTE [ AS ] 'quote']
        [FORCE QUOTE column [, ...]] ]

```

Description

COPY moves data between Greenplum Database tables and standard file-system files. COPY TO copies the contents of a table to a file, while COPY FROM copies data from a file to a table (appending the data to whatever is in the table already). COPY TO can also copy the results of a SELECT query.

If a list of columns is specified, COPY will only copy the data in the specified columns to or from the file. If there are any columns in the table that are not in the column list, COPY FROM will insert the default values for those columns.

COPY with a file name instructs the Greenplum Database master host to directly read from or write to a file. The file must be accessible to the master host and the name must be specified from the viewpoint of the master host. When STDIN or STDOUT is specified, data is transmitted via the connection between the client and the master.

If SEGMENT REJECT LIMIT is used, then a COPY FROM operation will operate in single row error isolation mode. In this release, single row error isolation mode only applies to rows in the input file with format errors — for example, extra or missing attributes, attributes of a wrong data type, or invalid client encoding sequences. Constraint errors

such as violation of a `NOT NULL`, `CHECK`, or `UNIQUE` constraint will still be handled in ‘all-or-nothing’ input mode. The user can specify the number of error rows acceptable (on a per-segment basis), after which the entire `COPY FROM` operation will be aborted and no rows will be loaded. Note that the count of error rows is per-segment, not per entire load operation. If the per-segment reject limit is not reached, then all rows not containing an error will be loaded. If the limit is not reached, all good rows will be loaded and any error rows discarded. If you would like to keep error rows for further examination, you can optionally declare an error table using the `LOG ERRORS INTO` clause. Any rows containing a format error would then be logged to the specified error table.

Outputs

On successful completion, a `COPY` command returns a command tag of the form, where *count* is the number of rows copied:

```
COPY count
```

If running a `COPY FROM` command in single row error isolation mode, the following notice message will be returned if any rows were not loaded due to format errors, where *count* is the number of rows rejected:

```
NOTICE: Rejected count badly formatted rows.
```

Parameters

table

The name (optionally schema-qualified) of an existing table.

column

An optional list of columns to be copied. If no column list is specified, all columns of the table will be copied.

query

A `SELECT` or `VALUES` command whose results are to be copied. Note that parentheses are required around the query.

file

The absolute path name of the input or output file.

STDIN

Specifies that input comes from the client application.

STDOUT

Specifies that output goes to the client application.

BINARY

Causes all data to be stored or read in binary format rather than as text. You cannot specify the `DELIMITER`, `NULL`, or `CSV` options in binary mode. Also, you cannot use single-row error isolation mode (`SEGMENT REJECT LIMIT` clause) with binary files.

OIDS

Specifies copying the OID for each row. (An error is raised if OIDS is specified for a table that does not have OIDs, or in the case of copying a query.)

delimiter

The single ASCII character that separates columns within each row (line) of the file. The default is a tab character in text mode, a comma in CSV mode.

null string

The string that represents a null value. The default is `\N` (backslash-N) in text mode, and an empty value with no quotes in CSV mode. You might prefer an empty string even in text mode for cases where you don't want to distinguish nulls from empty strings. When using `COPY FROM`, any data item that matches this string will be stored as a null value, so you should make sure that you use the same string as you used with `COPY TO`.

escape

Specifies the single character that is used for C escape sequences (such as `\n`, `\t`, `\100`, and so on) and for quoting data characters that might otherwise be taken as row or column delimiters. Make sure to choose an escape character that is not used anywhere in your actual column data. The default escape character is a `\` (backslash), however it is possible to specify any other character to represent an escape. It is also possible to disable escaping by specifying the value `OFF` as the escape value. This is very useful for data such as web log data that has many embedded backslashes that are not intended to be escapes.

NEWLINE

Specifies the newline used in your data files — `LF` (Line feed, 0x0A), `CR` (Carriage return, 0x0D), or `CRLF` (Carriage return plus line feed, 0x0D 0x0A). If not specified, a Greenplum Database segment will detect the newline type by looking at the first row of data it receives and using the first newline type encountered.

CSV

Selects Comma Separated Value (CSV) mode.

HEADER

Specifies that a CSV file contains a header line with the names of each column in the file. On output, the first line contains the column names from the table, and on input, the first line is ignored.

quote

Specifies the quotation character in CSV mode. The default is double-quote.

FORCE QUOTE

In CSV `COPY TO` mode, forces quoting to be used for all non-NULL values in each specified column. NULL output is never quoted.

FORCE NOT NULL

In CSV COPY FROM mode, process each specified column as though it were quoted and hence not a NULL value. For the default null string in CSV mode (' '), this causes missing values to be input as zero-length strings.

FILL MISSING FIELDS

In COPY FROM mode for both TEXT and CSV, specifying FILL MISSING FIELDS will set missing trailing field values to NULL (instead of reporting an error) when a row of data has missing data fields at the end of a line or row. Blank rows, fields with a NOT NULL constraint, and trailing delimiters on a line will still report an error.

LOG ERRORS INTO *error_table* [KEEP]

This is an optional clause that may precede a SEGMENT REJECT LIMIT clause. It specifies an error table where rows with formatting errors will be logged when running in single row error isolation mode. You can then examine this error table to see error rows that were not loaded (if any). If the *error_table* specified already exists, it will be used. If it does not exist, it will be automatically generated. If the command auto-generates the error table and no errors are produced, the default is to drop the error table after the operation completes unless KEEP is specified. If the table is auto-generated and the error limit is exceeded, the entire transaction is rolled back and no error data is saved. If you want the error table to persist in this case, create the error table prior to running the COPY. An error table is defined as follows:

```
CREATE TABLE error_table_name ( cmdtime timestamptz,
  relname text, filename text, linenum int, bytenum int,
  errmsg text, rawdata text, rawbytes bytea )
DISTRIBUTED RANDOMLY;
```

SEGMENT REJECT LIMIT *count* [ROWS | PERCENT]

Runs a COPY FROM operation in single row error isolation mode. If the input rows have format errors they will be discarded provided that the reject limit count is not reached on any Greenplum segment instance during the load operation. The reject limit count can be specified as number of rows (the default) or percentage of total rows (1-100). If PERCENT is used, the percentage of rows per segment is calculated based on the parameter `gp_reject_percent_threshold` (default is 300 rows). Constraint errors such as violation of a NOT NULL, CHECK, or UNIQUE constraint will still be handled in ‘all-or-nothing’ input mode. If the limit is not reached, all good rows will be loaded and any error rows discarded.

Notes

COPY can only be used with tables, not with views. However, you can write COPY (SELECT * FROM viewname) TO

The BINARY key word causes all data to be stored/read as binary format rather than as text. It is somewhat faster than the normal text mode, but a binary-format file is less portable across machine architectures and Greenplum Database versions. Also, you cannot run COPY FROM in single row error isolation mode if the data is in binary format.

You must have `SELECT` privilege on the table whose values are read by `COPY TO`, and insert privilege on the table into which values are inserted by `COPY FROM`.

Files named in a `COPY` command are read or written directly by the database server, not by the client application. Therefore, they must reside on or be accessible to the Greenplum Database master host machine, not the client. They must be accessible to and readable or writable by the Greenplum Database administrative user (the user ID the server runs as), not the client. `COPY` naming a file is only allowed to database superusers, since it allows reading or writing any file that the server has privileges to access.

`COPY FROM` will invoke any triggers and check constraints on the destination table. However, it will not invoke rewrite rules. Note that in this release, violations of constraints are not evaluated for single row error isolation mode.

`COPY` input and output is affected by `DateStyle`. To ensure portability to other Greenplum Database installations that might use non-default `DateStyle` settings, `DateStyle` should be set to `ISO` before using `COPY TO`.

By default, `COPY` stops operation at the first error. This should not lead to problems in the event of a `COPY TO`, but the target table will already have received earlier rows in a `COPY FROM`. These rows will not be visible or accessible, but they still occupy disk space. This may amount to a considerable amount of wasted disk space if the failure happened well into a large `COPY FROM` operation. You may wish to invoke `VACUUM` to recover the wasted space. Another option would be to use single row error isolation mode to filter out error rows while still loading good rows.

File Formats

Text Format

When `COPY` is used without the `BINARY` or `CSV` options, the data read or written is a text file with one line per table row. Columns in a row are separated by the *delimiter* character (tab by default). The column values themselves are strings generated by the output function, or acceptable to the input function, of each attribute's data type. The specified null string is used in place of columns that are null. `COPY FROM` will raise an error if any line of the input file contains more or fewer columns than are expected. If `oids` is specified, the OID is read or written as the first column, preceding the user data columns.

The data file has two reserved characters that have special meaning to `COPY`:

- The designated delimiter character (tab by default), which is used to separate fields in the data file.
- A newline character; `LF` (Line feed, `0x0A`), `CR` (Carriage return, `0x0D`), or `CRLF` (Carriage return plus line feed, `0x0D 0x0A`).

If your data contains any of these characters, you must escape the character so `COPY` treats it as data and not as a field separator or new row.

By default, the escape character is a `\` (backslash). If you want to use a different escape character, you can do so using the `ESCAPE AS` clause. Make sure to choose an escape character that is not used anywhere in your data file as an actual data value. If there is no need to escape the data, you can disable escaping by using `ESCAPE 'OFF'`.

For example, suppose you have a table with three columns and you want to load the following three fields using `COPY`.

- percentage sign = %
- vertical bar = |
- backslash = \

Your designated `DELIMITER` character is | (pipe character), and your designated `ESCAPE` character is * (asterisk). The formatted row in your data file would look like this:

```
percentage sign = % | vertical bar = *| | backslash = \
```

Notice how the pipe character that is part of the data has been escaped using the asterisk character (*). Also notice that we do not need to escape the backslash since we are using an alternative escape character.

By default, backslash characters (\) are used in the `COPY` data to quote data characters that might otherwise be taken as row or column delimiters. In particular, the following characters must be preceded by a backslash (or an alternative escape character) if they appear as part of a column value: backslash itself, newline, carriage return, and the current delimiter character. Alternatively, you can specify a different escape character using the `ESCAPE AS` clause or by turning escaping `OFF`, thus allowing for literal backslashes in the data.

CSV Format

This format is used for importing and exporting the Comma Separated Value (CSV) file format used by many other programs, such as spreadsheets. Instead of the escaping used by Greenplum Database standard text mode, it produces and recognizes the common CSV escaping mechanism.

The values in each record are separated by the `DELIMITER` character. If the value contains the delimiter character, the `QUOTE` character, the `NULL` string, a carriage return, or line feed character, then the whole value is prefixed and suffixed by the `QUOTE` character. You can also use `FORCE QUOTE` to force quotes when outputting non-`NULL` values in specific columns.

The CSV format has no standard way to distinguish a `NULL` value from an empty string. Greenplum Database `COPY` handles this by quoting. A `NULL` is output as the `NULL` string and is not quoted, while a data value matching the `NULL` string is quoted. Therefore, using the default settings, a `NULL` is written as an unquoted empty string, while an empty string is written with double quotes (""). Reading values follows similar rules. You can use `FORCE NOT NULL` to prevent `NULL` input comparisons for specific columns.

Because backslash is not a special character in the CSV format, \., the end-of-data marker, could also appear as a data value. To avoid any misinterpretation, a \. data value appearing as a lone entry on a line is automatically quoted on output, and on input, if quoted, is not interpreted as the end-of-data marker. If you are loading a file created by another application that has a single unquoted column and might have a value of \., you might need to quote that value in the input file.

Note: In CSV mode, all characters are significant. A quoted value surrounded by white space, or any characters other than `DELIMITER`, will include those characters. This can cause errors if you import data from a system that pads CSV lines with white space out to some fixed width. If such a situation arises you might need to preprocess the CSV file to remove the trailing white space, before importing the data into Greenplum Database.

Note: CSV mode will both recognize and produce CSV files with quoted values containing embedded carriage returns and line feeds. Thus the files are not strictly one line per table row like text-mode files.

Note: Many programs produce strange and occasionally perverse CSV files, so the file format is more a convention than a standard. Thus you might encounter some files that cannot be imported using this mechanism, and `COPY` might produce files that other programs cannot process.

Binary Format

The `BINARY` format consists of a file header, zero or more tuples containing the row data, and a file trailer. Headers and data are in network byte order.

- **File Header** — The file header consists of 15 bytes of fixed fields, followed by a variable-length header extension area. The fixed fields are:
 - **Signature** — 11-byte sequence `PGCOPY\n377\r\n0` — note that the zero byte is a required part of the signature. (The signature is designed to allow easy identification of files that have been munged by a non-8-bit-clean transfer. This signature will be changed by end-of-line-translation filters, dropped zero bytes, dropped high bits, or parity changes.)
 - **Flags field** — 32-bit integer bit mask to denote important aspects of the file format. Bits are numbered from 0 (LSB) to 31 (MSB). Note that this field is stored in network byte order (most significant byte first), as are all the integer fields used in the file format. Bits 16-31 are reserved to denote critical file format issues; a reader should abort if it finds an unexpected bit set in this range. Bits 0-15 are reserved to signal backwards-compatible format issues; a reader should simply ignore any unexpected bits set in this range. Currently only one flag is defined, and the rest must be zero (Bit 16: 1 if data has OIDs, 0 if not).
 - **Header extension area length** — 32-bit integer, length in bytes of remainder of header, not including self. Currently, this is zero, and the first tuple follows immediately. Future changes to the format might allow additional data to be present in the header. A reader should silently skip over any header extension data it does not know what to do with. The header extension area is envisioned to contain a sequence of self-identifying chunks. The flags field is not intended to tell readers what is in the extension area. Specific design of header extension contents is left for a later release.
- **Tuples** — Each tuple begins with a 16-bit integer count of the number of fields in the tuple. (Presently, all tuples in a table will have the same count, but that might not always be true.) Then, repeated for each field in the tuple, there is a 32-bit length word followed by that many bytes of field data. (The length word does not include itself, and can be zero.) As a special case, -1 indicates a NULL field value. No value bytes follow in the NULL case.

There is no alignment padding or any other extra data between fields.

Presently, all data values in a COPY BINARY file are assumed to be in binary format (format code one). It is anticipated that a future extension may add a header field that allows per-column format codes to be specified.

If OIDs are included in the file, the OID field immediately follows the field-count word. It is a normal field except that it's not included in the field-count. In particular it has a length word — this will allow handling of 4-byte vs. 8-byte OIDs without too much pain, and will allow OIDs to be shown as null if that ever proves desirable.

- **File Trailer** — The file trailer consists of a 16-bit integer word containing `-1`. This is easily distinguished from a tuple's field-count word. A reader should report an error if a field-count word is neither `-1` nor the expected number of columns. This provides an extra check against somehow getting out of sync with the data.

Examples

Copy a table to the client using the vertical bar (`|`) as the field delimiter:

```
COPY country TO STDOUT WITH DELIMITER '|';
```

Copy data from a file into the `country` table:

```
COPY country FROM '/home/usr1/sql/country_data' WITH
DELIMITER '|' NEWLINE 'LF';
```

Copy into a file just the countries whose names start with 'A':

```
COPY (SELECT * FROM country WHERE country_name LIKE 'A%') TO
'/home/usr1/sql/a_list_countries.copy';
```

Create an error table called `err_sales` to use with single row error isolation mode:

```
CREATE TABLE err_sales ( cmdtime timestamptz, relname
text, filename text, linenum int, bytenum int, errmsg
text, rawdata text, rawbytes bytea )
DISTRIBUTED RANDOMLY;
```

Copy data from a file into the `sales` table using single row error isolation mode:

```
COPY sales FROM '/home/usr1/sql/sales_data' LOG ERRORS INTO
err_sales SEGMENT REJECT LIMIT 10 ROWS;
```

Compatibility

There is no COPY statement in the SQL standard.

See Also

[CREATE EXTERNAL TABLE](#)

CREATE AGGREGATE

Defines a new aggregate function.

Synopsis

```
CREATE AGGREGATE name (input_data_type [ , ... ])
    ( SFUNC = sfunc,
      STYPE = state_data_type
      [, PREFUNC = prefunc]
      [, FINALFUNC = ffunc]
      [, INITCOND = initial_condition]
      [, SORTOP = sort_operator] )
```

Description

CREATE AGGREGATE defines a new aggregate function. Some basic and commonly-used aggregate functions such as count, min, max, sum, avg and so on are already provided in Greenplum Database. If one defines new types or needs an aggregate function not already provided, then CREATE AGGREGATE can be used to provide the desired features.

An aggregate function is identified by its name and input data type(s). Two aggregates in the same schema can have the same name if they operate on different input types. The name and input data type(s) of an aggregate must also be distinct from the name and input data type(s) of every ordinary function in the same schema.

An aggregate function is made from one, two or three ordinary functions (all of which must be IMMUTABLE functions): a state transition function *sfunc*, an optional preliminary segment-level calculation function *prefunc*, and an optional final calculation function *ffunc*. These are used as follows:

```
sfunc( internal-state, next-data-values ) ---> next-internal-state
prefunc( internal-state, internal-state ) ---> next-internal-state
ffunc( internal-state ) ---> aggregate-value
```

Greenplum Database creates a temporary variable of data type *stype* to hold the current internal state of the aggregate. At each input row, the aggregate argument value(s) are calculated and the state transition function is invoked with the current state value and the new argument value(s) to calculate a new internal state value. After all the rows have been processed, the final function is invoked once to calculate the aggregate's return value. If there is no final function then the ending state value is returned as-is.

An aggregate function may provide an initial condition, that is, an initial value for the internal state value. This is specified and stored in the database as a value of type text, but it must be a valid external representation of a constant of the state value data type. If it is not supplied then the state value starts out null.

If the state transition function is declared strict, then it cannot be called with null inputs. With such a transition function, aggregate execution behaves as follows. Rows with any null input values are ignored (the function is not called and the previous state

value is retained). If the initial state value is null, then at the first row with all-nonnull input values, the first argument value replaces the state value, and the transition function is invoked at subsequent rows with all-nonnull input values. This is handy for implementing aggregates like `max`. Note that this behavior is only available when `state_data_type` is the same as the first `input_data_type`. When these types are different, you must supply a nonnull initial condition or use a nonstrict transition function.

If the state transition function is not strict, then it will be called unconditionally at each input row, and must deal with null inputs and null transition values for itself. This allows the aggregate author to have full control over the aggregate's handling of null values.

If the final function is declared strict, then it will not be called when the ending state value is null; instead a null result will be returned automatically. (Of course this is just the normal behavior of strict functions.) In any case the final function has the option of returning a null value. For example, the final function for `avg` returns null when it sees there were zero input rows.

Single argument aggregate functions, such as `min` or `max`, can sometimes be optimized by looking into an index instead of scanning every input row. If this aggregate can be so optimized, indicate it by specifying a sort operator. The basic requirement is that the aggregate must yield the first element in the sort ordering induced by the operator; in other words

```
SELECT agg(col) FROM tab;
```

must be equivalent to:

```
SELECT col FROM tab ORDER BY col USING sortop LIMIT 1;
```

Further assumptions are that the aggregate ignores null inputs, and that it delivers a null result if and only if there were no non-null inputs. Ordinarily, a data type's `<` operator is the proper sort operator for `MIN`, and `>` is the proper sort operator for `MAX`. Note that the optimization will never actually take effect unless the specified operator is the “less than” or “greater than” strategy member of a B-tree index operator class.

Parameters

name

The name (optionally schema-qualified) of the aggregate function to create.

input_data_type

An input data type on which this aggregate function operates. To create a zero-argument aggregate function, write `*` in place of the list of input data types. An example of such an aggregate is `count(*)`.

sfunc

The name of the state transition function to be called for each input row. For an N-argument aggregate function, the *sfunc* must take N+1 arguments, the first being of type `state_data_type` and the rest matching the declared input data type(s) of

the aggregate. The function must return a value of type *state_data_type*. This function takes the current state value and the current input data value(s), and returns the next state value.

state_data_type

The data type for the aggregate's state value.

prefunc

The name of a preliminary aggregation function. This is a function of two arguments, both of type *state_data_type*. It must return a value of *state_data_type*. A preliminary function takes two transition state values and returns a new transition state value representing the combined aggregation. In Greenplum Database, if the result of the aggregate function is computed in a segmented fashion, the preliminary aggregation function is invoked on the individual internal states in order to combine them into an ending internal state.

ffunc

The name of the final function called to compute the aggregate's result after all input rows have been traversed. The function must take a single argument of type *state_data_type*. The return data type of the aggregate is defined as the return type of this function. If *ffunc* is not specified, then the ending state value is used as the aggregate's result, and the return type is *state_data_type*.

initial_condition

The initial setting for the state value. This must be a string constant in the form accepted for the data type *state_data_type*. If not specified, the state value starts out null.

sort_operator

The associated sort operator for a MIN- or MAX-like aggregate. This is just an operator name (possibly schema-qualified). The operator is assumed to have the same input data types as the aggregate (which must be a single-argument aggregate).

Notes

The ordinary functions used to define a new aggregate function must be defined first. Note that in this release of Greenplum Database, it is required that the *sfunc*, *ffunc*, and *prefunc* functions used to create the aggregate are defined as IMMUTABLE.

Any compiled code (shared library files) for custom functions must be placed in the same location on every host in your Greenplum Database array (master and all segments). This location must also be in the `LD_LIBRARY_PATH` so that the server can locate the files.

Examples

Create a sum of cubes aggregate:

```
CREATE FUNCTION scube_accum(numeric, numeric) RETURNS
numeric
```

```

AS 'select $1 + $2 * $2 * $2'
LANGUAGE SQL
IMMUTABLE
RETURNS NULL ON NULL INPUT;
CREATE AGGREGATE scube(numeric) (
  SFUNC = scube_accum,
  STYPE = numeric,
  INITCOND = 0 );

```

To test this aggregate:

```

CREATE TABLE x(a INT);
INSERT INTO x VALUES (1), (2), (3);
SELECT scube(a) FROM x;

```

Correct answer for reference:

```

SELECT sum(a*a*a) FROM x;

```

Compatibility

CREATE AGGREGATE is a Greenplum Database language extension. The SQL standard does not provide for user-defined aggregate functions.

See Also

[ALTER AGGREGATE](#), [DROP AGGREGATE](#), [CREATE FUNCTION](#)

CREATE CAST

Defines a new cast.

Synopsis

```
CREATE CAST (sourcetype AS targettype)
    WITH FUNCTION funcname (argtypes)
    [AS ASSIGNMENT | AS IMPLICIT]

CREATE CAST (sourcetype AS targettype) WITHOUT FUNCTION
    [AS ASSIGNMENT | AS IMPLICIT]
```

Description

`CREATE CAST` defines a new cast. A cast specifies how to perform a conversion between two data types. For example,

```
SELECT CAST(42 AS text);
```

converts the integer constant `42` to type `text` by invoking a previously specified function, in this case `text(int4)`. If no suitable cast has been defined, the conversion fails.

Two types may be binary compatible, which means that they can be converted into one another without invoking any function. This requires that corresponding values use the same internal representation. For instance, the types `text` and `varchar` are binary compatible.

By default, a cast can be invoked only by an explicit cast request, that is an explicit `CAST(x AS typename)` or `x::typename` construct.

If the cast is marked `AS ASSIGNMENT` then it can be invoked implicitly when assigning a value to a column of the target data type. For example, supposing that `foo.f1` is a column of type `text`, then

```
INSERT INTO foo (f1) VALUES (42);
```

will be allowed if the cast from type `integer` to type `text` is marked `AS ASSIGNMENT`, otherwise not. The term *assignment cast* is typically used to describe this kind of cast.

If the cast is marked `AS IMPLICIT` then it can be invoked implicitly in any context, whether assignment or internally in an expression. The term *implicit cast* is typically used to describe this kind of cast. For example, since `||` takes `text` operands,

```
SELECT 'The time is ' || now();
```

will be allowed only if the cast from type `timestamp` to `text` is marked `AS IMPLICIT`. Otherwise, it will be necessary to write the cast explicitly, for example

```
SELECT 'The time is ' || CAST(now() AS text);
```

It is wise to be conservative about marking casts as implicit. An overabundance of implicit casting paths can cause Greenplum Database to choose surprising interpretations of commands, or to be unable to resolve commands at all because there

are multiple possible interpretations. A good rule of thumb is to make a cast implicitly invocable only for information-preserving transformations between types in the same general type category. For example, the cast from `int2` to `int4` can reasonably be implicit, but the cast from `float8` to `int4` should probably be assignment-only. Cross-type-category casts, such as `text` to `int4`, are best made explicit-only.

To be able to create a cast, you must own the source or the target data type. To create a binary-compatible cast, you must be superuser.

Parameters

sourcetype

The name of the source data type of the cast.

targettype

The name of the target data type of the cast.

funcname (argtypes)

The function used to perform the cast. The function name may be schema-qualified. If it is not, the function will be looked up in the schema search path. The function's result data type must match the target type of the cast.

Cast implementation functions may have one to three arguments. The first argument type must be identical to the cast's source type. The second argument, if present, must be type `integer`; it receives the type modifier associated with the destination type, or `-1` if there is none. The third argument, if present, must be type `boolean`; it receives `true` if the cast is an explicit cast, `false` otherwise. The SQL specification demands different behaviors for explicit and implicit casts in some cases. This argument is supplied for functions that must implement such casts. It is not recommended that you design your own data types this way.

Ordinarily a cast must have different source and target data types. However, it is allowed to declare a cast with identical source and target types if it has a cast implementation function with more than one argument. This is used to represent type-specific length coercion functions in the system catalogs. The named function is used to coerce a value of the type to the type modifier value given by its second argument. (Since the grammar presently permits only certain built-in data types to have type modifiers, this feature is of no use for user-defined target types.)

When a cast has different source and target types and a function that takes more than one argument, it represents converting from one type to another and applying a length coercion in a single step. When no such entry is available, coercion to a type that uses a type modifier involves two steps, one to convert between data types and a second to apply the modifier.

WITHOUT FUNCTION

Indicates that the source type and the target type are binary compatible, so no function is required to perform the cast.

AS ASSIGNMENT

Indicates that the cast may be invoked implicitly in assignment contexts.

AS IMPLICIT

Indicates that the cast may be invoked implicitly in any context.

Notes

Note that in this release of Greenplum Database, user-defined functions used in a user-defined cast must be defined as `IMMUTABLE`. Any compiled code (shared library files) for custom functions must be placed in the same location on every host in your Greenplum Database array (master and all segments). This location must also be in the `LD_LIBRARY_PATH` so that the server can locate the files.

Remember that if you want to be able to convert types both ways you need to declare casts both ways explicitly.

It is recommended that you follow the convention of naming cast implementation functions after the target data type, as the built-in cast implementation functions are named. Many users are used to being able to cast data types using a function-style notation, that is `typename(x)`.

Examples

To create a cast from type `text` to type `int4` using the function `int4(text)` (This cast is already predefined in the system.):

```
CREATE CAST (text AS int4) WITH FUNCTION int4(text);
```

Compatibility

The `CREATE CAST` command conforms to the SQL standard, except that SQL does not make provisions for binary-compatible types or extra arguments to implementation functions. `AS IMPLICIT` is a Greenplum Database extension, too.

See Also

[CREATE FUNCTION](#), [CREATE TYPE](#), [DROP CAST](#)

CREATE CONVERSION

Defines a new encoding conversion.

Synopsis

```
CREATE [DEFAULT] CONVERSION name FOR source_encoding TO
dest_encoding FROM funcname
```

Description

`CREATE CONVERSION` defines a new conversion between character set encodings. Conversion names may be used in the `convert` function to specify a particular encoding conversion. Also, conversions that are marked `DEFAULT` can be used for automatic encoding conversion between client and server. For this purpose, two conversions, from encoding A to B and from encoding B to A, must be defined.

To create a conversion, you must have `EXECUTE` privilege on the function and `CREATE` privilege on the destination schema.

Parameters

DEFAULT

Indicates that this conversion is the default for this particular source to destination encoding. There should be only one default encoding in a schema for the encoding pair.

name

The name of the conversion. The conversion name may be schema-qualified. If it is not, the conversion is defined in the current schema. The conversion name must be unique within a schema.

source_encoding

The source encoding name.

dest_encoding

The destination encoding name.

funcname

The function used to perform the conversion. The function name may be schema-qualified. If it is not, the function will be looked up in the path. The function must have the following signature:

```
conv_proc (
    integer, -- source encoding ID
    integer, -- destination encoding ID
    cstring, -- source string (null terminated C string)
    internal, -- destination (fill with a null terminated C string)
    integer -- source string length
```

```
) RETURNS void;
```

Notes

Note that in this release of Greenplum Database, user-defined functions used in a user-defined conversion must be defined as `IMMUTABLE`. Any compiled code (shared library files) for custom functions must be placed in the same location on every host in your Greenplum Database array (master and all segments). This location must also be in the `LD_LIBRARY_PATH` so that the server can locate the files.

Examples

To create a conversion from encoding *UTF8* to *LATIN1* using *myfunc*:

```
CREATE CONVERSION myconv FOR 'UTF8' TO 'LATIN1' FROM myfunc;
```

Compatibility

There is no `CREATE CONVERSION` statement in the SQL standard.

See Also

[ALTER CONVERSION](#), [CREATE FUNCTION](#), [DROP CONVERSION](#)

CREATE DATABASE

Creates a new database.

Synopsis

```
CREATE DATABASE name [ [WITH] [OWNER [=] dbowner]
                    [TEMPLATE [=] template]
                    [ENCODING [=] encoding]
                    [TABLESPACE [=] tablespace]
                    [CONNECTION LIMIT [=] connlimit ] ]
```

Description

`CREATE DATABASE` creates a new database. To create a database, you must be a superuser or have the special `CREATEDB` privilege.

The creator becomes the owner of the new database by default. Superusers can create databases owned by other users by using the `OWNER` clause. They can even create databases owned by users with no special privileges. Non-superusers with `CREATEDB` privilege can only create databases owned by themselves.

By default, the new database will be created by cloning the standard system database *template1*. A different template can be specified by writing `TEMPLATE name`. In particular, by writing `TEMPLATE template0`, you can create a clean database containing only the standard objects predefined by Greenplum Database. This is useful if you wish to avoid copying any installation-local objects that may have been added to *template1*.

Parameters

name

The name of a database to create.

dbowner

The name of the database user who will own the new database, or `DEFAULT` to use the default owner (the user executing the command).

template

The name of the template from which to create the new database, or `DEFAULT` to use the default template (*template1*).

encoding

Character set encoding to use in the new database. Specify a string constant (such as `'SQL_ASCII'`), an integer encoding number, or `DEFAULT` to use the default encoding. See “[Character Set Support](#)” on page 52.

tablespace

The name of the tablespace that will be associated with the new database, or `DEFAULT` to use the template database's tablespace. This tablespace will be the default tablespace used for objects created in this database.

connlimit

Warning: Setting a connection limit at the database level cannot be enforced in Greenplum Database and may cause queries to fail. Leave this set at the default of `-1` (no limit). Connection limits may only be set at the system level in Greenplum Database. See “[Limiting Concurrent Connections](#)” on page 75 for more information.

Notes

`CREATE DATABASE` cannot be executed inside a transaction block.

When you copy a database by specifying its name as the template, no other sessions can be connected to the template database while it is being copied. New connections to the template database are locked out until `CREATE DATABASE` completes.

The `CONNECTION LIMIT` is not enforced against superusers.

Examples

To create a new database:

```
CREATE DATABASE gpdb;
```

To create a database *sales* owned by user *salesapp* with a default tablespace of *salesspace*:

```
CREATE DATABASE sales OWNER salesapp TABLESPACE salesspace;
```

To create a database *music* which supports the ISO-8859-1 character set:

```
CREATE DATABASE music ENCODING 'LATIN1';
```

Compatibility

There is no `CREATE DATABASE` statement in the SQL standard. Databases are equivalent to catalogs, whose creation is implementation-defined.

See Also

[ALTER DATABASE](#), [DROP DATABASE](#)

CREATE DOMAIN

Defines a new domain.

Synopsis

```
CREATE DOMAIN name [AS] data_type [DEFAULT expression]
    [CONSTRAINT constraint_name
    | NOT NULL | NULL
    | CHECK (expression) [...]]
```

Description

`CREATE DOMAIN` creates a new domain. A domain is essentially a data type with optional constraints (restrictions on the allowed set of values). The user who defines a domain becomes its owner. The domain name must be unique among the data types and domains existing in its schema.

Domains are useful for abstracting common constraints on fields into a single location for maintenance. For example, several tables might contain email address columns, all requiring the same `CHECK` constraint to verify the address syntax. It is easier to define a domain rather than setting up a column constraint for each table that has an email column.

Parameters

name

The name (optionally schema-qualified) of a domain to be created.

data_type

The underlying data type of the domain. This may include array specifiers.

DEFAULT *expression*

Specifies a default value for columns of the domain data type. The value is any variable-free expression (but subqueries are not allowed). The data type of the default expression must match the data type of the domain. If no default value is specified, then the default value is the null value. The default expression will be used in any insert operation that does not specify a value for the column. If a default value is defined for a particular column, it overrides any default associated with the domain. In turn, the domain default overrides any default value associated with the underlying data type.

CONSTRAINT *constraint_name*

An optional name for a constraint. If not specified, the system generates a name.

NOT NULL

Values of this domain are not allowed to be null.

NULL

Values of this domain are allowed to be null. This is the default. This clause is only intended for compatibility with nonstandard SQL databases. Its use is discouraged in new applications.

CHECK (*expression*)

CHECK clauses specify integrity constraints or tests which values of the domain must satisfy. Each constraint must be an expression producing a Boolean result. It should use the key word `VALUE` to refer to the value being tested. Currently, CHECK expressions cannot contain subqueries nor refer to variables other than `VALUE`.

Examples

Create the `us_zip_code` data type. A regular expression test is used to verify that the value looks like a valid US zip code.

```
CREATE DOMAIN us_zip_code AS TEXT CHECK
  ( VALUE ~ '^\\d{5}$' OR VALUE ~ '^\\d{5}-\\d{4}$' );
```

Compatibility

CREATE DOMAIN conforms to the SQL standard.

See Also

ALTER DOMAIN, DROP DOMAIN

CREATE EXTERNAL TABLE

Defines a new external table or web table.

Synopsis

```
CREATE EXTERNAL TABLE table_name
  ( column_name data_type [, ...] | LIKE other_table )
  LOCATION ('file://seghost[:port]/path/file' [, ...])
    | ('gpfdist://filehost[:port]/file_pattern' [, ...])
  FORMAT 'TEXT'
    [( [DELIMITER [AS] 'delimiter' | 'OFF']
      [NULL [AS] 'null string']
      [ESCAPE [AS] 'escape' | 'OFF']
      [NEWLINE [ AS ] 'LF' | 'CR' | 'CRLF']
      [FILL MISSING FIELDS] )]
  | 'CSV'
    [( [HEADER]
      [QUOTE [AS] 'quote']
      [DELIMITER [AS] 'delimiter']
      [NULL [AS] 'null string']
      [FORCE NOT NULL column [, ...]]
      [ESCAPE [AS] 'escape']
      [NEWLINE [ AS ] 'LF' | 'CR' | 'CRLF']
      [FILL MISSING FIELDS] )]
  [ ENCODING 'encoding' ]
  [ [LOG ERRORS INTO error_table] SEGMENT REJECT LIMIT count
    [ROWS | PERCENT] ]

CREATE EXTERNAL WEB TABLE table_name
  ( column_name data_type [, ...] | LIKE other_table )
  LOCATION ('http://webhost[:port]/path/file' [, ...])
  | EXECUTE 'command' [ON ALL
    | MASTER
    | number_of_segments
    | HOST ['segment_hostname']
    | SEGMENT segment_id ]
  FORMAT 'TEXT'
    [( [DELIMITER [AS] 'delimiter' | 'OFF']
      [NULL [AS] 'null string']
      [ESCAPE [AS] 'escape' | 'OFF']
      [NEWLINE [ AS ] 'LF' | 'CR' | 'CRLF']
      [FILL MISSING FIELDS] )]
  | 'CSV'
    [( [HEADER]
      [QUOTE [AS] 'quote']
      [DELIMITER [AS] 'delimiter']
      [NULL [AS] 'null string']
      [FORCE NOT NULL column [, ...]]
```

```

[ESCAPE [AS] 'escape']
[NEWLINE [ AS ] 'LF' | 'CR' | 'CRLF']
[FILL MISSING FIELDS] )]
[ ENCODING 'encoding' ]
[ [LOG ERRORS INTO error_table] SEGMENT REJECT LIMIT count
[ROWS | PERCENT] ]

```

Description

`CREATE EXTERNAL TABLE` or `CREATE EXTERNAL WEB TABLE` creates a new external table or web table in Greenplum Database. Once an external or web table is defined, its data can be queried directly (and in parallel) using SQL commands. You can, for example, select, join, or sort external or web table data. You can also create views for external or web tables. However external and web tables are read-only. DML operations (`UPDATE`, `INSERT`, `DELETE`, or `TRUNCATE`) are not allowed, and you cannot create indexes on external or web tables.

An external or web table definition can be thought of as a view that allows running any SQL query against external data sources without requiring that data to first be loaded into the database. External and web tables provide an easy way to perform basic extraction, transformation, and loading (ETL) tasks that are common in data warehousing. External and web table files are read in parallel by the Greenplum Database segment instances, so they also provide a means for fast data loading.

The main difference between external tables and web tables are their data sources. External tables access static flat files, whereas web tables access dynamic data sources — either on a web server or by executing OS commands or scripts. When a query is planned using an external table, the external table is considered rescannable since the data is thought to be static for the course of the query. For web tables, the data is not rescannable because there is the possibility that the data could change during the course of the query's execution.

About External Tables

`CREATE EXTERNAL TABLE` creates a regular external table definition in Greenplum Database. An external table allows you to access flat files as though they were a regular database table. External table data is considered *static* (meaning the data does not change midstream during the execution of a query). This allows the query planner to choose plans that allow for rescanning of the external table data.

You may specify multiple external data sources or URIs (uniform resource identifiers) with the `LOCATION` clause — up to the number of *primary* segment instances in your Greenplum Database array. Each URI points to an external data file or data source. These URIs do not need to exist prior to defining an external table (`CREATE EXTERNAL TABLE` does not validate the URIs specified). However you will get an error if they cannot be found when querying the external table.

There are two protocols that you can use to access the external table data sources. You may use one of the following protocols per `CREATE EXTERNAL TABLE` statement (cannot mix protocols):

- **gpfdist** — If using the `gpfdist://` protocol, you must have the Greenplum file distribution program (`gpfdist`) running on the host where the external data files reside. This program points to a given directory on the file host and serves external data files to all Greenplum Database segments in parallel. If files are compressed using `gzip` or `bzip2` (have a `.gz` or `.bz2` file extension), `gpfdist` will uncompress the files automatically (provided that `gunzip` or `bunzip2` is in your path).

All primary segments access the external file(s) in parallel regardless of how many URIs you specify when defining the external table. You can use multiple `gpfdist` data sources in a `CREATE EXTERNAL TABLE` statement to scale the scan performance of the external table.

When specifying which files to get using `gpfdist`, you can use the wildcard character (`*`) or other C-style pattern matching to denote multiple files. The files specified are assumed to be relative to the directory from which `gpfdist` is serving files (the directory specified when you started the `gpfdist` program).

`gpfdist` is located in `$GPHOME/bin` on your Greenplum Database master host. See the `gpfdist` reference documentation in the Greenplum Data for more information on using this file distribution program with external tables.

- **file** — If using the `file://` protocol the external data file(s) must reside on a segment host in a location accessible by the Greenplum super user (`gpadmin`). The number of URIs specified corresponds to the number of segment instances that will work in parallel to access the external table. So for example, if you have a Greenplum Database system with 8 primary segments and you specify 2 external files, only 2 of the 8 segments will access the external table in parallel at query time. The number of external files per segment host cannot exceed the number of primary segment instances on that host. For example, if your array has 4 primary segment instances per segment host, you may place 4 external files on each segment host. Also, the host name used in the URI must match the segment host name as registered in the `gp_configuration` system catalog table.

The `FORMAT` clause is used to describe how the external table files are formatted. The files can be in delimited text (`TEXT`) or comma separated values (`CSV`) format, similar to the formatting options available with the PostgreSQL `COPY` command. If the data in the file does not use the default column delimiter, escape character, null string and so on, you must specify the additional formatting options so that the data in the external file is read correctly by Greenplum Database.

About Web Tables

`CREATE EXTERNAL WEB TABLE` creates a web table definition in Greenplum Database. A web table is a type of external table that allows you to access dynamic data sources as though they were a regular database table. Web table data is considered *dynamic* (meaning the data could possibly change midstream during the execution of a query). Therefore, the query planner must choose plans that do not allow for rescanning of the web table data.

There are two forms of defining a web table. You may use one of the following forms per `CREATE EXTERNAL WEB TABLE` statement (cannot mix the two forms):

- **Web URLs.** Specify the `LOCATION` of files on a web server using the `http://` protocol. The web data file(s) must reside on a web server that is accessible by the Greenplum segment hosts. The number of URLs specified corresponds to the number of segment instances that will work in parallel to access the web table. So for example, if you have a Greenplum Database system with 8 primary segments and you specify 2 external files, only 2 of the 8 segments will access the web table in parallel at query runtime.
- **OS Command.** Specify a shell command or script to `EXECUTE` on one or more segments, the output of which will comprise the web table's data at the time of access. A web table defined with an `EXECUTE` clause will execute the given OS shell command or script on the specified segment host or hosts. By default, the command is executed by all active segment instances on all segment hosts. For example, if each segment host has four primary segment instances running, the command will be executed four times per segment host. You can optionally limit the number of segment instances that execute the web table command.
Web table data is comprised of the output of the command at the time the web table statement is executed. All segment instances included in the web table definition (as specified by the `ON` clause) execute the command in parallel.

Disabling EXECUTE Web Tables

Web tables that execute OS commands or scripts have a certain security risk associated with them. Some database administrators may decide that they do not want their Greenplum Database systems exposed to this functionality. If this is the case, you can disable the use of `EXECUTE` in external web table definitions by setting the following server configuration parameters in your master `postgresql.conf` file:

```
gp_external_enable_exec = off
```

Web Table Environment Variables

If you use environment variables in external web table commands (such as `$PATH`), keep in mind that the command is executed from within the database and not from a login shell. Therefore the `.bashrc` or `.profile` of the current user will not be sourced. However, you can set desired environment variables from within the `EXECUTE` clause of your external web table definition, for example:

```
CREATE EXTERNAL WEB TABLE blah (blah text) EXECUTE 'export
BLAH=blah-blah-blah; echo $BLAH' FORMAT 'TEXT';
SELECT * FROM blah;
      blah
-----
blah-blah-blah
blah-blah-blah
(2 rows)
```

The following additional Greenplum Database variables are also available for use in OS commands executed by an external web table. These variables are set as environment variables in the shell that executes the command(s). They can be used to identify a set of requests made by a web table statement across the Greenplum Database array of hosts and segment instances.

Table A.1 External Web Table Variables

Variable	Description
\$GP_CID	Command count of the session executing the external web table statement.
\$GP_DATABASE	The database that the external web table definition resides in.
\$GP_DATE	The date the external web table command was executed.
\$GP_MASTER_HOST	The host name of the Greenplum master host from which the external web table statement was dispatched.
\$GP_MASTER_PORT	The port number of the Greenplum master instance from which the external web table statement was dispatched.
\$GP_SEG_DATADIR	The location of the data directory of the segment instance executing the external web table command.
\$GP_SEG_PG_CONF	The location of the <code>postgresql.conf</code> file of the segment instance executing the external web table command.
\$GP_SEG_PORT	The port number of the segment instance executing the external web table command.
\$GP_SEGMENT_COUNT	The total number of primary segment instances in the Greenplum Database system.
\$GP_SEGMENT_ID	The ID number of the segment instance executing the external web table command (same as <code>dbid</code> in <code>gp_configuration</code>).
\$GP_SESSION_ID	The database session identifier number associated with the external web table statement.
\$GP_SN	Serial number of the external web table scan node in the query plan of the external table statement.
\$GP_TIME	The time the external web table command was executed.
\$GP_USER	The database user executing the external web table statement.
\$GP_XID	The transaction ID of the external web table statement.

External Tables and Single Row Error Isolation

The most common use of external tables is selecting data from them to load into regular database tables. This is typically done by issuing a `CREATE TABLE AS SELECT` or `INSERT INTO SELECT` command, where the `SELECT` statement queries external table data. By default, if the external table data contains an error, the entire command fails and no data is loaded into the target database table. To isolate data errors in external table data while still loading correctly formatted rows, you can define an external table with a `SEGMENT REJECT LIMIT` clause. The reject limit *count* can be specified as number of `ROWS` (the default) or a `PERCENT` of total rows (1-100).

When `SEGMENT REJECT LIMIT` is used, then the external data will be scanned in single row error isolation mode. This can be helpful in isolating errors when loading data from an external table using `CREATE TABLE AS SELECT` or `INSERT INTO SELECT`. In this release, single row error isolation mode only applies to external data rows with format errors — for example, extra or missing attributes, attributes of a wrong data type, or invalid client encoding sequences. Constraint errors such as violation of a `NOT NULL`, `CHECK`, or `UNIQUE` constraint will still be handled in ‘all-or-nothing’ input mode. The user can specify the number or percentage of error rows acceptable (on a per-segment basis), after which the entire external table operation will be aborted and no rows will be processed or loaded. Note that the count or percent of error rows is per-segment, not per entire operation. If the per-segment reject limit is not reached, then all rows not containing an error will be processed. If the limit is not reached, all good rows will be processed and any error rows discarded. If you would like to keep error rows for further examination, you can optionally declare an error table using the `LOG ERRORS INTO` clause. Any rows containing a format error would then be logged to the specified error table.

External Tables and Query Planner Statistics

Because the data sources for external and web tables are outside of the database, statistics needed by the query planner are not collected when you run the `ANALYZE` command. You can set some rough statistics for an external or web table by manually editing the system catalog table `pg_class` and specifying the number of rows and database pages (calculated as `data_size / 32K`). By default, `pg_class.reltuples` (number of rows) is set to 1000000 and `pg_class.relpages` (number of pages) is set to 1000 for all external tables and web tables when they are first defined. To change these defaults, you can update these values for your external table in `pg_class` (as the database superuser). For example:

```
UPDATE pg_class SET reltuples=500000, relpages=150 WHERE
relname='my_ext_table';
```

Parameters

WEB

Creates a web table definition in Greenplum Database. A web table is a type of external table that allows you to access dynamic data sources. There are two forms of web tables — those that access files via the `http://` protocol or those that access data by executing OS commands. Web tables are not rescannable during query execution.

table_name

The name of the new external or web table.

column name

The name of a column to create in the external or web table. Unlike regular tables, external and web tables do not have column constraints or default values, so do not specify those.

LIKE *other_table*

The `LIKE` clause specifies a table from which the new external table automatically copies all column names and their data types. If the original table specifies any column constraints or default column values, those will not be copied over to the new external table definition.

data_type

The data type of the column.

LOCATION ('*protocol*://*host*[:*port*]/*path*/*file*' [, ...])

Specifies the URI of the external data source(s) to be used to populate the external table or web table. Regular external tables allow the `gpfdist` or `file` protocols. Web tables allow the `http` protocol. If `port` is omitted for `http` and `gpfdist` protocols, port 8080 is assumed. If using the `gpfdist` protocol, the `path` is relative to the directory from which `gpfdist` is serving files (the directory specified when you started the `gpfdist` program). Also, `gpfdist` can use wildcards (or other C-style pattern matching) to denote multiple files in a directory. For example:

```
gpfdist://filehost:8081/*
gpfdist://masterhost/my_load_file
file://seghost1/dbfast1/external/myfile.txt
http://intranet.mycompany.com/finance/expenses.csv
```

EXECUTE '*command*' [ON ...]

Allowed for web tables only. Specifies the OS command to be executed by the segment instances. The *command* can be a single OS command or a script. The `ON` clause is used to specify which segment instances will execute the given command.

- **ON ALL** is the default. The command will be executed by every active (primary) segment instance on all segment hosts in the Greenplum Database system. If the command executes a script, that script must reside in the same location on all of the segment hosts and be executable by the Greenplum super user (`gadmin`).
- **ON MASTER** runs the command on the master host only.
- **ON *number*** means the command will be executed by the specified number of segments. The particular segments are chosen randomly at runtime by the Greenplum Database system. If the command executes a script, that script must reside in the same location on all of the segment hosts and be executable by the Greenplum superuser (`gadmin`).
- **HOST** means the command will be executed by one segment on each segment host (once per segment host), regardless of the number of active segment instances per host.
- **HOST *segment_hostname*** means the command will be executed by all active (primary) segment instances on the specified segment host.
- **SEGMENT *segment_id*** means the command will be executed only once by the specified segment. You can determine a segment instance's ID by looking at the *content* number in the system catalog table `gp_configuration`. The *content* ID of the Greenplum Database master is always -1.

FORMAT 'TEXT | CSV' (*options*)

Specifies the format of the external or web table data - either plain text (TEXT) or comma separated values (CSV) format.

DELIMITER

Specifies a single ASCII character that separates columns within each row (line) of data. The default is a tab character in TEXT mode, a comma in CSV mode. The delimiter can be set to OFF for special use cases in which unstructured data is loaded into a single-column table.

NULL

Specifies the string that represents a null value. The default is \N (backslash-N) in TEXT mode, and an empty value with no quotations in CSV mode. You might prefer an empty string even in TEXT mode for cases where you do not want to distinguish nulls from empty strings. When using external and web tables, any data item that matches this string will be considered a null value.

ESCAPE

Specifies the single character that is used for C escape sequences (such as \n, \t, \100, and so on) and for escaping data characters that might otherwise be taken as row or column delimiters. Make sure to choose an escape character that is not used anywhere in your actual column data. The default escape character is a \ (backslash), however it is possible to specify any other character to represent an escape. It is also possible to disable escaping by specifying the value 'OFF' as the escape value. This is very useful for data such as web log data that has many embedded backslashes that are not intended to be escapes.

NEWLINE

Specifies the newline used in your data files — LF (Line feed, 0x0A), CR (Carriage return, 0x0D), or CRLF (Carriage return plus line feed, 0x0D 0x0A). If not specified, a Greenplum Database segment will detect the newline type by looking at the first row of data it receives and using the first newline type encountered.

HEADER

For CSV formatted data, specifies that the first line in the data file(s) is a header row (contains the names of the table columns) and should not be included as data for the table. If using multiple flat files, all files must have a header row.

QUOTE

Specifies the quotation character for CSV mode. The default is double-quote (").

FORCE NOT NULL

In CSV mode, processes each specified column as though it were quoted and hence not a NULL value. For the default null string in CSV mode (nothing between two delimiters), this causes missing values to be evaluated as zero-length strings.

FILL MISSING FIELDS

In both `TEXT` and `CSV` mode, specifying `FILL MISSING FIELDS` will set missing trailing field values to `NULL` (instead of reporting an error) when a row of data has missing data fields at the end of a line or row. Blank rows, fields with a `NOT NULL` constraint, and trailing delimiters on a line will still report an error.

ENCODING 'encoding'

Character set encoding to use for the external table. Specify a string constant (such as `'SQL_ASCII'`), an integer encoding number, or `DEFAULT` to use the default client encoding. See “Character Set Support” on page 52.

LOG ERRORS INTO *error_table*

This is an optional clause that may precede a `SEGMENT REJECT LIMIT` clause. It specifies an error table where rows with formatting errors will be logged when running in single row error isolation mode. You can then examine this error table to see error rows that were not loaded (if any). If the *error_table* specified already exists, it will be used. If it does not exist, it will be automatically generated.

SEGMENT REJECT LIMIT *count* [ROWS | PERCENT]

Runs a `SELECT` from an external table operation in single row error isolation mode. If the input rows have format errors they will be discarded provided that the reject limit count is not reached on any Greenplum segment instance during the load operation. The reject limit count can be specified as number of rows (the default) or percentage of total rows (1-100). If `PERCENT` is used, the percentage of rows per segment is calculated based on the parameter `gp_reject_percent_threshold` (default is 300 rows). Constraint errors such as violation of a `NOT NULL`, `CHECK`, or `UNIQUE` constraint will still be handled in ‘all-or-nothing’ mode, meaning a single constraint error will cause the entire external table operation to fail. If the limit is not reached, all good rows will be processed and any error rows discarded.

Notes

There is a system view named `pg_max_external_files` that you can use to determine how many external table files are permitted per external table. This view lists the available file slots per segment host (if using the `file://` protocol). For example:

```
SELECT * FROM pg_max_external_files;
```

If using the `gpfdist://` protocol, the following server configuration parameter can be used to set the maximum number of segment instances that will go to one `gpfdist` file distribution program to get external table data in parallel. The default is 64 segment instances.

```
gp_external_max_segs = <integer>
```

The `gpfdist` program serves files via the `HTTP` protocol. Queries of external tables that use a `LIMIT` clause will break off the `HTTP` connection after retrieving the rows causing an `HTTP` socket error. If using `LIMIT` in queries of external tables that use the `gpfdist://` or `http://` protocols, it is safe to ignore these `HTTP` socket errors from the `gpfdist` or web server — data is still returned to the database client as expected.

During dump/restore operations, only external and web table *definitions* will be backed up and restored. The data sources will not be included.

You can use `CREATE TABLE AS` or `INSERT...SELECT` to load external and web table data into another (non-external) database table, and the data will be loaded in parallel according to the external or web table definition.

If an external table file or web table data source has an error, any operation that reads from that table will fail. Similar to `COPY`, loading from external and web tables is an all or nothing operation.

Examples

Start the `gpfdist` file server program in the background on port `8081` serving files from directory `/var/data/staging`:

```
gpfdist -p 8081 -d /var/data/staging -l /home/gpadmin/log &
```

Create an external table named `ext_customer` using the `gpfdist` protocol and any text formatted files (`*.txt`) found in the `gpfdist` directory. The files are formatted with a pipe (`|`) as the column delimiter and an empty space as null. Also access the external table in single row error isolation mode:

```
CREATE EXTERNAL TABLE ext_customer
  (id int, name text, sponsor text)
  LOCATION ( 'gpfdist://filehost:8081/*.txt' )
  FORMAT 'TEXT' ( DELIMITER '|' NULL ' ')
  LOG ERRORS INTO err_customer SEGMENT REJECT LIMIT 5;
```

Create the same external table definition as above, but with CSV formatted files:

```
CREATE EXTERNAL TABLE ext_customer
  (id int, name text, sponsor text)
  LOCATION ( 'gpfdist://filehost:8081/*.csv' )
  FORMAT 'CSV' ( DELIMITER ',' );
```

Create an external table named `ext_expenses` using the `file` protocol and several CSV formatted files that have a header row:

```
CREATE EXTERNAL TABLE ext_expenses (name text, date date,
amount float4, category text, description text)
LOCATION (
'file://seghost1/dbfast/external/expenses1.csv',
'file://seghost1/dbfast/external/expenses2.csv',
'file://seghost2/dbfast/external/expenses3.csv',
'file://seghost2/dbfast/external/expenses4.csv',
'file://seghost3/dbfast/external/expenses5.csv',
'file://seghost3/dbfast/external/expenses6.csv',
)
FORMAT 'CSV' ( HEADER );
```

Create a web table that executes a script once per segment host:

```
CREATE EXTERNAL WEB TABLE log_output (linenum int, message
text) EXECUTE '/var/load_scripts/get_log_data.sh' ON HOST
FORMAT 'TEXT' (DELIMITER '|');
```

Compatibility

`CREATE EXTERNAL TABLE` is a Greenplum Database extension. The SQL standard makes no provisions for external tables.

See Also

`CREATE TABLE AS`, `CREATE TABLE`, `COPY`, `SELECT INTO`, `INSERT`

CREATE FUNCTION

Defines a new function.

Synopsis

```
CREATE [OR REPLACE] FUNCTION name
  ( [ [argmode] [argname] argtype [, ...] ] )
  [RETURNS rettype]
  { LANGUAGE langname
  | IMMUTABLE | STABLE | VOLATILE
  | CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT
  | [EXTERNAL] SECURITY INVOKER | [EXTERNAL] SECURITY DEFINER
  | AS 'definition'
  | AS 'obj_file', 'link_symbol' } ...
```

Description

CREATE FUNCTION defines a new function. CREATE OR REPLACE FUNCTION will either create a new function, or replace an existing definition.

The name of the new function must not match any existing function with the same argument types in the same schema. However, functions of different argument types may share a name (overloading).

To update the definition of an existing function, use CREATE OR REPLACE FUNCTION. It is not possible to change the name or argument types of a function this way (this would actually create a new, distinct function). Also, CREATE OR REPLACE FUNCTION will not let you change the return type of an existing function. To do that, you must drop and recreate the function. If you drop and then recreate a function, you will have to drop existing objects (rules, views, triggers, and so on) that refer to the old function. Use CREATE OR REPLACE FUNCTION to change a function definition without breaking objects that refer to the function.

For more information about creating functions, see the [User Defined Functions](#) section of the PostgreSQL documentation.

Limited Use of VOLATILE and STABLE Functions

To prevent data from becoming out-of-sync across the segments in Greenplum Database, any function classified as STABLE or VOLATILE cannot be executed at the segment level if it contains SQL or modifies the database in any way. For example, functions such as random() or timeofday() are not allowed to execute on distributed data in Greenplum Database because they could potentially cause inconsistent data between the segment instances.

Also, STABLE and VOLATILE functions cannot be used in UPDATE or DELETE statements at all if mirrors are enabled. For example, the following statement would not be allowed if you had mirror segments because UPDATE and DELETE statements are run on the primary segments and the mirror segments independently. The following statement would cause a mirror and its primary to be out of sync:

```
UPDATE mytable SET id = nextval(myseq);
```

To ensure data consistency, `VOLATILE` and `STABLE` functions can safely be used in statements that are evaluated on and execute from the master. For example, the following statements are always executed on the master (statements without a `FROM` clause):

```
SELECT setval('myseq', 201);
SELECT foo();
```

The following statement executes on the segments (has a `FROM` clause), however it would be allowed provided that the function used in the `FROM` clause simply returns a set of rows:

```
SELECT * FROM foo();
```

One exception to this rule are functions that return a table reference (`rangeFuncs`) or functions that use the `refCursor` data type. These types of functions cannot be used at all in Greenplum Database. However, they are not very commonly used anyways.

Parameters

name

The name (optionally schema-qualified) of the function to create.

argmode

The mode of an argument: either `IN`, `OUT`, or `INOUT`. If omitted, the default is `IN`.

argname

The name of an argument. Some languages (currently only PL/pgSQL) let you use the name in the function body. For other languages the name of an input argument is just extra documentation. But the name of an output argument is significant, since it defines the column name in the result row type. (If you omit the name for an output argument, the system will choose a default column name.)

argtype

The data type(s) of the function's arguments (optionally schema-qualified), if any. The argument types may be base, composite, or domain types, or may reference the type of a table column.

Depending on the implementation language it may also be allowed to specify pseudotypes such as `cstring`. Pseudotypes indicate that the actual argument type is either incompletely specified, or outside the set of ordinary SQL data types.

The type of a column is referenced by writing `tablename.columnname%TYPE`. Using this feature can sometimes help make a function independent of changes to the definition of a table.

rettype

The return data type (optionally schema-qualified). The return type may be a base, composite, or domain type, or may reference the type of a table column. Depending on the implementation language it may also be allowed to specify pseudotypes such as `cstring`. If the function is not supposed to return a value, specify `void` as the return type.

When there are `OUT` or `INOUT` parameters, the `RETURNS` clause may be omitted. If present, it must agree with the result type implied by the output parameters: `RECORD` if there are multiple output parameters, or the same type as the single output parameter.

The `SETOF` modifier indicates that the function will return a set of items, rather than a single item.

The type of a column is referenced by writing `tablename.columnname%TYPE`.

langname

The name of the language that the function is implemented in. May be `SQL`, `C`, `internal`, or the name of a user-defined procedural language. For backward compatibility, the name may be enclosed by single quotes.

IMMUTABLE**STABLE****VOLATILE**

These attributes inform the query optimizer about the behavior of the function. At most one choice may be specified. If none of these appear, `VOLATILE` is the default assumption. Since Greenplum Database currently has limited use of `VOLATILE` functions, if a function is truly `IMMUTABLE`, you must declare it as so to be able to use it without restrictions.

`IMMUTABLE` indicates that the function cannot modify the database and always returns the same result when given the same argument values. It does not do database lookups or otherwise use information not directly present in its argument list. If this option is given, any call of the function with all-constant arguments can be immediately replaced with the function value.

`STABLE` indicates that the function cannot modify the database, and that within a single table scan it will consistently return the same result for the same argument values, but that its result could change across SQL statements. This is the appropriate selection for functions whose results depend on database lookups, parameter values (such as the current time zone), and so on. Also note that the *current_timestamp* family of functions qualify as stable, since their values do not change within a transaction.

`VOLATILE` indicates that the function value can change even within a single table scan, so no optimizations can be made. Relatively few database functions are volatile in this sense; some examples are `random()`, `curval()`, `timeofday()`. But note that any function that has side-effects must be classified volatile, even if its result is quite predictable, to prevent calls from being optimized away; an example is `setval()`.

CALLED ON NULL INPUT
RETURNS NULL ON NULL INPUT
STRICT

CALLED ON NULL INPUT (the default) indicates that the function will be called normally when some of its arguments are null. It is then the function author's responsibility to check for null values if necessary and respond appropriately.

RETURNS NULL ON NULL INPUT or **STRICT** indicates that the function always returns null whenever any of its arguments are null. If this parameter is specified, the function is not executed when there are null arguments; instead a null result is assumed automatically.

[EXTERNAL] SECURITY INVOKER
[EXTERNAL] SECURITY DEFINER

SECURITY INVOKER (the default) indicates that the function is to be executed with the privileges of the user that calls it. **SECURITY DEFINER** specifies that the function is to be executed with the privileges of the user that created it. The key word **EXTERNAL** is allowed for SQL conformance, but it is optional since, unlike in SQL, this feature applies to all functions not just external ones.

definition

A string constant defining the function; the meaning depends on the language. It may be an internal function name, the path to an object file, an SQL command, or text in a procedural language.

obj_file, link_symbol

This form of the **AS** clause is used for dynamically loadable C language functions when the function name in the C language source code is not the same as the name of the SQL function. The string *obj_file* is the name of the file containing the dynamically loadable object, and *link_symbol* is the name of the function in the C language source code. If the link symbol is omitted, it is assumed to be the same as the name of the SQL function being defined. It is recommended to locate shared libraries either relative to `$libdir` (which is located at `$GPHOME/lib`) or through the dynamic library path (set by the `dynamic_library_path` server configuration parameter). This simplifies version upgrades if the new installation is at a different location.

Notes

Any compiled code (shared library files) for custom functions must be placed in the same location on every host in your Greenplum Database array (master and all segments). This location must also be in the `LD_LIBRARY_PATH` so that the server can locate the files. It is recommended to locate shared libraries either relative to `$libdir` (which is located at `$GPHOME/lib`) or through the dynamic library path (set by the `dynamic_library_path` server configuration parameter) on all master segment instances in the Greenplum array.

The full SQL type syntax is allowed for input arguments and return value. However, some details of the type specification (such as the precision field for type *numeric*) are the responsibility of the underlying function implementation and are not recognized or enforced by the `CREATE FUNCTION` command.

Greenplum Database allows function overloading. The same name can be used for several different functions so long as they have distinct argument types. However, the C names of all functions must be different, so you must give overloaded C functions different C names (for example, use the argument types as part of the C names).

Two functions are considered the same if they have the same names and input argument types, ignoring any OUT parameters. Thus for example these declarations conflict:

```
CREATE FUNCTION foo(int) ...
CREATE FUNCTION foo(int, out text) ...
```

When repeated `CREATE FUNCTION` calls refer to the same object file, the file is only loaded once. To unload and reload the file, use the `LOAD` command.

To be able to define a function, the user must have the `USAGE` privilege on the language.

It is often helpful to use dollar quoting to write the function definition string, rather than the normal single quote syntax. Without dollar quoting, any single quotes or backslashes in the function definition must be escaped by doubling them. A dollar-quoted string constant consists of a dollar sign (`$`), an optional tag of zero or more characters, another dollar sign, an arbitrary sequence of characters that makes up the string content, a dollar sign, the same tag that began this dollar quote, and a dollar sign. Inside the dollar-quoted string, single quotes, backslashes, or any character can be used without escaping. The string content is always written literally. For example, here are two different ways to specify the string “Dianne’s horse” using dollar quoting:

```
$$Dianne's horse$$
$SomeTag$Dianne's horse$SomeTag$
```

Examples

A very simple addition function:

```
CREATE FUNCTION add(integer, integer) RETURNS integer
AS 'select $1 + $2;'
LANGUAGE SQL
IMMUTABLE
RETURNS NULL ON NULL INPUT;
```

Increment an integer, making use of an argument name, in PL/pgSQL:

```
CREATE OR REPLACE FUNCTION increment(i integer) RETURNS
integer AS $$
BEGIN
    RETURN i + 1;
END;
$$ LANGUAGE plpgsql;
```

Return a record containing multiple output parameters:

```
CREATE FUNCTION dup(in int, out f1 int, out f2 text)
```

```

        AS $$ SELECT $1, CAST($1 AS text) || ' is text' $$
LANGUAGE SQL;
SELECT * FROM dup(42);

```

You can do the same thing more verbosely with an explicitly named composite type:

```

CREATE TYPE dup_result AS (f1 int, f2 text);
CREATE FUNCTION dup(int) RETURNS dup_result
    AS $$ SELECT $1, CAST($1 AS text) || ' is text' $$
LANGUAGE SQL;
SELECT * FROM dup(42);

```

Compatibility

`CREATE FUNCTION` is defined in SQL:1999 and later. The Greenplum Database version is similar but not fully compatible. The attributes are not portable, neither are the different available languages.

For compatibility with some other database systems, *argmode* can be written either before or after *argname*. But only the first way is standard-compliant.

See Also

[ALTER FUNCTION](#), [DROP FUNCTION](#), [LOAD](#)

CREATE GROUP

Defines a new database role.

Synopsis

```
CREATE GROUP name [ [WITH] option [ ... ] ]
```

where *option* can be:

```

SUPERUSER | NOSUPERUSER
| CREATEDB | NOCREATEDB
| CREATEROLE | NOCREATEROLE
| CREATEUSER | NOCREATEUSER
| INHERIT | NOINHERIT
| LOGIN | NOLOGIN
| [ ENCRYPTED | UNENCRYPTED ] PASSWORD 'password'
| VALID UNTIL 'timestamp'
| IN ROLE rolename [, ...]
| IN GROUP rolename [, ...]
| ROLE rolename [, ...]
| ADMIN rolename [, ...]
| USER rolename [, ...]
| SYSID uid

```

Description

As of Greenplum Database release 2.2, `CREATE GROUP` has been replaced by `CREATE ROLE`, although it is still accepted for backwards compatibility.

Compatibility

There is no `CREATE GROUP` statement in the SQL standard.

See Also

[CREATE ROLE](#)

CREATE INDEX

Defines a new index.

Synopsis

```
CREATE [UNIQUE] INDEX [CONCURRENTLY] name ON table
    [USING method]
    ( {column | (expression)} [opclass] [, ...] )
    [ WITH ( FILLFACTOR = value ) ]
    [TABLESPACE tablespace]
    [WHERE predicate]
```

Description

`CREATE INDEX` constructs an index on the specified table. Indexes are primarily used to enhance database performance (though inappropriate use can result in slower performance).

The key field(s) for the index are specified as column names, or alternatively as expressions written in parentheses. Multiple fields can be specified if the index method supports multicolumn indexes.

An index field can be an expression computed from the values of one or more columns of the table row. This feature can be used to obtain fast access to data based on some transformation of the basic data. For example, an index computed on `upper(col)` would allow the clause `WHERE upper(col) = 'JIM'` to use an index.

Greenplum Database provides the index methods B-tree, bitmap, hash, GiST, and GIN. Users can also define their own index methods, but that is fairly complicated.

When the `WHERE` clause is present, a partial index is created. A partial index is an index that contains entries for only a portion of a table, usually a portion that is more useful for indexing than the rest of the table. For example, if you have a table that contains both billed and unbilled orders where the unbilled orders take up a small fraction of the total table and yet is most often selected, you can improve performance by creating an index on just that portion.

The expression used in the `WHERE` clause may refer only to columns of the underlying table, but it can use all columns, not just the ones being indexed. Subqueries and aggregate expressions are also forbidden in `WHERE`. The same restrictions apply to index fields that are expressions.

All functions and operators used in an index definition must be immutable. Their results must depend only on their arguments and never on any outside influence (such as the contents of another table or a parameter value). This restriction ensures that the behavior of the index is well-defined. To use a user-defined function in an index expression or `WHERE` clause, remember to mark the function `IMMUTABLE` when you create it.

Parameters

UNIQUE

Checks for duplicate values in the table when the index is created and each time data is added. Duplicate entries will generate an error. Unique indexes only apply to B-tree indexes. In Greenplum Database, unique indexes are allowed only if the columns of the index key are the same as (or a superset of) the Greenplum distribution key.

CONCURRENTLY

A concurrent index build creates the index without taking any locks that prevent concurrent inserts, updates, or deletes on the table. A standard index build locks out writes (but not reads) on the table until it is done. When this option is used, Greenplum Database must perform two scans of the table, and in addition it must wait for all existing transactions to terminate. Therefore this method requires more total work than a standard index build and takes significantly longer to complete. However, since it allows normal operations to continue while the index is built, this method is useful for adding new indexes in a production environment. Of course, the extra CPU and I/O load imposed by the index creation may slow other operations. Bitmap indexes cannot be built concurrently.

name

The name of the index to be created. The index is always created in the same schema as its parent table.

table

The name (optionally schema-qualified) of the table to be indexed.

method

The name of the index method to be used. Choices are `btree`, `bitmap`, `hash`, `gist`, and `gin`. The default method is `btree`.

column

The name of a column of the table on which to create the index. Only the B-tree, bitmap, and GiST index methods support multicolumn indexes.

expression

An expression based on one or more columns of the table. The expression usually must be written with surrounding parentheses, as shown in the syntax. However, the parentheses may be omitted if the expression has the form of a function call.

opclass

The name of an operator class. The operator class identifies the operators to be used by the index for that column. For example, a B-tree index on four-byte integers would use the `int4_ops` class (this operator class includes comparison functions for four-byte integers). In practice the default operator class for the column's data type is usually sufficient. The main point of having operator classes is that for some data

types, there could be more than one meaningful ordering. For example, a complex-number data type could be sorted by either absolute value or by real part. We could do this by defining two operator classes for the data type and then selecting the proper class when making an index.

FILLFACTOR

The fillfactor for an index is a percentage that determines how full the index method will try to pack index pages. For B-trees, leaf pages are filled to this percentage during initial index build, and also when extending the index at the right (largest key values). If pages subsequently become completely full, they will be split, leading to gradual degradation in the index's efficiency.

B-trees use a default fillfactor of 90, but any value from 10 to 100 can be selected. If the table is static then fillfactor 100 is best to minimize the index's physical size, but for heavily updated tables a smaller fillfactor is better to minimize the need for page splits. The other index methods use fillfactor in different but roughly analogous ways; the default fillfactor varies between methods.

tablespace

The tablespace in which to create the index. If not specified, the default tablespace is used.

predicate

The constraint expression for a partial index.

Notes

UNIQUE indexes are allowed only if the index columns are the same as (or a superset of) the Greenplum distribution key columns.

Indexes are not used for IS NULL clauses by default. The best way to use indexes in such cases is to create a partial index using an IS NULL predicate.

Prior releases of Greenplum Database also had an R-tree index method. This method has been removed because it had no significant advantages over the GiST method. If USING rtree is specified, CREATE INDEX will interpret it as USING gist.

Examples

To create a B-tree index on the column *title* in the table *films*:

```
CREATE UNIQUE INDEX title_idx ON films (title);
```

To create a bitmap index on the column *gender* in the table *employee*:

```
CREATE INDEX gender_bmp_idx ON employee USING bitmap
(gender);
```

To create an index on the expression *lower(title)*, allowing efficient case-insensitive searches:

```
CREATE INDEX lower_title_idx ON films ((lower(title)));
```

To create an index with non-default fill factor:

```
CREATE UNIQUE INDEX title_idx ON films (title) WITH  
(fillfactor = 70);
```

To create an index on the column *code* in the table *films* and have the index reside in the tablespace *indexspace*:

```
CREATE INDEX code_idx ON films (code) TABLESPACE indexspace;
```

Compatibility

CREATE INDEX is a Greenplum Database language extension. There are no provisions for indexes in the SQL standard. Greenplum Database does not support the concurrent creation of indexes as does PostgreSQL (CONCURRENTLY keyword not supported).

See Also

[ALTER INDEX](#), [DROP INDEX](#), [CREATE TABLE](#), [CREATE OPERATOR CLASS](#)

CREATE LANGUAGE

Defines a new procedural language.

Synopsis

```
CREATE [PROCEDURAL] LANGUAGE name
CREATE [TRUSTED] [PROCEDURAL] LANGUAGE name
    HANDLER call_handler [VALIDATOR valfunction]
```

Description

`CREATE LANGUAGE` registers a new procedural language with a Greenplum database. Subsequently, functions and trigger procedures can be defined in this new language. You must be a superuser to register a new language.

`CREATE LANGUAGE` effectively associates the language name with a call handler that is responsible for executing functions written in that language. For a function written in a procedural language (a language other than C or SQL), the database server has no built-in knowledge about how to interpret the function's source code. The task is passed to a special handler that knows the details of the language. The handler could either do all the work of parsing, syntax analysis, execution, and so on or it could serve as a bridge between Greenplum Database and an existing implementation of a programming language. The handler itself is a C language function compiled into a shared object and loaded on demand, just like any other C function. There are currently three procedural language packages included in the standard Greenplum Database distribution: PL/pgSQL, PL/Perl and PL/Python. A language handler has also been added for PL/R, but the PL/R language package is not pre-installed with Greenplum Database. There is also a package available for PL/Tcl, which can be enabled if needed. See the section on [Procedural Languages](#) in the PostgreSQL documentation for more information.

Users who wish to use the PL/Perl procedural language must make sure that the systems that run Greenplum Database (master and all segments) have a shared version of Perl installed. For 64-bit systems, you will need a 64-bit shared version of Perl. Solaris does not have a 64-bit shared version of Perl by default. Most Linux distributions typically have a shared Perl and Python preinstalled in `/usr/lib64` (or `/usr/lib` on 32-bit systems). Greenplum provides a 64-bit shared version of Perl for both Solaris and Linux platforms. If you need a 64-bit shared Perl install package, download it from the [Greenplum Network](#) Download page.

Users who wish to use the PL/R procedural language must make sure that the systems that run Greenplum Database (master and all segments) have the R language installed and the PL/R package library (`plr.so`) added to their Greenplum installation on all hosts. Greenplum provides compiled packages for R and PL/R that you can install. Contact [Greenplum Customer Support](#) if you need the install packages for R or PL/R.

There are two forms of the `CREATE LANGUAGE` command. In the first form, the user supplies just the name of the desired language, and the Greenplum Database server consults the `pg_pltemplate` system catalog to determine the correct parameters. In the second form, the user supplies the language parameters along with the language name. The second form can be used to create a language that is not defined in `pg_pltemplate`.

When the server finds an entry in the `pg_pltemplate` catalog for the given language name, it will use the catalog data even if the command includes language parameters. This behavior simplifies loading of old dump files, which are likely to contain out-of-date information about language support functions.

Parameters

TRUSTED

Ignored if the server has an entry for the specified language name in `pg_pltemplate`. Specifies that the call handler for the language is safe and does not offer an unprivileged user any functionality to bypass access restrictions. If this key word is omitted when registering the language, only users with the superuser privilege can use this language to create new functions.

PROCEDURAL

This is a noise word.

name

The name of the new procedural language. The language name is case insensitive. The name must be unique among the languages in the database. Built-in support is included for `plpgsql`, `plperl`, `plpython`, `plpythonu`, and `plr`.

HANDLER *call_handler*

Ignored if the server has an entry for the specified language name in `pg_pltemplate`. The name of a previously registered function that will be called to execute the procedural language functions. The call handler for a procedural language must be written in a compiled language such as C with version 1 call convention and registered with Greenplum Database as a function taking no arguments and returning the `language_handler` type, a placeholder type that is simply used to identify the function as a call handler.

VALIDATOR *valfunction*

Ignored if the server has an entry for the specified language name in `pg_pltemplate`. *valfunction* is the name of a previously registered function that will be called when a new function in the language is created, to validate the new function. If no validator function is specified, then a new function will not be checked when it is created. The validator function must take one argument of type `oid`, which will be the OID of the to-be-created function, and will typically return `void`.

A validator function would typically inspect the function body for syntactical correctness, but it can also look at other properties of the function, for example if the language cannot handle certain argument types. To signal an error, the validator function should use the `ereport()` function. The return value of the function is ignored.

Notes

The system catalog `pg_language` records information about the currently installed languages.

To create functions in a procedural language, a user must have the `USAGE` privilege for the language. By default, `USAGE` is granted to `PUBLIC` (everyone) for trusted languages. This may be revoked if desired.

Procedural languages are local to individual databases. However, a language can be installed into the `template1` database, which will cause it to be available automatically in all subsequently-created databases.

The call handler function and the validator function (if any) must already exist if the server does not have an entry for the language in `pg_pltemplate`. But when there is an entry, the functions need not already exist; they will be automatically defined if not present in the database.

Any shared library that implements a language must be located in the same `LD_LIBRARY_PATH` location on all segment hosts in your Greenplum Database array.

Examples

The preferred way of creating any of the standard procedural languages:

```
CREATE LANGUAGE plpgsql;
CREATE LANGUAGE plr;
```

For a language not known in the `pg_pltemplate` catalog:

```
CREATE FUNCTION plsample_call_handler() RETURNS
language_handler
AS '$libdir/plsample'
LANGUAGE C;
CREATE LANGUAGE plsample
HANDLER plsample_call_handler;
```

Compatibility

`CREATE LANGUAGE` is a Greenplum Database extension.

See Also

[ALTER LANGUAGE](#), [CREATE FUNCTION](#), [DROP LANGUAGE](#)

CREATE OPERATOR

Defines a new operator.

Synopsis

```
CREATE OPERATOR name (
    PROCEDURE = funcname
    [, LEFTARG = lefttype] [, RIGHTARG = righttype]
    [, COMMUTATOR = com_op] [, NEGATOR = neg_op]
    [, RESTRICT = res_proc] [, JOIN = join_proc]
    [, HASHES] [, MERGES]
    [, SORT1 = left_sort_op] [, SORT2 = right_sort_op]
    [, LTCMP = less_than_op] [, GTCMP = greater_than_op] )
```

Description

CREATE OPERATOR defines a new operator. The user who defines an operator becomes its owner.

The operator name is a sequence of up to NAMEDATALEN-1 (63 by default) characters from the following list: + - * / < > = ~ ! @ # % ^ & | ` ?

There are a few restrictions on your choice of name:

- -- and /* cannot appear anywhere in an operator name, since they will be taken as the start of a comment.
- A multicharacter operator name cannot end in + or -, unless the name also contains at least one of these characters: ~ ! @ # % ^ & | ` ?

For example, @- is an allowed operator name, but *- is not. This restriction allows Greenplum Database to parse SQL-compliant commands without requiring spaces between tokens.

The operator != is mapped to <> on input, so these two names are always equivalent.

At least one of LEFTARG and RIGHTARG must be defined. For binary operators, both must be defined. For right unary operators, only LEFTARG should be defined, while for left unary operators only RIGHTARG should be defined.

The *funcname* procedure must have been previously defined using CREATE FUNCTION, must be IMMUTABLE, and must be defined to accept the correct number of arguments (either one or two) of the indicated types.

The other clauses specify optional operator optimization clauses. These clauses should be provided whenever appropriate to speed up queries that use the operator. But if you provide them, you must be sure that they are correct. Incorrect use of an optimization clause can result in server process crashes, subtly wrong output, or other unexpected results. You can always leave out an optimization clause if you are not sure about it.

Parameters

name

The (optionally schema-qualified) name of the operator to be defined. Two operators in the same schema can have the same name if they operate on different data types.

funcname

The function used to implement this operator (must be an IMMUTABLE function).

lefttype

The data type of the operator's left operand, if any. This option would be omitted for a left-unity operator.

righttype

The data type of the operator's right operand, if any. This option would be omitted for a right-unity operator.

com_op

The optional `COMMUTATOR` clause names an operator that is the commutator of the operator being defined. We say that operator A is the commutator of operator B if $(x A y)$ equals $(y B x)$ for all possible input values x, y . Notice that B is also the commutator of A. For example, operators `<` and `>` for a particular data type are usually each others commutators, and operator `+` is usually commutative with itself. But operator `-` is usually not commutative with anything. The left operand type of a commutable operator is the same as the right operand type of its commutator, and vice versa. So the name of the commutator operator is all that needs to be provided in the `COMMUTATOR` clause.

neg_op

The optional `NEGATOR` clause names an operator that is the negator of the operator being defined. We say that operator A is the negator of operator B if both return Boolean results and $(x A y)$ equals `NOT (x B y)` for all possible inputs x, y . Notice that B is also the negator of A. For example, `<` and `>=` are a negator pair for most data types. An operator's negator must have the same left and/or right operand types as the operator to be defined, so only the operator name need be given in the `NEGATOR` clause.

res_proc

The optional `RESTRICT` names a restriction selectivity estimation function for the operator. Note that this is a function name, not an operator name. `RESTRICT` clauses only make sense for binary operators that return `boolean`. The idea behind a restriction selectivity estimator is to guess what fraction of the rows in a table will satisfy a `WHERE`-clause condition of the form:

```
column OP constant
```

for the current operator and a particular constant value. This assists the optimizer by giving it some idea of how many rows will be eliminated by `WHERE` clauses that have this form.

You can usually just use one of the following system standard estimator functions for many of your own operators:

```
eqsel for =
neqsel for <>
scalarltsel for < or <=
scalargtsel for > or >=
```

join_proc

The optional `JOIN` clause names a join selectivity estimation function for the operator. Note that this is a function name, not an operator name. `JOIN` clauses only make sense for binary operators that return `boolean`. The idea behind a join selectivity estimator is to guess what fraction of the rows in a pair of tables will satisfy a `WHERE`-clause condition of the form

```
table1.column1 OP table2.column2
```

for the current operator. This helps the optimizer by letting it figure out which of several possible join sequences is likely to take the least work.

You can usually just use one of the following system standard join selectivity estimator functions for many of your own operators:

```
eqjoinsel for =
neqjoinsel for <>
scalarltjoinsel for < or <=
scalargtjoinsel for > or >=
areajoinsel for 2D area-based comparisons
positionjoinsel for 2D position-based comparisons
contjoinsel for 2D containment-based comparisons
```

HASHES

The optional `HASHES` clause tells the system that it is permissible to use the hash join method for a join based on this operator. `HASHES` only makes sense for a binary operator that returns `boolean`. The hash join operator can only return true for pairs of left and right values that hash to the same hash code. If two values get put in different hash buckets, the join will never compare them at all, implicitly assuming that the result of the join operator must be false. So it never makes sense to specify `HASHES` for operators that do not represent equality.

To be marked `HASHES`, the join operator must appear in a hash index operator class. Attempts to use the operator in hash joins will fail at run time if no such operator class exists. The system needs the operator class to find the data-type-specific hash function for the operator's input data type. You must also supply a suitable hash

function before you can create the operator class. Care should be exercised when preparing a hash function, because there are machine-dependent ways in which it might fail to do the right thing.

MERGES

The `MERGES` clause, if present, tells the system that it is permissible to use the merge-join method for a join based on this operator. `MERGES` only makes sense for a binary operator that returns `boolean`, and in practice the operator must represent equality for some data type or pair of data types.

Merge join is based on the idea of sorting the left- and right-hand tables into order and then scanning them in parallel. So, both data types must be capable of being fully ordered, and the join operator must be one that can only succeed for pairs of values that fall at the same place in the sort order. In practice this means that the join operator must behave like equality. It is possible to merge-join two distinct data types so long as they are logically compatible. For example, the `smallint-versus-integer` equality operator is merge-joinable. We only need sorting operators that will bring both data types into a logically compatible sequence.

Execution of a merge join requires that the system be able to identify four operators related to the merge-join equality operator: less-than comparison for the left operand data type, less-than comparison for the right operand data type, less-than comparison between the two data types, and greater-than comparison between the two data types. It is possible to specify these operators individually by name, as the `SORT1`, `SORT2`, `LTCMP`, and `GTCMP` options respectively. The system will fill in the default names if any of these are omitted when `MERGES` is specified.

left_sort_op

If this operator can support a merge join, the less-than operator that sorts the left-hand data type of this operator. `<` is the default if not specified.

right_sort_op

If this operator can support a merge join, the less-than operator that sorts the right-hand data type of this operator. `<` is the default if not specified.

less_than_op

If this operator can support a merge join, the less-than operator that compares the input data types of this operator. `<` is the default if not specified.

greater_than_op

If this operator can support a merge join, the greater-than operator that compares the input data types of this operator. `>` is the default if not specified.

To give a schema-qualified operator name in optional arguments, use the `OPERATOR()` syntax, for example:

```
COMMUTATOR = OPERATOR (myschema.===) ,
```

Notes

Any functions used to implement the operator must be defined as `IMMUTABLE`.

Examples

Here is an example of creating an operator for adding two complex numbers, assuming we have already created the definition of type `complex`. First define the function that does the work, then define the operator:

```
CREATE FUNCTION complex_add(complex, complex)
  RETURNS complex
  AS 'filename', 'complex_add'
  LANGUAGE C IMMUTABLE STRICT;
CREATE OPERATOR + (
  leftarg = complex,
  rightarg = complex,
  procedure = complex_add,
  commutator = +
);
```

To use this operator in a query:

```
SELECT (a + b) AS c FROM test_complex;
```

Compatibility

`CREATE OPERATOR` is a Greenplum Database language extension. The SQL standard does not provide for user-defined operators.

See Also

[CREATE FUNCTION](#), [CREATE TYPE](#), [ALTER OPERATOR](#), [DROP OPERATOR](#)

CREATE OPERATOR CLASS

Defines a new operator class.

Synopsis

```
CREATE OPERATOR CLASS name [DEFAULT] FOR TYPE data_type
  USING index_method AS
  {
    OPERATOR strategy_number op_name [(op_type, op_type)] [RECHECK]
    | FUNCTION support_number funcname (argument_type [, ...] )
    | STORAGE storage_type
  } [, ... ]
```

Description

CREATE OPERATOR CLASS creates a new operator class. An operator class defines how a particular data type can be used with an index. The operator class specifies that certain operators will fill particular roles or strategies for this data type and this index method. The operator class also specifies the support procedures to be used by the index method when the operator class is selected for an index column. All the operators and functions used by an operator class must be defined before the operator class is created. Any functions used to implement the operator class must be defined as IMMUTABLE.

CREATE OPERATOR CLASS does not presently check whether the operator class definition includes all the operators and functions required by the index method, nor whether the operators and functions form a self-consistent set. It is the user's responsibility to define a valid operator class.

You must be a superuser to create an operator class.

Parameters

name

The (optionally schema-qualified) name of the operator class to be defined. Two operator classes in the same schema can have the same name only if they are for different index methods.

DEFAULT

Makes the operator class the default operator class for its data type. At most one operator class can be the default for a specific data type and index method.

data_type

The column data type that this operator class is for.

index_method

The name of the index method this operator class is for. Choices are `btree`, `bitmap`, `hash`, `gist`, and `gin`.

strategy_number

The operators associated with an operator class are identified by *strategy numbers*, which serve to identify the semantics of each operator within the context of its operator class. For example, B-trees impose a strict ordering on keys, lesser to greater, and so operators like *less than* and *greater than or equal to* are interesting with respect to a B-tree. These strategies can be thought of as generalized operators. Each operator class specifies which actual operator corresponds to each strategy for a particular data type and interpretation of the index semantics. The corresponding strategy numbers for each index method are as follows:

Table A.1 B-tree and Bitmap Strategies

Operation	Strategy Number
less than	1
less than or equal	2
equal	3
greater than or equal	4
greater than	5

Table A.2 Hash Strategies

Operation	Strategy Number
equal	1

Table A.3 GiST Two-Dimensional Strategies (R-Tree)

Operation	Strategy Number
strictly left of	1
does not extend to right of	2
overlaps	3
does not extend to left of	4
strictly right of	5
same	6
contains	7
contained by	8
does not extend above	9
strictly below	10
strictly above	11
does not extend below	12

Table A.4 GIN Array Strategies

Operation	Strategy Number
overlap	1
contains	2
is contained by	3
equal	4

operator_name

The name (optionally schema-qualified) of an operator associated with the operator class.

op_type

The operand data type(s) of an operator, or *NONE* to signify a left-unity or right-unity operator. The operand data types may be omitted in the normal case where they are the same as the operator class data type.

RECHECK

If present, the index is “lossy” for this operator, and so the rows retrieved using the index must be rechecked to verify that they actually satisfy the qualification clause involving this operator.

support_number

Index methods require additional support routines in order to work. These operations are administrative routines used internally by the index methods. As with strategies, the operator class identifies which specific functions should play each of these roles for a given data type and semantic interpretation. The index method defines the set of functions it needs, and the operator class identifies the correct functions to use by assigning them to the *support function numbers* as follows:

Table A.5 B-tree and Bitmap Support Functions

Function	Support Number
Compare two keys and return an integer less than zero, zero, or greater than zero, indicating whether the first key is less than, equal to, or greater than the second.	1

Table A.6 Hash Support Functions

Function	Support Number
Compute the hash value for a key.	1

Table A.7 GiST Support Functions

Function	Support Number
consistent - determine whether key satisfies the query qualifier.	1
union - compute union of a set of keys.	2

Table A.7 GiST Support Functions

Function	Support Number
compress - compute a compressed representation of a key or value to be indexed.	3
decompress - compute a decompressed representation of a compressed key.	4
penalty - compute penalty for inserting new key into subtree with given subtree's key.	5
picksplit - determine which entries of a page are to be moved to the new page and compute the union keys for resulting pages.	6
equal - compare two keys and return true if they are equal.	7

Table A.8 GIN Support Functions

Function	Support Number
compare - compare two keys and return an integer less than zero, zero, or greater than zero, indicating whether the first key is less than, equal to, or greater than the second.	1
extractValue - extract keys from a value to be indexed.	2
extractQuery - extract keys from a query condition.	3
consistent - determine whether value matches query condition.	4

funcname

The name (optionally schema-qualified) of a function that is an index method support procedure for the operator class.

argument_types

The parameter data type(s) of the function.

storage_type

The data type actually stored in the index. Normally this is the same as the column data type, but GIN and GiST index methods allow it to be different. The `STORAGE` clause must be omitted unless the index method allows a different type to be used.

Notes

Because the index machinery does not check access permissions on functions before using them, including a function or operator in an operator class is the same as granting public execute permission on it. This is usually not an issue for the sorts of functions that are useful in an operator class.

The operators should not be defined by SQL functions. A SQL function is likely to be inlined into the calling query, which will prevent the optimizer from recognizing that the query matches an index.

Any functions used to implement the operator class must be defined as `IMMUTABLE`.

Examples

The following example command defines a GiST index operator class for the data type `_int4` (array of `int4`):

```
CREATE OPERATOR CLASS gist__int_ops
  DEFAULT FOR TYPE _int4 USING gist AS
  OPERATOR 3 &&,
  OPERATOR 6 = RECHECK,
  OPERATOR 7 @>,
  OPERATOR 8 <@,
  OPERATOR 20 @@ (_int4, query_int),
  FUNCTION 1 g_int_consistent (internal, _int4, int4),
  FUNCTION 2 g_int_union (bytea, internal),
  FUNCTION 3 g_int_compress (internal),
  FUNCTION 4 g_int_decompress (internal),
  FUNCTION 5 g_int_penalty (internal, internal, internal),
  FUNCTION 6 g_int_picksplit (internal, internal),
  FUNCTION 7 g_int_same (_int4, _int4, internal);
```

Compatibility

`CREATE OPERATOR CLASS` is a Greenplum Database extension. There is no `CREATE OPERATOR CLASS` statement in the SQL standard.

See Also

[ALTER OPERATOR CLASS](#), [DROP OPERATOR CLASS](#), [CREATE FUNCTION](#)

CREATE RESOURCE QUEUE

Defines a new resource queue.

Synopsis

```
CREATE RESOURCE QUEUE name ACTIVE THRESHOLD integer
[COST THRESHOLD float [OVERCOMMIT | NOOVERCOMMIT]
                        [IGNORE THRESHOLD float]]

CREATE RESOURCE QUEUE name COST THRESHOLD float
[OVERCOMMIT | NOOVERCOMMIT] [IGNORE THRESHOLD float]
[ACTIVE THRESHOLD integer]
```

Description

Creates a new resource queue for Greenplum Database workload management. A resource queue must have either an `ACTIVE THRESHOLD` or a `COST THRESHOLD` value (or it can have both). Only a superuser can create a resource queue.

Resource queues with an `ACTIVE THRESHOLD` limit the number of queries that can be executed by roles assigned to that queue. It controls the number of active queries that are allowed to run at the same time. The value for `ACTIVE THRESHOLD` should be an integer greater than 0.

Resource queues with a `COST THRESHOLD` limit the total cost of queries that can be executed by roles assigned to that queue. Cost is measured in the *estimated total cost* for the query as determined by the Greenplum Database query planner (as shown in the `EXPLAIN` output for a query). Therefore, an administrator must be familiar with the queries typically executed on the system in order to set an appropriate cost threshold for a queue. Cost is measured in units of disk page fetches; 1.0 equals one sequential disk page read. The value for `COST THRESHOLD` is specified as a floating point number (for example 100.0) or can also be specified as an exponent (for example 1e+2). If a resource queue is limited based on a cost threshold, then the administrator can allow `OVERCOMMIT` (the default). This means that a query that exceeds the allowed cost threshold will be allowed to run but only when the queue is idle. If `NOOVERCOMMIT` is specified, queries that exceed the cost limit will always be rejected and never allowed to run. Specifying `IGNORE THRESHOLD` allows the administrator to define a cost for ‘small queries’ that will be exempt from resource queueing.

If a value is not defined for `ACTIVE THRESHOLD` or `COST THRESHOLD`, it is set to -1 by default (meaning no limit). After defining a resource queue, you must assign roles to the queue using the `ALTER ROLE` or `CREATE ROLE` command.

Parameters

name

The name of the resource queue.

ACTIVE THRESHOLD *integer*

Resource queues with an `ACTIVE THRESHOLD` limit the number of queries that can be executed by roles assigned to that queue. It controls the number of active queries that are allowed to run at the same time. The value for `ACTIVE THRESHOLD` should be an integer greater than 0.

COST THRESHOLD *float*

Resource queues with a `COST THRESHOLD` limit the total cost of queries that can be executed by roles assigned to that queue. Cost is measured in the *estimated total cost* for the query as determined by the Greenplum Database query planner (as shown in the `EXPLAIN` output for a query). Therefore, an administrator must be familiar with the queries typically executed on the system in order to set an appropriate cost threshold for a queue. Cost is measured in units of disk page fetches; 1.0 equals one sequential disk page read. The value for `COST THRESHOLD` is specified as a floating point number (for example 100.0) or can also be specified as an exponent (for example 1e+2).

**OVERCOMMIT
NOOVERCOMMIT**

If a resource queue is limited based `COST THRESHOLD`, then the administrator can allow `OVERCOMMIT` (the default). This means that a query that exceeds the allowed cost threshold will be allowed to run but only when the queue is idle. If `NOOVERCOMMIT` is specified, queries that exceed the cost limit will always be rejected and never allowed to run.

IGNORE THRESHOLD *float*

The query cost limit of what is considered a ‘small query’. Queries with a cost under this limit will not be queued and run immediately. Cost is measured in the *estimated total cost* for the query as determined by the Greenplum Database query planner (as shown in the `EXPLAIN` output for a query). Therefore, an administrator must be familiar with the queries typically executed on the system in order to set an appropriate cost for what is considered a small query. Cost is measured in units of disk page fetches; 1.0 equals one sequential disk page read. The value for `IGNORE THRESHOLD` is specified as a floating point number (for example 100.0) or can also be specified as an exponent (for example 1e+2).

Notes

Use the `pg_resqueue_status` system view to see the limit settings and current status of a resource queue:

```
SELECT * from pg_resqueue_status WHERE rsqname='queue_name';
```

There is also another system view named `pg_stat_resqueues` which shows statistical metrics for a resource queue over time. To use this view, however, you must enable the `stats_queue_level` server configuration parameter. See [“Managing Workload and Resources”](#) on page 85 for more information about using resource queues.

Examples

Create a resource queue with an active query limit of 20:

```
CREATE RESOURCE QUEUE myqueue ACTIVE THRESHOLD 20;
```

Create a resource queue with a query cost limit of 3000.0:

```
CREATE RESOURCE QUEUE myqueue COST THRESHOLD 3000.0;
```

Create a resource queue with a query cost limit of 3^{10} (or 30000000000.0) and do not allow overcommit. Allow small queries with a cost under 500 to run immediately:

```
CREATE RESOURCE QUEUE myqueue COST THRESHOLD 3e+10  
NOOVERCOMMIT IGNORE THRESHOLD 500.0;
```

Create a resource queue with both an active query limit and a query cost limit:

```
CREATE RESOURCE QUEUE myqueue ACTIVE THRESHOLD 30 COST  
THRESHOLD 5000.00;
```

Compatibility

CREATE RESOURCE QUEUE is a Greenplum Database extension. There is no provision for resource queues or workload management in the SQL standard.

See Also

[ALTER ROLE](#), [CREATE ROLE](#), [ALTER RESOURCE QUEUE](#), [DROP RESOURCE QUEUE](#)

CREATE ROLE

Defines a new database role (user or group).

Synopsis

```
CREATE ROLE name [[WITH] option [ ... ]]
```

where *option* can be:

```

SUPERUSER | NOSUPERUSER
| CREATEDB | NOCREATEDB
| CREATEROLE | NOCREATEROLE
| INHERIT | NOINHERIT
| LOGIN | NOLOGIN
| CONNECTION LIMIT connlimit
| [ ENCRYPTED | UNENCRYPTED ] PASSWORD 'password'
| VALID UNTIL 'timestamp'
| IN ROLE rolename [, ...]
| ROLE rolename [, ...]
| ADMIN rolename [, ...]
| RESOURCE QUEUE queue_name

```

Description

CREATE ROLE adds a new role to a Greenplum Database system. A role is an entity that can own database objects and have database privileges. A role can be considered a user, a group, or both depending on how it is used. You must have CREATEROLE privilege or be a database superuser to use this command.

Note that roles are defined at the system-level and are valid for all databases in your Greenplum Database system.

Parameters

name

The name of the new role.

SUPERUSER **NOSUPERUSER**

If SUPERUSER is specified, the role being defined will be a superuser, who can override all access restrictions within the database. Superuser status is dangerous and should be used only when really needed. You must yourself be a superuser to create a new superuser. NOSUPERUSER is the default.

CREATEDB **NOCREATEDB**

If CREATEDB is specified, the role being defined will be allowed to create new databases. NOCREATEDB (the default) will deny a role the ability to create databases.

**CREATEROLE
NOCREATEROLE**

If `CREATEDB` is specified, the role being defined will be allowed to create new roles, alter other roles, and drop other roles. `NOCREATEROLE` (the default) will deny a role the ability to create roles or modify roles other than their own.

**INHERIT
NOINHERIT**

If specified, `INHERIT` (the default) allows the role to use whatever database privileges have been granted to all roles it is directly or indirectly a member of. With `NOINHERIT`, membership in another role only grants the ability to `SET ROLE` to that other role.

**LOGIN
NOLOGIN**

If specified, `LOGIN` allows a role to log in to a database. A role having the `LOGIN` attribute can be thought of as a user. Roles with `NOLOGIN` (the default) are useful for managing database privileges, and can be thought of as groups.

CONNECTION LIMIT *connlimit*

Warning: Setting a connection limit at the role level cannot be enforced in Greenplum Database and may cause queries to fail. Leave this set at the default of `-1` (no limit). Connection limits may only be set at the system level in Greenplum Database. See “[Limiting Concurrent Connections](#)” on page 75 for more information.

PASSWORD *password*

Sets the user password for roles with the `LOGIN` attribute. If you do not plan to use password authentication you can omit this option. If no password is specified, the password will be set to null and password authentication will always fail for that user. A null password can optionally be written explicitly as `PASSWORD NULL`.

**ENCRYPTED
UNENCRYPTED**

These key words control whether the password is stored encrypted in the system catalogs. (If neither is specified, the default behavior is determined by the configuration parameter `password_encryption`.) If the presented password string is already in MD5-encrypted format, then it is stored encrypted as-is, regardless of whether `ENCRYPTED` or `UNENCRYPTED` is specified (since the system cannot decrypt the specified encrypted password string). This allows reloading of encrypted passwords during dump/restore.

Note that older clients may lack support for the MD5 authentication mechanism that is needed to work with passwords that are stored encrypted.

VALID UNTIL '*timestamp*'

The `VALID UNTIL` clause sets a date and time after which the role’s password is no longer valid. If this clause is omitted the password will never expire.

IN ROLE *rolename*

Adds the new role as a member of the named roles. Note that there is no option to add the new role as an administrator; use a separate `GRANT` command to do that.

ROLE *rolename*

Adds the named roles as members of this role, making this new role a group.

ADMIN *rolename*

The `ADMIN` clause is like `ROLE`, but the named roles are added to the new role `WITH ADMIN OPTION`, giving them the right to grant membership in this role to others.

RESOURCE QUEUE *queue_name*

The name of the resource queue to which the new user-level role is to be assigned. Only roles with `LOGIN` privilege can be assigned to a resource queue. The special keyword `NONE` means that no resource queue is assigned. A role can only belong to one resource queue.

Notes

The preferred way to add and remove role members (manage groups) is to use `GRANT` and `REVOKE`.

The `VALID UNTIL` clause defines an expiration time for a password only, not for the role. The expiration time is not enforced when logging in using a non-password-based authentication method.

The `INHERIT` attribute governs inheritance of grantable privileges (access privileges for database objects and role memberships). It does not apply to the special role attributes set by `CREATE ROLE` and `ALTER ROLE`. For example, being a member of a role with `CREATEDB` privilege does not immediately grant the ability to create databases, even if `INHERIT` is set.

The `INHERIT` attribute is the default for reasons of backwards compatibility. In prior releases of Greenplum Database, users always had access to all privileges of groups they were members of. However, `NOINHERIT` provides a closer match to the semantics specified in the SQL standard.

Be careful with the `CREATEROLE` privilege. There is no concept of inheritance for the privileges of a `CREATEROLE`-role. That means that even if a role does not have a certain privilege but is allowed to create other roles, it can easily create another role with different privileges than its own (except for creating roles with superuser privileges). For example, if a role has the `CREATEROLE` privilege but not the `CREATEDB` privilege, it can create a new role with the `CREATEDB` privilege. Therefore, regard roles that have the `CREATEROLE` privilege as almost-superuser-roles.

The `CONNECTION LIMIT` option is never enforced for superusers.

Caution must be exercised when specifying an unencrypted password with this command. The password will be transmitted to the server in clear-text, and it might also be logged in the client's command history or the server log. The client program `createuser`, however, transmits the password encrypted. Also, `psql` contains a command `\password` that can be used to safely change the password later.

Examples

Create a role that can log in, but don't give it a password:

```
CREATE ROLE jonathan LOGIN;
```

Create a role that belongs to a resource queue:

```
CREATE ROLE jonathan LOGIN RESOURCE QUEUE poweruser;
```

Create a role with a password that is valid until the end of 2009 (`CREATE USER` is the same as `CREATE ROLE` except that it implies `LOGIN`):

```
CREATE USER joelle WITH PASSWORD 'jw8s0F4' VALID UNTIL
'2010-01-01';
```

Create a role that can create databases and manage other roles:

```
CREATE ROLE admin WITH CREATEDB CREATEROLE;
```

Compatibility

The SQL standard defines the concepts of users and roles, but it regards them as distinct concepts and leaves all commands defining users to be specified by the database implementation. In Greenplum Database users and roles are unified into a single type of object. Roles therefore have many more optional attributes than they do in the standard.

`CREATE ROLE` is in the SQL standard, but the standard only requires the syntax:

```
CREATE ROLE name [WITH ADMIN rolename]
```

Allowing multiple initial administrators, and all the other options of `CREATE ROLE`, are Greenplum Database extensions.

The behavior specified by the SQL standard is most closely approximated by giving users the `NOINHERIT` attribute, while roles are given the `INHERIT` attribute.

See Also

[SET ROLE](#), [ALTER ROLE](#), [DROP ROLE](#), [GRANT](#), [REVOKE](#), [CREATE RESOURCE QUEUE](#)

CREATE RULE

Defines a new rewrite rule.

Synopsis

```
CREATE [OR REPLACE] RULE name AS ON event
  TO table [WHERE condition]
  DO [ALSO | INSTEAD] { NOTHING | command | (command; command
  ...) }
```

Description

`CREATE RULE` defines a new rule applying to a specified table or view. `CREATE OR REPLACE RULE` will either create a new rule, or replace an existing rule of the same name for the same table.

The Greenplum Database rule system allows one to define an alternate action to be performed on insertions, updates, or deletions in database tables. A rule causes additional or alternate commands to be executed when a given command on a given table is executed. Rules can be used on views as well. It is important to realize that a rule is really a command transformation mechanism, or command macro. The transformation happens before the execution of the commands starts. It does not operate independently for each physical row as does a trigger.

`ON SELECT` rules must be unconditional `INSTEAD` rules and must have actions that consist of a single `SELECT` command. Thus, an `ON SELECT` rule effectively turns the table into a view, whose visible contents are the rows returned by the rule's `SELECT` command rather than whatever had been stored in the table (if anything). It is considered better style to write a `CREATE VIEW` command than to create a real table and define an `ON SELECT` rule for it.

You can create the illusion of an updatable view by defining `ON INSERT`, `ON UPDATE`, and `ON DELETE` rules to replace update actions on the view with appropriate updates on other tables. If you want to support `INSERT RETURNING` and so on, then be sure to put a suitable `RETURNING` clause into each of these rules.

Rules are also helpful for managing partitioned tables. You can define `ON INSERT` rules on the parent table to route inserted rows to the correct partitioned child table. Note that rules do not work for `COPY` commands.

There is a catch if you try to use conditional rules for view updates: there must be an unconditional `INSTEAD` rule for each action you wish to allow on the view. If the rule is conditional, or is not `INSTEAD`, then the system will still reject attempts to perform the update action, because it thinks it might end up trying to perform the action on the dummy table of the view in some cases. If you want to handle all the useful cases in conditional rules, add an unconditional `DO INSTEAD NOTHING` rule to ensure that the system understands it will never be called on to update the dummy table. Then make the conditional rules non-`INSTEAD`; in the cases where they are applied, they add to the default `INSTEAD NOTHING` action. (This method does not currently work to support `RETURNING` queries, however.)

Parameters

name

The name of a rule to create. This must be distinct from the name of any other rule for the same table. Multiple rules on the same table and same event type are applied in alphabetical name order.

event

The event is one of `SELECT`, `INSERT`, `UPDATE`, or `DELETE`.

table

The name (optionally schema-qualified) of the table or view the rule applies to.

condition

Any SQL conditional expression (returning boolean). The condition expression may not refer to any tables except `NEW` and `OLD`, and may not contain aggregate functions. `NEW` and `OLD` refer to values in the referenced table. `NEW` is valid in `ON INSERT` and `ON UPDATE` rules to refer to the new row being inserted or updated. `OLD` is valid in `ON UPDATE` and `ON DELETE` rules to refer to the existing row being updated or deleted.

INSTEAD

`INSTEAD` indicates that the commands should be executed instead of the original command.

ALSO

`ALSO` indicates that the commands should be executed in addition to the original command. If neither `ALSO` nor `INSTEAD` is specified, `ALSO` is the default.

command

The command or commands that make up the rule action. Valid commands are `SELECT`, `INSERT`, `UPDATE`, or `DELETE`. The special table names `NEW` and `OLD` may be used to refer to values in the referenced table. `NEW` is valid in `ON INSERT` and `ON UPDATE` rules to refer to the new row being inserted or updated. `OLD` is valid in `ON UPDATE` and `ON DELETE` rules to refer to the existing row being updated or deleted.

Notes

You must be the owner of a table to create or change rules for it.

In a rule for `INSERT`, `UPDATE`, or `DELETE` on a view, you can add a `RETURNING` clause that emits the view's columns. This clause will be used to compute the outputs if the rule is triggered by an `INSERT RETURNING`, `UPDATE RETURNING`, or `DELETE RETURNING` command respectively. When the rule is triggered by a command without `RETURNING`, the rule's `RETURNING` clause will be ignored. The current implementation allows only unconditional `INSTEAD` rules to contain `RETURNING`; furthermore there can be at most one `RETURNING` clause among all the rules for the same event. (This

ensures that there is only one candidate `RETURNING` clause to be used to compute the results.) `RETURNING` queries on the view will be rejected if there is no `RETURNING` clause in any available rule.

It is very important to take care to avoid circular rules. Recursive rules are not validated at rule create time, but will report an error at execution time.

Examples

Create a rule that inserts rows into the child table `b2001` when a user tries to insert into the partitioned parent table `rank`:

```
CREATE RULE b2001 AS ON INSERT TO rank WHERE gender='M' and
year='2001' DO INSTEAD INSERT INTO b2001 VALUES (NEW.id,
NEW.rank, NEW.year, NEW.gender, NEW.count);
```

Compatibility

`CREATE RULE` is a Greenplum Database language extension, as is the entire query rewrite system.

See Also

[DROP RULE](#), [CREATE TABLE](#), [CREATE VIEW](#)

CREATE SCHEMA

Defines a new schema.

Synopsis

```
CREATE SCHEMA schema_name [AUTHORIZATION username]
[schema_element [ ... ]]

CREATE SCHEMA AUTHORIZATION rolename [schema_element [ ... ]]
```

Description

CREATE SCHEMA enters a new schema into the current database. The schema name must be distinct from the name of any existing schema in the current database.

A schema is essentially a namespace: it contains named objects (tables, data types, functions, and operators) whose names may duplicate those of other objects existing in other schemas. Named objects are accessed either by qualifying their names with the schema name as a prefix, or by setting a search path that includes the desired schema(s). A CREATE command specifying an unqualified object name creates the object in the current schema (the one at the front of the search path, which can be determined with the function `current_schema`).

Optionally, CREATE SCHEMA can include subcommands to create objects within the new schema. The subcommands are treated essentially the same as separate commands issued after creating the schema, except that if the AUTHORIZATION clause is used, all the created objects will be owned by that role.

Parameters

schema_name

The name of a schema to be created. If this is omitted, the user name is used as the schema name. The name cannot begin with `pg_`, as such names are reserved for system catalog schemas.

rolename

The name of the role who will own the schema. If omitted, defaults to the role executing the command. Only superusers may create schemas owned by roles other than themselves.

schema_element

An SQL statement defining an object to be created within the schema. Currently, only CREATE TABLE, CREATE VIEW, CREATE INDEX, CREATE SEQUENCE, CREATE TRIGGER and GRANT are accepted as clauses within CREATE SCHEMA. Other kinds of objects may be created in separate commands after the schema is created.

Notes

To create a schema, the invoking user must have the `CREATE` privilege for the current database or be a superuser.

Examples

Create a schema:

```
CREATE SCHEMA myschema;
```

Create a schema for role *joe* (the schema will also be named *joe*):

```
CREATE SCHEMA AUTHORIZATION joe;
```

Compatibility

The SQL standard allows a `DEFAULT CHARACTER SET` clause in `CREATE SCHEMA`, as well as more subcommand types than are presently accepted by Greenplum Database.

The SQL standard specifies that the subcommands in `CREATE SCHEMA` may appear in any order. The present Greenplum Database implementation does not handle all cases of forward references in subcommands; it may sometimes be necessary to reorder the subcommands in order to avoid forward references.

According to the SQL standard, the owner of a schema always owns all objects within it. Greenplum Database allows schemas to contain objects owned by users other than the schema owner. This can happen only if the schema owner grants the `CREATE` privilege on the schema to someone else.

See Also

[ALTER SCHEMA](#), [DROP SCHEMA](#)

CREATE SEQUENCE

Defines a new sequence generator.

Synopsis

```
CREATE [TEMPORARY | TEMP] SEQUENCE name
      [INCREMENT [BY] value]
      [MINVALUE minvalue | NO MINVALUE]
      [MAXVALUE maxvalue | NO MAXVALUE]
      [START [ WITH ] start]
      [CACHE cache]
      [[NO] CYCLE]
      [OWNED BY { table.column | NONE }]
```

Description

CREATE SEQUENCE creates a new sequence number generator. This involves creating and initializing a new special single-row table. The generator will be owned by the user issuing the command.

If a schema name is given, then the sequence is created in the specified schema. Otherwise it is created in the current schema. Temporary sequences exist in a special schema, so a schema name may not be given when creating a temporary sequence. The sequence name must be distinct from the name of any other sequence, table, index, or view in the same schema.

After a sequence is created, you use the `nextval` function to operate on the sequence. For example, to insert a row into a table that gets the next value of a sequence:

```
INSERT INTO distributors VALUES (nextval('myserial'),
                                'acme');
```

You can also use the function `setval` to operate on a sequence, but only for queries that do not operate on distributed data. For example, the following query is allowed because it resets the sequence counter value for the sequence generator process on the master:

```
SELECT setval('myserial', 201);
```

But the following query will be rejected in Greenplum Database because it operates on distributed data:

```
INSERT INTO product VALUES (setval('myserial', 201),
                              'gizmo');
```

In a regular (non-distributed) database, functions that operate on the sequence go to the local sequence table to get values as they are needed. In Greenplum Database, however, keep in mind that each segment is its own distinct database process. Therefore the segments need a single point of truth to go for sequence values so that all segments get incremented correctly and the sequence moves forward in the right order. A sequence server process runs on the master and is the point-of-truth for a sequence in a Greenplum distributed database. Segments get sequence values at runtime from the master.

Because of this distributed sequence design, there are some limitations on the functions that operate on a sequence in Greenplum Database:

- `lastval` and `currval` functions are not supported.
- `setval` can only be used to set the value of the sequence generator on the master, it cannot be used in subqueries to update records on distributed table data.
- `nextval` sometimes grabs a block of values from the master for a segment to use, depending on the query. So values may sometimes be skipped in the sequence if all of the block turns out not to be needed at the segment level. Note that a regular PostgreSQL database does this too, so this is not something unique to Greenplum Database.

Although you cannot update a sequence directly, you can use a query like:

```
SELECT * FROM sequence_name;
```

to examine the parameters and current state of a sequence. In particular, the `last_value` field of the sequence shows the last value allocated by any session.

Parameters

TEMPORARY | TEMP

If specified, the sequence object is created only for this session, and is automatically dropped on session exit. Existing permanent sequences with the same name are not visible (in this session) while the temporary sequence exists, unless they are referenced with schema-qualified names.

name

The name (optionally schema-qualified) of the sequence to be created.

increment

Specifies which value is added to the current sequence value to create a new value. A positive value will make an ascending sequence, a negative one a descending sequence. The default value is 1.

minvalue

NO MINVALUE

Determines the minimum value a sequence can generate. If this clause is not supplied or `NO MINVALUE` is specified, then defaults will be used. The defaults are 1 and -263-1 for ascending and descending sequences, respectively.

maxvalue

NO MAXVALUE

Determines the maximum value for the sequence. If this clause is not supplied or `NO MAXVALUE` is specified, then default values will be used. The defaults are 263-1 and -1 for ascending and descending sequences, respectively.

start

Allows the sequence to begin anywhere. The default starting value is *minvalue* for ascending sequences and *maxvalue* for descending ones.

cache

Specifies how many sequence numbers are to be preallocated and stored in memory for faster access. The minimum (and default) value is 1 (no cache).

CYCLE**NO CYCLE**

Allows the sequence to wrap around when the *maxvalue* (for ascending) or *minvalue* (for descending) has been reached. If the limit is reached, the next number generated will be the *minvalue* (for ascending) or *maxvalue* (for descending). If **NO CYCLE** is specified, any calls to `nextval` after the sequence has reached its maximum value will return an error. If not specified, **NO CYCLE** is the default.

OWNED BY *table.column***OWNED BY NONE**

Causes the sequence to be associated with a specific table column, such that if that column (or its whole table) is dropped, the sequence will be automatically dropped as well. The specified table must have the same owner and be in the same schema as the sequence. **OWNED BY NONE**, the default, specifies that there is no such association.

Notes

Sequences are based on bigint arithmetic, so the range cannot exceed the range of an eight-byte integer (-9223372036854775808 to 9223372036854775807).

Although multiple sessions are guaranteed to allocate distinct sequence values, the values may be generated out of sequence when all the sessions are considered. For example, session A might reserve values 1..10 and return `nextval=1`, then session B might reserve values 11..20 and return `nextval=11` before session A has generated `nextval=2`. Thus, you should only assume that the `nextval` values are all distinct, not that they are generated purely sequentially. Also, *last_value* will reflect the latest value reserved by any session, whether or not it has yet been returned by `nextval`.

Examples

Create a sequence named *myseq*:

```
CREATE SEQUENCE myseq START 101;
```

Insert a row into a table that gets the next value:

```
INSERT INTO distributors VALUES (nextval('myseq'), 'acme');
```

Reset the sequence counter value on the master:

```
SELECT setval('myseq', 201);
```

Illegal use of `setval` in Greenplum Database (setting sequence values on distributed data):

```
INSERT INTO product VALUES (setval('myseq', 201), 'gizmo');
```

Compatibility

`CREATE SEQUENCE` conforms to the SQL standard, with the following exceptions:

- The `AS data_type` expression specified in the SQL standard is not supported.
- Obtaining the next value is done using the `nextval()` function instead of the `NEXT VALUE FOR` expression specified in the SQL standard.
- The `OWNED BY` clause is a Greenplum Database extension.

See Also

`ALTER SEQUENCE`, `DROP SEQUENCE`

CREATE TABLE

Defines a new table.

Synopsis

```
CREATE [[GLOBAL | LOCAL] {TEMPORARY | TEMP}] TABLE table_name (
    [ { column_name data_type [DEFAULT default_expr]
      [column_constraint [ ... ] ]
      | table_constraint
      | LIKE other_table [{INCLUDING | EXCLUDING}
                          {DEFAULTS | CONSTRAINTS}] ...}
      [, ... ] ] )
[ INHERITS ( parent_table [, ... ] ) ]
[ WITH ( storage_parameter=value [, ... ] ) ]
[ ON COMMIT {PRESERVE ROWS | DELETE ROWS | DROP} ]
[ TABLESPACE tablespace ]
[ DISTRIBUTED BY (column, [ ... ] ) | DISTRIBUTED RANDOMLY ]
[ PARTITION BY partition_type (column)
  [ SUBPARTITION BY partition_type (column) ]
  [ SUBPARTITION TEMPLATE ( template_spec ) ]
  [...]
  ( partition_spec )
  | [ SUBPARTITION BY partition_type (column) ]
  [...]
  ( partition_spec
    [ ( subpartition_spec
        [(...)]
      ) ]
  ) ]
)
```

where *storage_parameter* is:

```
APPENDONLY={TRUE|FALSE}
ORIENTATION={COLUMN|ROW}
COMPRESSTYPE={ZLIB|QUICKLZ}
COMPRESSLEVEL={0-9 | 1}
FILLFACTOR={10-100}
OIDS [=TRUE|FALSE]
```

where *column_constraint* is:

```
[CONSTRAINT constraint_name]
NOT NULL | NULL
| UNIQUE [USING INDEX TABLESPACE tablespace]
          [WITH ( FILLFACTOR = value )]
| PRIMARY KEY [USING INDEX TABLESPACE tablespace]
              [WITH ( FILLFACTOR = value )]
| CHECK ( expression )
```

and *table_constraint* is:

```
[CONSTRAINT constraint_name]
```

```

UNIQUE ( column_name [, ... ] )
    [USING INDEX TABLESPACE tablespace]
    [WITH ( FILLFACTOR=value )]
| PRIMARY KEY ( column_name [, ... ] )
    [USING INDEX TABLESPACE tablespace]
    [WITH ( FILLFACTOR=value )]
| CHECK ( expression )

```

where *partition_type* is:

```

LIST
| RANGE

```

where *partition_specification* is:

partition_element [, ...]

and *partition_element* is:

```

DEFAULT PARTITION name
| [PARTITION name] VALUES (list_value [,...])
| [PARTITION name]
    START ([datatype] 'start_value') [INCLUSIVE | EXCLUSIVE]
    [ END ([datatype] 'end_value') [INCLUSIVE | EXCLUSIVE] ]
    [ EVERY ([datatype] [number | INTERVAL] 'interval_value') ]
| [PARTITION name]
    END ([datatype] 'end_value') [INCLUSIVE | EXCLUSIVE]
    [ EVERY ([datatype] [number | INTERVAL] 'interval_value') ]
[ WITH ( partition_storage_parameter=value [, ... ] ) ]
[ TABLESPACE tablespace ]

```

where *subpartition_spec* or *template_spec* is:

subpartition_element [, ...]

and *subpartition_element* is:

```

DEFAULT SUBPARTITION name
| [SUBPARTITION name] VALUES (list_value [,...])
| [SUBPARTITION name]
    START ([datatype] 'start_value') [INCLUSIVE | EXCLUSIVE]
    [ END ([datatype] 'end_value') [INCLUSIVE | EXCLUSIVE] ]
    [ EVERY ([datatype] [number | INTERVAL] 'interval_value') ]
| [SUBPARTITION name]
    END ([datatype] 'end_value') [INCLUSIVE | EXCLUSIVE]
    [ EVERY ([datatype] [number | INTERVAL] 'interval_value') ]
[ WITH ( partition_storage_parameter=value [, ... ] ) ]
[ TABLESPACE tablespace ]

```

Description

`CREATE TABLE` will create a new, initially empty table in the current database. The table will be owned by the user issuing the command.

If a schema name is given then the table is created in the specified schema. Otherwise it is created in the current schema. Temporary tables exist in a special schema, so a schema name may not be given when creating a temporary table. The name of the table must be distinct from the name of any other table, external table, sequence, index, or view in the same schema.

The optional constraint clauses specify conditions that new or updated rows must satisfy for an insert or update operation to succeed. A constraint is an SQL object that helps define the set of valid values in the table in various ways.

There are two ways to define constraints: table constraints and column constraints. A column constraint is defined as part of a column definition. A table constraint definition is not tied to a particular column, and it can encompass more than one column. Every column constraint can also be written as a table constraint; a column constraint is only a notational convenience for use when the constraint only affects one column.

When creating a table, there is an additional clause to declare the Greenplum Database distribution policy. If a `DISTRIBUTED BY` or `DISTRIBUTED RANDOMLY` clause is not supplied, then Greenplum assigns a hash distribution policy to the table using either the `PRIMARY KEY` (if the table has one) or the first column of the table as the distribution key. Columns of geometric or user-defined data types are not eligible as Greenplum distribution key columns. If a table does not have a column of an eligible data type, the rows are distributed based on a round-robin or random distribution. To ensure an even distribution of data in your Greenplum Database system, you want to choose a distribution key that is unique for each record, or if that is not possible, then choose `DISTRIBUTED RANDOMLY`.

The `PARTITION BY` clause allows you to divide the table into multiple sub-tables (or child tables) that inherit from the parent table. Table partitioning creates a persistent relationship between the child table partitions and the parent table, so that all of the schema information from the parent table propagates to the child table partitions. Each child table partition is created with a distinct `CHECK` constraint, which limits the data the table can contain based on some defining criteria. The `CHECK` constraints are also used by the query planner to determine which table partitions to scan in order to satisfy a given query predicate.

Parameters

GLOBAL | **LOCAL**

These keywords are present for SQL standard compatibility, but have no effect in Greenplum Database.

TEMPORARY | **TEMP**

If specified, the table is created as a temporary table. Temporary tables are automatically dropped at the end of a session, or optionally at the end of the current transaction (see `ON COMMIT`). Existing permanent tables with the same name are not visible to the current session while the temporary table exists, unless they are referenced with schema-qualified names. Any indexes created on a temporary table are automatically temporary as well.

table_name

The name (optionally schema-qualified) of the table to be created.

column_name

The name of a column to be created in the new table.

data_type

The data type of the column. This may include array specifiers.

DEFAULT *default_expr*

The **DEFAULT** clause assigns a default data value for the column whose column definition it appears within. The value is any variable-free expression (subqueries and cross-references to other columns in the current table are not allowed). The data type of the default expression must match the data type of the column. The default expression will be used in any insert operation that does not specify a value for the column. If there is no default for a column, then the default is null.

INHERITS

The optional **INHERITS** clause specifies a list of tables from which the new table automatically inherits all columns. Use of **INHERITS** creates a persistent relationship between the new child table and its parent table(s). Schema modifications to the parent(s) normally propagate to children as well, and by default the data of the child table is included in scans of the parent(s).

In Greenplum Database, the **INHERITS** clause is not used when creating partitioned tables. Although the concept of inheritance is used in partition hierarchies, the inheritance structure of a partitioned table is created using the **PARTITION BY** clause.

If the same column name exists in more than one parent table, an error is reported unless the data types of the columns match in each of the parent tables. If there is no conflict, then the duplicate columns are merged to form a single column in the new table. If the column name list of the new table contains a column name that is also inherited, the data type must likewise match the inherited column(s), and the column definitions are merged into one. However, inherited and new column declarations of the same name need not specify identical constraints: all constraints provided from any declaration are merged together and all are applied to the new table. If the new table explicitly specifies a default value for the column, this default overrides any defaults from inherited declarations of the column. Otherwise, any parents that specify default values for the column must all specify the same default, or an error will be reported.

LIKE *other_table* [{INCLUDING | EXCLUDING} {DEFAULTS | CONSTRAINTS}]

The **LIKE** clause specifies a table from which the new table automatically copies all column names, their data types, and their not-null constraints. Unlike **INHERITS**, the new table and original table are completely decoupled after creation is complete.

Default expressions for the copied column definitions will only be copied if **INCLUDING DEFAULTS** is specified. The default behavior is to exclude default expressions, resulting in the copied columns in the new table having null defaults.

Not-null constraints are always copied to the new table. `CHECK` constraints will only be copied if `INCLUDING CONSTRAINTS` is specified; other types of constraints will *never* be copied. Also, no distinction is made between column constraints and table constraints — when constraints are requested, all check constraints are copied.

Note also that unlike `INHERITS`, copied columns and constraints are not merged with similarly named columns and constraints. If the same name is specified explicitly or in another `LIKE` clause an error is signalled.

CONSTRAINT *constraint_name*

An optional name for a column or table constraint. If the constraint is violated, the constraint name is present in error messages, so constraint names like *column must be positive* can be used to communicate helpful constraint information to client applications. (Double-quotes are needed to specify constraint names that contain spaces.) If a constraint name is not specified, the system generates a name.

NULL | NOT NULL

Specifies if the column is or is not allowed to contain null values. `NULL` is the default.

UNIQUE (*column constraint*)

UNIQUE (*column_name* [, ...]) (*table constraint*)

The `UNIQUE` constraint specifies that a group of one or more columns of a table may contain only unique values. The behavior of the unique table constraint is the same as that for column constraints, with the additional capability to span multiple columns. For the purpose of a unique constraint, null values are not considered equal. The column(s) that are unique must also contain all of the Greenplum distribution key column(s).

PRIMARY KEY (*column constraint*)

PRIMARY KEY (*column_name* [, ...]) (*table constraint*)

The primary key constraint specifies that a column or columns of a table may contain only unique (non-duplicate), non-null values. Technically, `PRIMARY KEY` is merely a combination of `UNIQUE` and `NOT NULL`, but identifying a set of columns as primary key also provides metadata about the design of the schema, as a primary key implies that other tables may rely on this set of columns as a unique identifier for rows. For a table to have a primary key, it must be hash distributed (not randomly distributed), and the primary key must contain all (or a superset) of the Greenplum distribution key column(s).

CHECK (*expression*)

The `CHECK` clause specifies an expression producing a Boolean result which new or updated rows must satisfy for an insert or update operation to succeed. Expressions evaluating to `TRUE` or `UNKNOWN` succeed. Should any row of an insert or update operation produce a `FALSE` result an error exception is raised and the insert or update does not alter the database. A check constraint specified as a column constraint should reference that column's value only, while an expression appearing in a table constraint may reference multiple columns. `CHECK` expressions cannot contain subqueries nor refer to variables other than columns of the current row.

WITH (*storage_option=value*)

The **WITH** clause can be used to set storage options for the table or its indexes. Note that you can also set storage parameters on a particular partition or subpartition by declaring the **WITH** clause in the partition specification. The following storage options are available:

APPENDONLY - Set to **TRUE** to create the table as an append-only table. If **FALSE** or not declared, the table will be created as a regular heap-storage table.

ORIENTATION - Set to **column** for column-oriented storage, or **row** (the default) for row-oriented storage. This option is only valid if **APPENDONLY=TRUE**. Heap-storage tables can only be row-oriented.

COMPRESSTYPE - Set to **ZLIB** (the default) or **QUICKLZ** to specify the type of compression used. QuickLZ uses less CPU power and compresses data faster at a lower compression ratio than zlib. Conversely, zlib provides more compact compression ratios at lower speeds. This option is only valid if **APPENDONLY=TRUE**.

COMPRESSLEVEL - For zlib compression of append-only tables, set to a value between 1 (fastest compression) to 9 (highest compression ratio). QuickLZ compression level can only be set to 1. If not declared, the default is 0 (no compression). This option is only valid if **APPENDONLY=TRUE**.

FILLFACTOR - See **CREATE INDEX** for more information about this index storage parameter.

OIDS - Set to **OIDS=FALSE** (the default) so that rows do not have object identifiers assigned to them. Greenplum strongly recommends that you do not enable OIDS when creating a table. On large tables, such as those in a typical Greenplum Database system, using OIDs for table rows can cause wrap-around of the 32-bit OID counter. Once the counter wraps around, OIDs can no longer be assumed to be unique, which not only makes them useless to user applications, but can also cause problems in the Greenplum Database system catalog tables. In addition, excluding OIDs from a table reduces the space required to store the table on disk by 4 bytes per row, slightly improving performance. OIDs are not allowed on column-oriented tables.

ON COMMIT

The behavior of temporary tables at the end of a transaction block can be controlled using **ON COMMIT**. The three options are:

PRESERVE ROWS

No special action is taken at the ends of transactions for temporary tables. This is the default behavior.

DELETE ROWS

All rows in the temporary table will be deleted at the end of each transaction block. Essentially, an automatic **TRUNCATE** is done at each commit.

DROP

The temporary table will be dropped at the end of the current transaction block.

TABLESPACE *tablespace*

Currently not supported in Greenplum Database. The tablespace is the name of the tablespace in which the new table is to be created. If not specified, the database's default tablespace is used.

USING INDEX TABLESPACE *tablespace*

This clause allows selection of the tablespace in which the index associated with a `UNIQUE` or `PRIMARY KEY` constraint will be created. If not specified, the database's default tablespace is used.

**DISTRIBUTED BY (*column*, [...])
DISTRIBUTED RANDOMLY**

Used to declare the Greenplum Database distribution policy for the table. `DISTRIBUTED BY` uses hash distribution with one or more columns declared as the distribution key. For the most even data distribution, the distribution key should be the primary key of the table or a unique column (or set of columns). If that is not possible, then you may choose `DISTRIBUTED RANDOMLY`, which will send the data round-robin to the segment instances. If not supplied, then hash distribution is chosen using the `PRIMARY KEY` (if the table has one) or the first eligible column of the table as the distribution key.

PARTITION BY

Declares one or more columns by which to partition the table.

partition_type

Declares partition type: `LIST` (list of values) or `RANGE` (a numeric or date range).

partition_specification

Declares the individual partitions to create. Each partition can be defined individually or, for range partitions, you can use the `EVERY` clause (with a `START` and optional `END` clause) to define an increment pattern to use to create the individual partitions.

DEFAULT PARTITION *name* - Declares a default partition. When data does not match to an existing partition, it is inserted into the default partition. Partition designs that do not have a default partition will reject incoming rows that do not match to an existing partition.

PARTITION *name* - Declares a name to use for the partition. Partitions are created using the following naming convention:

parentname_level#_prt_givename.

VALUES - For list partitions, defines the value(s) that the partition will contain.

START - For range partitions, defines the starting range value for the partition. By default, start values are `INCLUSIVE`. For example, if you declared a start date of `'2008-01-01'`, then the partition would contain all dates greater than or equal to `'2008-01-01'`. Typically the data type of the `START` expression is the same type as the partition key column. If that is not the case, then you must explicitly cast to the intended data type.

END - For range partitions, defines the ending range value for the partition. By default, end values are `EXCLUSIVE`. For example, if you declared an end date of `'2008-02-01'`, then the partition would contain all dates less than but not equal to `'2008-02-01'`. Typically the data type of the `END` expression is the same type as the partition key column. If that is not the case, then you must explicitly cast to the intended data type.

EVERY - For range partitions, defines how to increment the values from `START` to `END` to create individual partitions. Typically the data type of the `EVERY` expression is the same type as the partition key column. If that is not the case, then you must explicitly cast to the intended data type.

WITH - Sets the table storage options for a partition. For example, you may want older partitions to be append-only tables and newer partitions to be regular heap tables.

TABLESPACE - Currently not supported in Greenplum Database. The name of the tablespace in which the partition is to be created.

SUBPARTITION BY

Declares one or more columns by which to subpartition the first-level partitions of the table. The format of the subpartition specification is similar to that of a partition specification described above.

SUBPARTITION TEMPLATE

Instead of declaring each subpartition definition individually for each partition, you can optionally declare a subpartition template to be used to create the subpartitions. This subpartition specification would then apply to all parent partitions.

Notes

Using OIDs in new applications is not recommended: where possible, using a `SERIAL` or other sequence generator as the table's primary key is preferred. However, if your application does make use of OIDs to identify specific rows of a table, it is recommended to create a unique constraint on the OID column of that table, to ensure that OIDs in the table will indeed uniquely identify rows even after counter wrap-around. Avoid assuming that OIDs are unique across tables; if you need a database-wide unique identifier, use the combination of table OID and row OID for the purpose.

Greenplum Database has some special conditions for primary key and unique constraints with regards to columns that are the *distribution key* in a Greenplum table. For a unique constraint to be enforced in Greenplum Database, the table must be hash-distributed (not `DISTRIBUTED RANDOMLY`), and the constraint columns must be the same as (or a superset of) the table's distribution key columns. A primary key constraint is simply a combination of a unique constraint and a not-null constraint.

Greenplum Database automatically creates an index for each unique constraint or primary key constraint to enforce uniqueness. Thus, it is not necessary to create an index explicitly for primary key columns.

Foreign key constraints are not supported in Greenplum Database.

For inherited tables, unique constraints, primary key constraints, indexes and table privileges *are not* inherited in the current implementation.

Examples

Create a table named *rank* in the schema named *baby* and distribute the data using the columns *rank*, *gender*, and *year*:

```
CREATE TABLE baby.rank (id int, rank int, year smallint,
gender char(1), count int ) DISTRIBUTED BY (rank, gender,
year);
```

Create table *films* and table *distributors* (the primary key will be used as the Greenplum distribution key by default):

```
CREATE TABLE films (
code          char(5) CONSTRAINT firstkey PRIMARY KEY,
title        varchar(40) NOT NULL,
did          integer NOT NULL,
date_prod    date,
kind         varchar(10),
len          interval hour to minute
);
```

```
CREATE TABLE distributors (
did          integer PRIMARY KEY DEFAULT nextval('serial'),
name        varchar(40) NOT NULL CHECK (name <> '')
);
```

Create a gzip-compressed, append-only table:

```
CREATE TABLE sales (txn_id int, qty int, date date)
WITH (appendonly=true, compresslevel=5)
DISTRIBUTED BY (txn_id);
```

Create a three level partitioned table using subpartition templates and default partitions at each level:

```
CREATE TABLE sales (id int, year int, month int, day int,
region text)
DISTRIBUTED BY (id)
PARTITION BY RANGE (year)

SUBPARTITION BY RANGE (month)
SUBPARTITION TEMPLATE (
START (1) END (13) EVERY (1),
DEFAULT SUBPARTITION other_months )

SUBPARTITION BY LIST (region)
SUBPARTITION TEMPLATE (
SUBPARTITION usa VALUES ('usa'),
SUBPARTITION europe VALUES ('europe'),
SUBPARTITION asia VALUES ('asia'),
DEFAULT SUBPARTITION other_regions)
```

```
( START (2002) END (2010) EVERY (1),
  DEFAULT PARTITION outlying_years);
```

Compatibility

`CREATE TABLE` command conforms to the SQL standard, with the following exceptions:

- Temporary Tables** — In the SQL standard, temporary tables are defined just once and automatically exist (starting with empty contents) in every session that needs them. Greenplum Database instead requires each session to issue its own `CREATE TEMPORARY TABLE` command for each temporary table to be used. This allows different sessions to use the same temporary table name for different purposes, whereas the standard's approach constrains all instances of a given temporary table name to have the same table structure.

The standard's distinction between global and local temporary tables is not in Greenplum Database. Greenplum Database will accept the `GLOBAL` and `LOCAL` keywords in a temporary table declaration, but they have no effect.

If the `ON COMMIT` clause is omitted, the SQL standard specifies that the default behavior as `ON COMMIT DELETE ROWS`. However, the default behavior in Greenplum Database is `ON COMMIT PRESERVE ROWS`. The `ON COMMIT DROP` option does not exist in the SQL standard.
- Column Check Constraints** — The SQL standard says that `CHECK` column constraints may only refer to the column they apply to; only `CHECK` table constraints may refer to multiple columns. Greenplum Database does not enforce this restriction; it treats column and table check constraints alike.
- NULL Constraint** — The `NULL` constraint is a Greenplum Database extension to the SQL standard that is included for compatibility with some other database systems (and for symmetry with the `NOT NULL` constraint). Since it is the default for any column, its presence is not required.
- Inheritance** — Multiple inheritance via the `INHERITS` clause is a Greenplum Database language extension. SQL:1999 and later define single inheritance using a different syntax and different semantics. SQL:1999-style inheritance is not yet supported by Greenplum Database.
- Partitioning** — Table partitioning via the `PARTITION BY` clause is a Greenplum Database language extension.
- Zero-column tables** — Greenplum Database allows a table of no columns to be created (for example, `CREATE TABLE foo();`). This is an extension from the SQL standard, which does not allow zero-column tables. Zero-column tables are not in themselves very useful, but disallowing them creates odd special cases for `ALTER TABLE DROP COLUMN`, so Greenplum decided to ignore this spec restriction.
- WITH clause** — The `WITH` clause is a Greenplum Database extension; neither storage parameters nor OIDs are in the standard.
- Tablespaces** — The Greenplum Database concept of tablespaces is not part of the SQL standard. The clauses `TABLESPACE` and `USING INDEX TABLESPACE` are extensions.

- **Data Distribution** — The Greenplum Database concept of a parallel or distributed database is not part of the SQL standard. The `DISTRIBUTED` clauses are extensions.

See Also

`ALTER TABLE`, `DROP TABLE`, `CREATE EXTERNAL TABLE`, `CREATE TABLE AS`

CREATE TABLE AS

Defines a new table from the results of a query.

Synopsis

```
CREATE [ [GLOBAL | LOCAL] {TEMPORARY | TEMP} ] TABLE table_name
    [(column_name [, ...] )]
    [ WITH ( storage_parameter=value [, ...] ) ]
    [ON COMMIT {PRESERVE ROWS | DELETE ROWS | DROP}]
    [TABLESPACE tablespace]
    AS query
    [DISTRIBUTED BY (column, [ ... ] ) | DISTRIBUTED RANDOMLY]
```

where *storage_parameter* is:

```
APPENDONLY={TRUE | FALSE}
ORIENTATION={COLUMN | ROW}
COMPRESSTYPE={ZLIB | QUICKLZ}
COMPRESSLEVEL={0-9 | 1}
FILLFACTOR={10-100}
OIDS [=TRUE | FALSE]
```

Description

CREATE TABLE AS creates a table and fills it with data computed by a [SELECT](#) command. The table columns have the names and data types associated with the output columns of the [SELECT](#), however you can override the column names by giving an explicit list of new column names.

CREATE TABLE AS creates a new table and evaluates the query just once to fill the new table initially. The new table will not track subsequent changes to the source tables of the query.

Parameters

GLOBAL | LOCAL

These keywords are present for SQL standard compatibility, but have no effect in Greenplum Database.

TEMPORARY | TEMP

If specified, the new table is created as a temporary table. Temporary tables are automatically dropped at the end of a session, or optionally at the end of the current transaction (see [ON COMMIT](#)). Existing permanent tables with the same name are not visible to the current session while the temporary table exists, unless they are referenced with schema-qualified names. Any indexes created on a temporary table are automatically temporary as well.

table_name

The name (optionally schema-qualified) of the new table to be created.

column_name

The name of a column in the new table. If column names are not provided, they are taken from the output column names of the query. If the table is created from an `EXECUTE` command, a column name list cannot be specified.

WITH (storage_parameter=value)

The `WITH` clause can be used to set storage options for the table or its indexes. Note that you can also set different storage parameters on a particular partition or subpartition by declaring the `WITH` clause in the partition specification. The following storage options are available:

APPENDONLY - Set to `TRUE` to create the table as an append-only table. If `FALSE` or not declared, the table will be created as a regular heap-storage table.

ORIENTATION - Set to `column` for column-oriented storage, or `row` (the default) for row-oriented storage. This option is only valid if `APPENDONLY=TRUE`. Heap-storage tables can only be row-oriented.

COMPRESSTYPE - Set to `ZLIB` (the default) or `QUICKLZ` to specify the type of compression used. QuickLZ uses less CPU power and compresses data faster at a lower compression ratio than zlib. Conversely, zlib provides more compact compression ratios at lower speeds. This option is only valid if `APPENDONLY=TRUE`.

COMPRESSLEVEL - For zlib compression of append-only tables, set to a value between 1 (fastest compression) to 9 (highest compression ratio). QuickLZ compression level can only be set to 1. If not declared, the default is 0 (no compression). This option is only valid if `APPENDONLY=TRUE`.

FILLFACTOR - See `CREATE INDEX` for more information about this index storage parameter.

OIDS - Set to `OIDS=FALSE` (the default) so that rows do not have object identifiers assigned to them. Greenplum strongly recommends that you do not enable OIDS when creating a table. On large tables, such as those in a typical Greenplum Database system, using OIDs for table rows can cause wrap-around of the 32-bit OID counter. Once the counter wraps around, OIDs can no longer be assumed to be unique, which not only makes them useless to user applications, but can also cause problems in the Greenplum Database system catalog tables. In addition, excluding OIDs from a table reduces the space required to store the table on disk by 4 bytes per row, slightly improving performance. OIDs are not allowed on column-oriented tables.

ON COMMIT

The behavior of temporary tables at the end of a transaction block can be controlled using `ON COMMIT`. The three options are:

PRESERVE ROWS

No special action is taken at the ends of transactions for temporary tables. This is the default behavior.

DELETE ROWS

All rows in the temporary table will be deleted at the end of each transaction block. Essentially, an automatic `TRUNCATE` is done at each commit.

DROP

The temporary table will be dropped at the end of the current transaction block.

TABLESPACE *tablespace*

The tablespace is the name of the tablespace in which the new table is to be created. If not specified, the database's default tablespace is used. This is currently not supported in Greenplum Database.

AS *query*

A `SELECT` or `VALUES` command, or an `EXECUTE` command that runs a prepared `SELECT` or `VALUES` query.

DISTRIBUTED BY (*column*, [...]) DISTRIBUTED RANDOMLY

Used to declare the Greenplum Database distribution policy for the table. One or more columns can be used as the distribution key, meaning those columns are used by the hashing algorithm to divide the data evenly across all of the segments. The distribution key should be the primary key of the table or a unique column (or set of columns). If that is not possible, then you may choose to distribute randomly, which will send the data round-robin to the segment instances. If not supplied, then either the `PRIMARY KEY` (if the table has one) or the first eligible column of the table will be used.

Notes

This command is functionally similar to `SELECT INTO`, but it is preferred since it is less likely to be confused with other uses of the `SELECT INTO` syntax. Furthermore, `CREATE TABLE AS` offers a superset of the functionality offered by `SELECT INTO`.

`CREATE TABLE AS` can be used for fast data loading from external table data sources. See `CREATE EXTERNAL TABLE`.

Examples

Create a new table `films_recent` consisting of only recent entries from the table `films`:

```
CREATE TABLE films_recent AS SELECT * FROM films WHERE
date_prod >= '2007-01-01';
```

Create a new temporary table `films_recent`, consisting of only recent entries from the table `films`, using a prepared statement. The new table will be dropped at commit:

```
PREPARE recentfilms(date) AS SELECT * FROM films WHERE
date_prod > $1;
CREATE TEMP TABLE films_recent ON COMMIT DROP AS EXECUTE
recentfilms('2007-01-01');
```

Compatibility

`CREATE TABLE AS` conforms to the SQL standard, with the following exceptions:

- The standard requires parentheses around the subquery clause; in Greenplum Database, these parentheses are optional.
- The standard defines a `WITH [NO] DATA` clause; this is not currently implemented by Greenplum Database. The behavior provided by Greenplum Database is equivalent to the standard's `WITH DATA` case. `WITH NO DATA` can be simulated by appending `LIMIT 0` to the query.
- Greenplum Database handles temporary tables differently from the standard; see `CREATE TABLE` for details.
- The `WITH` clause is a Greenplum Database extension; neither storage parameters nor `OIDs` are in the standard.
- The Greenplum Database concept of tablespaces is not part of the standard. The `TABLESPACE` clause is an extension.

See Also

[CREATE EXTERNAL TABLE](#), [CREATE EXTERNAL TABLE](#), [EXECUTE](#), [SELECT](#), [SELECT INTO](#), [VALUES](#)

CREATE TABLESPACE

Defines a new tablespace.



Note: Tablespaces are currently disabled in Greenplum Database.

Synopsis

```
CREATE TABLESPACE tablespacename [OWNER username]
    LOCATION 'segcontent0dir', 'segcontent1dir', ...
[MIRROR LOCATION 'segcontent0dir', 'segcontent1dir', ...]
[MASTER LOCATION 'mastercontentdir']
```

Description

`CREATE TABLESPACE` registers a new tablespace for your Greenplum Database system. The tablespace name must be distinct from the name of any existing tablespace in the system.

A tablespace allows superusers to define an alternative location on the file system where the data files containing database objects (such as tables and indexes) may reside.

A user with appropriate privileges can pass a tablespace name to `CREATE DATABASE`, `CREATE TABLE`, or `CREATE INDEX` to have the data files for these objects stored within the specified tablespace.

In Greenplum Database, you must declare the tablespace directory location for each segment instance (primary and mirror) in your Greenplum Database system. Primary segments must have a different tablespace location from their corresponding mirror segments. Tablespace locations (and mirror locations) must be declared in order of their corresponding segment content ID number.

To find out how many primary and mirror segment instances are in your system and their corresponding content IDs, run the following query:

```
SELECT content, hostname, definedprimary, datadir FROM
gp_configuration ORDER BY content;
content| hostname | definedprimary | datadir
-----|-----|-----|-----
-1 | gpmaster | t | /dbfast1/gp-1
0 | seghost1 | t | /dbfast1/gp0
0 | seghost2 | f | /dbfast2/gp2
1 | seghost2 | t | /dbfast1/gp1
1 | seghost1 | f | /dbfast2/gp3
```

Note that a primary segment and its corresponding mirror have the same content identifier. A content ID of -1 always indicates the master. In the above example, this Greenplum Database system has a total of 4 segment instances (2 primary and 2 mirror) on 2 segment hosts. Therefore we must specify two tablespace directories for

`LOCATION` (you can use the same location for each primary) and two tablespace directories for `MIRROR LOCATION` (you can use the same location for each mirror, but it must be a different location than what is used for the primaries).

A master instance does not have any user data and its content id is always -1. It is optional to declare a tablespace location for the master (or standby master) in `CREATE TABLESPACE`. If you do not declare `MASTER LOCATION`, the first primary segment tablespace location will be used as the master tablespace location by default.

Parameters

tablespacename

The name of a tablespace to be created. The name cannot begin with `pg_`, as such names are reserved for system tablespaces.

`OWNER username`

The name of the user who will own the tablespace. If omitted, defaults to the user executing the command. Only superusers may create tablespaces, but they can assign ownership of tablespaces to non-superusers.

`LOCATION 'segcontent0dir', 'segcontent1dir', ...`

Declares the tablespace directories used by the primary segments in Greenplum Database. You must give a directory location for each primary segment specified in the order of its corresponding content ID. You may use the same directory location for all primary segments if you wish. The directory must be empty and must be owned by the Greenplum Database system user (`gpadmin`). The directory must be specified by an absolute path name.

`MIRROR LOCATION 'segcontent0dir', 'segcontent1dir', ...`

Required only if mirrors are deployed in your Greenplum Database system. Declares the tablespace directories used by the mirror segments in Greenplum Database. You must give a directory location for each mirror segment specified in the order of its corresponding content ID. You may use the same directory location for all mirror segments if you wish, but the location must be different than what was declared for primaries. The directory must be empty and must be owned by the Greenplum Database system user (`gpadmin`). The directory must be specified by an absolute path name.

`MASTER LOCATION 'mastercontentdir'`

Optional tablespace location on the master instance. If omitted, the first primary segment tablespace location will be used as the master tablespace location.

Notes

Tablespaces are only supported on systems that support symbolic links.

`CREATE TABLESPACE` cannot be executed inside a transaction block.

Although the Greenplum master host does not utilize tablespaces, the tablespace directory locations declared in `CREATE TABLESPACE` must exist on the master host.

Examples

Create the new tablespace by specifying the directory location for each primary segment and each mirror segment:

```
CREATE TABLESPACE mytblspace LOCATION '/dbfast3/tblspc',
    '/dbfast3/tblspc' MIRROR LOCATION '/dbfast4/tblspc',
    '/dbfast4/tblspc' MASTER LOCATION 'dbfast1/tblspc'
```

Compatibility

CREATE TABLESPACE is a Greenplum Database extension.

See Also

[CREATE DATABASE](#), [CREATE TABLE](#), [CREATE INDEX](#), [DROP TABLESPACE](#), [ALTER TABLESPACE](#)

CREATE TRIGGER

Defines a new trigger.

Synopsis

```
CREATE TRIGGER name {BEFORE | AFTER} {event [OR ...]}
  ON table [ FOR [EACH] {ROW | STATEMENT} ]
  EXECUTE PROCEDURE funcname ( arguments )
```

Description

CREATE TRIGGER creates a new trigger. The trigger will be associated with the specified table and will execute the specified function when certain events occur.

Due to the distributed nature of a Greenplum Database system, the use of triggers is somewhat limited. The function used in the trigger must be IMMUTABLE, meaning it cannot use information not directly present in its argument list. The function specified in the trigger also cannot execute any SQL or modify the database in any way. Given that triggers are most often used to alter the database (for example, update these other rows when this row is updated), these limitations offer very little practical use of triggers in Greenplum Database.

If multiple triggers of the same kind are defined for the same event, they will be fired in alphabetical order by name.

SELECT does not modify any rows so you can not create SELECT triggers. Rules and views are more appropriate in such cases.

Parameters

name

The name to give the new trigger. This must be distinct from the name of any other trigger for the same table.

BEFORE AFTER

Determines whether the function is called before or after the event. If the trigger fires before the event, the trigger may skip the operation for the current row, or change the row being inserted (for INSERT and UPDATE operations only). If the trigger fires after the event, all changes, including the last insertion, update, or deletion, are visible to the trigger.

event

Specifies the event that will fire the trigger (INSERT, UPDATE, or DELETE). Multiple events can be specified using OR.

table

The name (optionally schema-qualified) of the table the trigger is for.

FOR EACH ROW
FOR EACH STATEMENT

This specifies whether the trigger procedure should be fired once for every row affected by the trigger event, or just once per SQL statement. If neither is specified, `FOR EACH STATEMENT` is the default. A trigger that is marked `FOR EACH ROW` is called once for every row that the operation modifies. In contrast, a trigger that is marked `FOR EACH STATEMENT` only executes once for any given operation, regardless of how many rows it modifies.

funcname

A user-supplied function that is declared as `IMMUTABLE`, taking no arguments, and returning type `trigger`, which is executed when the trigger fires. This function must not execute SQL or modify the database in any way.

arguments

An optional comma-separated list of arguments to be provided to the function when the trigger is executed. The arguments are literal string constants. Simple names and numeric constants may be written here, too, but they will all be converted to strings. Please check the description of the implementation language of the trigger function about how the trigger arguments are accessible within the function; it may be different from normal function arguments.

Notes

To create a trigger on a table, the user must have the `TRIGGER` privilege on the table.

Examples

Declare the trigger function and then a trigger:

```
CREATE FUNCTION sendmail() RETURNS trigger AS
'$GPHOME/lib/emailtrig.so' LANGUAGE C IMMUTABLE;

CREATE TRIGGER t_sendmail AFTER INSERT OR UPDATE OR DELETE
ON mytable FOR EACH STATEMENT EXECUTE PROCEDURE sendmail();
```

Compatibility

The `CREATE TRIGGER` statement in Greenplum Database implements a subset of the SQL standard. The following functionality is currently missing:

- The current release of Greenplum Database has strict limitations on the function that is called by a trigger, which makes triggers currently not very useful in Greenplum Database.
- SQL allows triggers to fire on updates to specific columns (e.g., `AFTER UPDATE OF col1, col2`).

- SQL allows you to define aliases for the ‘old’ and ‘new’ rows or tables for use in the definition of the triggered action (e.g., `CREATE TRIGGER ... ON tablename REFERENCING OLD ROW AS somename NEW ROW AS othertype ...`). Since Greenplum Database allows trigger procedures to be written in any number of user-defined languages, access to the data is handled in a language-specific way.
- Greenplum Database only allows the execution of a user-defined function for the triggered action. The standard allows the execution of a number of other SQL commands, such as `CREATE TABLE` as the triggered action. This limitation is not hard to work around by creating a user-defined function that executes the desired commands.
- SQL specifies that multiple triggers should be fired in time-of-creation order. Greenplum Database uses name order, which was judged to be more convenient.
- SQL specifies that `BEFORE DELETE` triggers on cascaded deletes fire after the cascaded `DELETE` completes. The Greenplum Database behavior is for `BEFORE DELETE` to always fire before the delete action, even a cascading one. This is considered more consistent. There is also unpredictable behavior when `BEFORE` triggers modify rows that are later to be modified by referential actions. This can lead to constraint violations or stored data that does not honor the referential constraint.
- The ability to specify multiple actions for a single trigger using `OR` is a Greenplum Database extension of the SQL standard.

See Also

[CREATE FUNCTION](#), [ALTER TRIGGER](#), [DROP TRIGGER](#), [CREATE RULE](#)

CREATE TYPE

Defines a new data type.

Synopsis

```
CREATE TYPE name AS ( attribute_name data_type [, ... ] )

CREATE TYPE name (
    INPUT = input_function,
    OUTPUT = output_function
    [, RECEIVE = receive_function]
    [, SEND = send_function]
    [, ANALYZE = analyze_function]
    [, INTERNALLENGTH = {internallength | VARIABLE}]
    [, PASSEDBYVALUE]
    [, ALIGNMENT = alignment]
    [, STORAGE = storage]
    [, DEFAULT = default]
    [, ELEMENT = element]
    [, DELIMITER = delimiter]
)

CREATE TYPE name
```

Description

`CREATE TYPE` registers a new data type for use in the current database. The user who defines a type becomes its owner.

If a schema name is given then the type is created in the specified schema. Otherwise it is created in the current schema. The type name must be distinct from the name of any existing type or domain in the same schema. The type name must also be distinct from the name of any existing table in the same schema.

Composite Types

The first form of `CREATE TYPE` creates a composite type. The composite type is specified by a list of attribute names and data types. This is essentially the same as the row type of a table, but using `CREATE TYPE` avoids the need to create an actual table when all that is wanted is to define a type. A stand-alone composite type is useful as the argument or return type of a function.

Base Types

The second form of `CREATE TYPE` creates a new base type (scalar type). The parameters may appear in any order, not only that shown in the syntax, and most are optional. You must register two or more functions (using `CREATE FUNCTION`) before defining the type. The support functions `input_function` and `output_function` are required, while the functions `receive_function`, `send_function` and `analyze_function` are optional. Generally these functions have to be coded in C or another low-level language. In Greenplum Database, any function used to implement a data type must be defined as `IMMUTABLE`.

The *input_function* converts the type's external textual representation to the internal representation used by the operators and functions defined for the type. *output_function* performs the reverse transformation. The input function may be declared as taking one argument of type `cstring`, or as taking three arguments of types `cstring`, `oid`, `integer`. The first argument is the input text as a C string, the second argument is the type's own OID (except for array types, which instead receive their element type's OID), and the third is the `typmod` of the destination column, if known (-1 will be passed if not). The input function must return a value of the data type itself. Usually, an input function should be declared `STRICT`; if it is not, it will be called with a `NULL` first parameter when reading a `NULL` input value. The function must still return `NULL` in this case, unless it raises an error. (This case is mainly meant to support domain input functions, which may need to reject `NULL` inputs.) The output function must be declared as taking one argument of the new data type. The output function must return type `cstring`. Output functions are not invoked for `NULL` values.

The optional *receive_function* converts the type's external binary representation to the internal representation. If this function is not supplied, the type cannot participate in binary input. The binary representation should be chosen to be cheap to convert to internal form, while being reasonably portable. (For example, the standard integer data types use network byte order as the external binary representation, while the internal representation is in the machine's native byte order.) The receive function should perform adequate checking to ensure that the value is valid. The receive function may be declared as taking one argument of type `internal`, or as taking three arguments of types `internal`, `oid`, `integer`. The first argument is a pointer to a `StringInfo` buffer holding the received byte string; the optional arguments are the same as for the text input function. The receive function must return a value of the data type itself. Usually, a receive function should be declared `STRICT`; if it is not, it will be called with a `NULL` first parameter when reading a `NULL` input value. The function must still return `NULL` in this case, unless it raises an error. (This case is mainly meant to support domain receive functions, which may need to reject `NULL` inputs.) Similarly, the optional *send_function* converts from the internal representation to the external binary representation. If this function is not supplied, the type cannot participate in binary output. The send function must be declared as taking one argument of the new data type. The send function must return type `bytea`. Send functions are not invoked for `NULL` values.

You should at this point be wondering how the input and output functions can be declared to have results or arguments of the new type, when they have to be created before the new type can be created. The answer is that the type should first be defined as a shell type, which is a placeholder type that has no properties except a name and an owner. This is done by issuing the command `CREATE TYPE name`, with no additional parameters. Then the I/O functions can be defined referencing the shell type. Finally, `CREATE TYPE` with a full definition replaces the shell entry with a complete, valid type definition, after which the new type can be used normally.

The optional *analyze_function* performs type-specific statistics collection for columns of the data type. By default, `ANALYZE` will attempt to gather statistics using the type's 'equals' and 'less-than' operators, if there is a default b-tree operator class for the type. For non-scalar types this behavior is likely to be unsuitable, so it can be overridden by specifying a custom analysis function. The analysis function must be

declared to take a single argument of type `internal`, and return a `boolean` result. The detailed API for analysis functions appears in `$GPHOME/include/commands/vacuum.h`.

While the details of the new type's internal representation are only known to the I/O functions and other functions you create to work with the type, there are several properties of the internal representation that must be declared to Greenplum Database. Foremost of these is `internallength`. Base data types can be fixed-length, in which case `internallength` is a positive integer, or variable length, indicated by setting `internallength` to `VARIABLE`. (Internally, this is represented by setting `typelen` to `-1`.) The internal representation of all variable-length types must start with a 4-byte integer giving the total length of this value of the type.

The optional flag `PASSEDBYVALUE` indicates that values of this data type are passed by value, rather than by reference. You may not pass by value types whose internal representation is larger than the size of the `Datum` type (4 bytes on most machines, 8 bytes on a few).

The `alignment` parameter specifies the storage alignment required for the data type. The allowed values equate to alignment on 1, 2, 4, or 8 byte boundaries. Note that variable-length types must have an alignment of at least 4, since they necessarily contain an `int4` as their first component.

The `storage` parameter allows selection of storage strategies for variable-length data types. (Only `plain` is allowed for fixed-length types.) `plain` specifies that data of the type will always be stored in-line and not compressed. `extended` specifies that the system will first try to compress a long data value, and will move the value out of the main table row if it's still too long. `external` allows the value to be moved out of the main table, but the system will not try to compress it. `main` allows compression, but discourages moving the value out of the main table. (Data items with this storage strategy may still be moved out of the main table if there is no other way to make a row fit, but they will be kept in the main table preferentially over `extended` and `external` items.)

A default value may be specified, in case a user wants columns of the data type to default to something other than the null value. Specify the default with the `DEFAULT` key word. (Such a default may be overridden by an explicit `DEFAULT` clause attached to a particular column.)

To indicate that a type is an array, specify the type of the array elements using the `ELEMENT` key word. For example, to define an array of 4-byte integers (`int4`), specify `ELEMENT = int4`. More details about array types appear below.

To indicate the delimiter to be used between values in the external representation of arrays of this type, `delimiter` can be set to a specific character. The default delimiter is the comma (`,`). Note that the delimiter is associated with the array element type, not the array type itself.

Array Types

Whenever a user-defined base data type is created, Greenplum Database automatically creates an associated array type, whose name consists of the base type's name prepended with an underscore. The parser understands this naming convention, and

translates requests for columns of type `foo[]` into requests for type `_foo`. The implicitly-created array type is variable length and uses the built-in input and output functions `array_in` and `array_out`.

You might reasonably ask why there is an `ELEMENT` option, if the system makes the correct array type automatically. The only case where it's useful to use `ELEMENT` is when you are making a fixed-length type that happens to be internally an array of a number of identical things, and you want to allow these things to be accessed directly by subscripting, in addition to whatever operations you plan to provide for the type as a whole. For example, type `name` allows its constituent `char` elements to be accessed this way. A 2-D point type could allow its two component numbers to be accessed like `point[0]` and `point[1]`. Note that this facility only works for fixed-length types whose internal form is exactly a sequence of identical fixed-length fields. A subscriptable variable-length type must have the generalized internal representation used by `array_in` and `array_out`. For historical reasons, subscripting of fixed-length array types starts from zero, rather than from one as for variable-length arrays.

Parameters

name

The name (optionally schema-qualified) of a type to be created.

attribute_name

The name of an attribute (column) for the composite type.

data_type

The name of an existing data type to become a column of the composite type.

input_function

The name of a function that converts data from the type's external textual form to its internal form.

output_function

The name of a function that converts data from the type's internal form to its external textual form.

receive_function

The name of a function that converts data from the type's external binary form to its internal form.

send_function

The name of a function that converts data from the type's internal form to its external binary form.

analyze_function

The name of a function that performs statistical analysis for the data type.

internallength

A numeric constant that specifies the length in bytes of the new type's internal representation. The default assumption is that it is variable-length.

alignment

The storage alignment requirement of the data type. Must be one of `char`, `int2`, `int4`, or `double`. The default is `int4`.

storage

The storage strategy for the data type. Must be one of `plain`, `external`, `extended`, or `main`. The default is `plain`.

default

The default value for the data type. If this is omitted, the default is null.

element

The type being created is an array; this specifies the type of the array elements.

delimiter

The delimiter character to be used between values in arrays made of this type.

Notes

User-defined type names cannot begin with the underscore character (`_`) and can only be 62 characters long (or in general `NAMEDATALEN - 2`, rather than the `NAMEDATALEN - 1` characters allowed for other names). Type names beginning with underscore are reserved for internally-created array type names.

Because there are no restrictions on use of a data type once it's been created, creating a base type is tantamount to granting public execute permission on the functions mentioned in the type definition. (The creator of the type is therefore required to own these functions.) This is usually not an issue for the sorts of functions that are useful in a type definition. But you might want to think twice before designing a type in a way that would require 'secret' information to be used while converting it to or from external form.

Before Greenplum Database version 2.4, the syntax `CREATE TYPE name` did not exist. The way to create a new base type was to create its input function first. In this approach, Greenplum Database will first see the name of the new data type as the return type of the input function. The shell type is implicitly created in this situation, and then it can be referenced in the definitions of the remaining I/O functions. This approach still works, but is deprecated and may be disallowed in some future release. Also, to avoid accidentally cluttering the catalogs with shell types as a result of simple typos in function definitions, a shell type will only be made this way when the input function is written in C.

Examples

This example creates a composite type and uses it in a function definition:

```
CREATE TYPE compfoo AS (f1 int, f2 text);

CREATE FUNCTION getfoo() RETURNS SETOF compfoo AS $$
    SELECT fooid, fooname FROM foo
$$ LANGUAGE SQL;
```

This example creates the base data type *box* and then uses the type in a table definition:

```
CREATE TYPE box;

CREATE FUNCTION my_box_in_function(cstring) RETURNS box AS
... ;
CREATE FUNCTION my_box_out_function(box) RETURNS cstring AS
... ;

CREATE TYPE box (
    INTERNALLENGTH = 16,
    INPUT = my_box_in_function,
    OUTPUT = my_box_out_function
);

CREATE TABLE myboxes (
    id integer,
    description box
);
```

If the internal structure of *box* were an array of four `float4` elements, we might instead use:

```
CREATE TYPE box (
    INTERNALLENGTH = 16,
    INPUT = my_box_in_function,
    OUTPUT = my_box_out_function,
    ELEMENT = float4
);
```

which would allow a *box* value's component numbers to be accessed by subscripting. Otherwise the type behaves the same as before.

This example creates a large object type and uses it in a table definition:

```
CREATE TYPE bigobj (
    INPUT = lo_filein, OUTPUT = lo_fileout,
    INTERNALLENGTH = VARIABLE
);

CREATE TABLE big_objs (
```

```
        id integer,  
        obj bigobj  
    );
```

Compatibility

This `CREATE TYPE` command is a Greenplum Database extension. There is a `CREATE TYPE` statement in the SQL standard that is rather different in detail.

See Also

`CREATE FUNCTION`, `ALTER TYPE`, `DROP TYPE`, `CREATE DOMAIN`

CREATE USER

Defines a new database role with the `LOGIN` privilege by default.

Synopsis

```
CREATE USER name [ [WITH] option [ ... ] ]
```

where *option* can be:

```

SUPERUSER | NOSUPERUSER
| CREATEDB | NOCREATEDB
| CREATEROLE | NOCREATEROLE
| CREATEUSER | NOCREATEUSER
| INHERIT | NOINHERIT
| LOGIN | NOLOGIN
| [ ENCRYPTED | UNENCRYPTED ] PASSWORD 'password'
| VALID UNTIL 'timestamp'
| IN ROLE rolename [, ...]
| IN GROUP rolename [, ...]
| ROLE rolename [, ...]
| ADMIN rolename [, ...]
| USER rolename [, ...]
| SYSID uid

```

Description

As of Greenplum Database release 2.2, `CREATE USER` has been replaced by `CREATE ROLE`, although it is still accepted for backwards compatibility.

The only difference between `CREATE ROLE` and `CREATE USER` is that `LOGIN` is assumed by default with `CREATE USER`, whereas `NOLOGIN` is assumed by default with `CREATE ROLE`.

Compatibility

There is no `CREATE USER` statement in the SQL standard.

See Also

[CREATE ROLE](#)

CREATE VIEW

Defines a new view.

Synopsis

```
CREATE [OR REPLACE] [TEMP | TEMPORARY] VIEW name
  [ ( column_name [, ...] ) ]
  AS query
```

Description

`CREATE VIEW` defines a view of a query. The view is not physically materialized. Instead, the query is run every time the view is referenced in a query.

`CREATE OR REPLACE VIEW` is similar, but if a view of the same name already exists, it is replaced. You can only replace a view with a new query that generates the identical set of columns (same column names and data types).

If a schema name is given then the view is created in the specified schema. Otherwise it is created in the current schema. Temporary views exist in a special schema, so a schema name may not be given when creating a temporary view. The name of the view must be distinct from the name of any other view, table, sequence, or index in the same schema.

Parameters

TEMPORARY | TEMP

If specified, the view is created as a temporary view. Temporary views are automatically dropped at the end of the current session. Existing permanent relations with the same name are not visible to the current session while the temporary view exists, unless they are referenced with schema-qualified names. If any of the tables referenced by the view are temporary, the view is created as a temporary view (whether `TEMPORARY` is specified or not).

name

The name (optionally schema-qualified) of a view to be created.

column_name

An optional list of names to be used for columns of the view. If not given, the column names are deduced from the query.

query

A `SELECT` or `VALUES` command which will provide the columns and rows of the view.

Notes

Views in Greenplum Database are read only. The system will not allow an insert, update, or delete on a view. You can get the effect of an updatable view by creating rewrite rules on the view into appropriate actions on other tables. For more information see `CREATE RULE`.

Be careful that the names and data types of the view's columns will be assigned the way you want. For example:

```
CREATE VIEW vista AS SELECT 'Hello World';
```

is bad form in two ways: the column name defaults to `?column?`, and the column data type defaults to `unknown`. If you want a string literal in a view's result, use something like:

```
CREATE VIEW vista AS SELECT text 'Hello World' AS hello;
```

Access to tables referenced in the view is determined by permissions of the view owner not the current user (even if the current user is a superuser). This can be confusing in the case of superusers, since superusers typically have access to all objects. In the case of a view, even superusers must be explicitly granted access to tables referenced in the view if they are not the owner of the view.

However, functions called in the view are treated the same as if they had been called directly from the query using the view. Therefore the user of a view must have permissions to call any functions used by the view.

If you create a view with an `ORDER BY` clause, the `ORDER BY` clause is ignored when you do a `SELECT` from the view.

Examples

Create a view consisting of all comedy films:

```
CREATE VIEW comedies AS SELECT * FROM films WHERE kind =
'comedy';
```

Create a view that gets the top ten ranked baby names:

```
CREATE VIEW topten AS SELECT name, rank, gender, year FROM
names, rank WHERE rank < '11' AND names.id=rank.id;
```

Compatibility

The SQL standard specifies some additional capabilities for the `CREATE VIEW` statement that are not in Greenplum Database. The optional clauses for the full SQL command in the standard are:

- **CHECK OPTION** — This option has to do with updatable views. All `INSERT` and `UPDATE` commands on the view will be checked to ensure data satisfy the view-defining condition (that is, the new data would be visible through the view). If they do not, the update will be rejected.
- **LOCAL** — Check for integrity on this view.

- **CASCADED** — Check for integrity on this view and on any dependent view. **CASCADED** is assumed if neither **CASCADED** nor **LOCAL** is specified.

CREATE OR REPLACE VIEW is a Greenplum Database language extension. So is the concept of a temporary view.

See Also

[SELECT](#), [DROP VIEW](#)

DEALLOCATE

Deallocates a prepared statement.

Synopsis

```
DEALLOCATE [PREPARE] name
```

Description

`DEALLOCATE` is used to deallocate a previously prepared SQL statement. If you do not explicitly deallocate a prepared statement, it is deallocated when the session ends.

For more information on prepared statements, see [PREPARE](#).

Parameters

PREPARE

Optional key word which is ignored.

name

The name of the prepared statement to deallocate.

Examples

Deallocated the previously prepared statement named *insert_names*:

```
DEALLOCATE insert_names;
```

Compatibility

The SQL standard includes a `DEALLOCATE` statement, but it is only for use in embedded SQL.

See Also

[EXECUTE](#), [PREPARE](#)

DECLARE

Defines a cursor.

Synopsis

```
DECLARE name [BINARY] [INSENSITIVE] [NO SCROLL] CURSOR
        [{WITH | WITHOUT} HOLD]
        FOR query [FOR READ ONLY]
```

Description

DECLARE allows a user to create cursors, which can be used to retrieve a small number of rows at a time out of a larger query. Cursors can return data either in text or in binary format using `FETCH`.

Normal cursors return data in text format, the same as a `SELECT` would produce. Since data is stored natively in binary format, the system must do a conversion to produce the text format. Once the information comes back in text form, the client application may need to convert it to a binary format to manipulate it. In addition, data in the text format is often larger in size than in the binary format. Binary cursors return the data in a binary representation that may be more easily manipulated. Nevertheless, if you intend to display the data as text anyway, retrieving it in text form will save you some effort on the client side.

As an example, if a query returns a value of one from an integer column, you would get a string of 1 with a default cursor whereas with a binary cursor you would get a 4-byte field containing the internal representation of the value (in big-endian byte order).

Binary cursors should be used carefully. Many applications, including `psql`, are not prepared to handle binary cursors and expect data to come back in the text format.

Note: When the client application uses the ‘extended query’ protocol to issue a `FETCH` command, the Bind protocol message specifies whether data is to be retrieved in text or binary format. This choice overrides the way that the cursor is defined. The concept of a binary cursor as such is thus obsolete when using extended query protocol — any cursor can be treated as either text or binary.

Parameters

name

The name of the cursor to be created.

BINARY

Causes the cursor to return data in binary rather than in text format.

INSENSITIVE

Indicates that data retrieved from the cursor should be unaffected by updates to the tables underlying the cursor while the cursor exists. In Greenplum Database, all cursors are insensitive. This key word currently has no effect and is present for compatibility with the SQL standard.

NO SCROLL

A cursor cannot be used to retrieve rows in a nonsequential fashion. This is the default behavior in Greenplum Database, since scrollable cursors (**SCROLL**) are not supported.

**WITH HOLD
WITHOUT HOLD**

WITH HOLD specifies that the cursor may continue to be used after the transaction that created it successfully commits. **WITHOUT HOLD** specifies that the cursor cannot be used outside of the transaction that created it. **WITHOUT HOLD** is the default.

query

A **SELECT** or **VALUES** command which will provide the rows to be returned by the cursor.

FOR READ ONLY

Cursors can only be used in a read-only mode in Greenplum Database. Greenplum Database does not support updateable cursors (**FOR UPDATE**), so this is the default behavior.

Notes

Unless **WITH HOLD** is specified, the cursor created by this command can only be used within the current transaction. Thus, **DECLARE** without **WITH HOLD** is useless outside a transaction block: the cursor would survive only to the completion of the statement. Therefore Greenplum Database reports an error if this command is used outside a transaction block. Use **BEGIN**, **COMMIT** and **ROLLBACK** to define a transaction block.

If **WITH HOLD** is specified and the transaction that created the cursor successfully commits, the cursor can continue to be accessed by subsequent transactions in the same session. (But if the creating transaction is aborted, the cursor is removed.) A cursor created with **WITH HOLD** is closed when an explicit **CLOSE** command is issued on it, or the session ends. In the current implementation, the rows represented by a held cursor are copied into a temporary file or memory area so that they remain available for subsequent transactions.

Scrollable cursors are not currently supported in Greenplum Database. You can only use **FETCH** to move the cursor position forward, not backwards.

You can see all available cursors by querying the *pg_cursors* system view.

Examples

Declare a cursor:

```
DECLARE mycursor CURSOR FOR SELECT * FROM mytable;
```

Compatibility

SQL standard allows cursors only in embedded SQL and in modules. Greenplum Database permits cursors to be used interactively.

Greenplum Database does not implement an `OPEN` statement for cursors. A cursor is considered to be open when it is declared.

The SQL standard allows cursors to update table data. All Greenplum Database cursors are read only.

The SQL standard allows cursors to move both forward and backward. All Greenplum Database cursors are forward moving only (not scrollable).

Binary cursors are a Greenplum Database extension.

See Also

[CLOSE](#), [FETCH](#), [MOVE](#), [SELECT](#)

DELETE

Deletes rows from a table.

Synopsis

```
DELETE FROM [ONLY] table [[AS] alias]
      [USING usinglist]
      [WHERE condition]
```

Description

`DELETE` deletes rows that satisfy the `WHERE` clause from the specified table. If the `WHERE` clause is absent, the effect is to delete all rows in the table. The result is a valid, but empty table.

By default, `DELETE` will delete rows in the specified table and all its child tables. If you wish to delete only from the specific table mentioned, you must use the `ONLY` clause.

There are two ways to delete rows in a table using information contained in other tables in the database: using sub-selects, or specifying additional tables in the `USING` clause. Which technique is more appropriate depends on the specific circumstances. Note that if you do join two or more tables in a `DELETE` command, those tables must have the same Greenplum distribution key column(s) and also be joined on the distribution key column(s).

You must have the `DELETE` privilege on the table to delete from it.

Outputs

On successful completion, a `DELETE` command returns a command tag of the form

```
DELETE count
```

The count is the number of rows deleted. If count is 0, no rows matched the condition (this is not considered an error).

Parameters

ONLY

If specified, delete rows from the named table only. When not specified, any tables inheriting from the named table are also processed.

table

The name (optionally schema-qualified) of an existing table.

alias

A substitute name for the target table. When an alias is provided, it completely hides the actual name of the table. For example, given `DELETE FROM foo AS f`, the remainder of the `DELETE` statement must refer to this table as `f` not `foo`.

usinglist

A list of table expressions, allowing columns from other tables to appear in the `WHERE` condition. This is similar to the list of tables that can be specified in the `FROM` Clause of a `SELECT` statement; for example, an alias for the table name can be specified. Do not repeat the target table in the `usinglist`, unless you wish to set up a self-join.

condition

An expression returning a value of type `boolean`, which determines the rows that are to be deleted.

Notes

You cannot use `STABLE` or `VOLATILE` functions in a `DELETE` statement if mirrors are enabled. This can potentially cause the primary segment and its mirror to become out-of-sync because the command is run first on the primary and then a second time on the mirror in the current Greenplum Database implementation.

Greenplum Database lets you reference columns of other tables in the `WHERE` condition by specifying the other tables in the `USING` clause. For example, to the name *Hannah* from the *rank* table, one might do:

```
DELETE FROM rank USING names WHERE names.id = rank.id AND
name = 'Hannah';
```

What is essentially happening here is a join between *rank* and *names*, with all successfully joined rows being marked for deletion. This syntax is not standard. However, this join style is usually easier to write and faster to execute than a more standard sub-select style, such as:

```
DELETE FROM rank WHERE id IN (SELECT id FROM names WHERE name
= 'Hannah');
```

Note also that in Greenplum Database, any tables that are joined in a delete operation must have the *same Greenplum distribution key* and be joined on the column(s) that make up the distribution key (equijoin).

Examples

Delete all films but musicals:

```
DELETE FROM films WHERE kind <> 'Musical';
```

Clear the table films:

```
DELETE FROM films;
```

Delete using an equijoin (assuming both Greenplum Database tables are distributed on the *id* column):

```
DELETE FROM rank USING names WHERE names.id = rank.id AND
name = 'Hannah';
```

Compatibility

This command conforms to the SQL standard, except that the `USING` clause is a Greenplum Database extension.

See Also

[TRUNCATE](#)

DROP AGGREGATE

Removes an aggregate function.

Synopsis

```
DROP AGGREGATE [IF EXISTS] name ( type [, ...] ) [CASCADE |  
RESTRICT]
```

Description

DROP AGGREGATE will delete an existing aggregate function. To execute this command the current user must be the owner of the aggregate function.

Parameters

IF EXISTS

Do not throw an error if the aggregate does not exist. A notice is issued in this case.

name

The name (optionally schema-qualified) of an existing aggregate function.

type

An input data type on which the aggregate function operates. To reference a zero-argument aggregate function, write * in place of the list of input data types.

CASCADE

Automatically drop objects that depend on the aggregate function.

RESTRICT

Refuse to drop the aggregate function if any objects depend on it. This is the default.

Examples

To remove the aggregate function *myavg* for type *integer*:

```
DROP AGGREGATE myavg(integer);
```

Compatibility

There is no DROP AGGREGATE statement in the SQL standard.

See Also

[ALTER AGGREGATE](#), [CREATE AGGREGATE](#)

DROP CAST

Removes a cast.

Synopsis

```
DROP CAST [IF EXISTS] (sourcetype AS targettype) [CASCADE |  
RESTRICT]
```

Description

`DROP CAST` will delete a previously defined cast. To be able to drop a cast, you must own the source or the target data type. These are the same privileges that are required to create a cast.

Parameters

IF EXISTS

Do not throw an error if the cast does not exist. A notice is issued in this case.

sourcetype

The name of the source data type of the cast.

targettype

The name of the target data type of the cast.

CASCADE

RESTRICT

These keywords have no effect since there are no dependencies on casts.

Examples

To drop the cast from type *text* to type *int*:

```
DROP CAST (text AS int);
```

Compatibility

There `DROP CAST` command conforms to the SQL standard.

See Also

[CREATE CAST](#)

DROP CONVERSION

Removes a conversion.

Synopsis

```
DROP CONVERSION [IF EXISTS] name [CASCADE | RESTRICT]
```

Description

`DROP CONVERSION` removes a previously defined conversion. To be able to drop a conversion, you must own the conversion.

Parameters

IF EXISTS

Do not throw an error if the conversion does not exist. A notice is issued in this case.

name

The name of the conversion. The conversion name may be schema-qualified.

CASCADE

RESTRICT

These keywords have no effect since there are no dependencies on conversions.

Examples

Drop the conversion named *myname*:

```
DROP CONVERSION myname;
```

Compatibility

There is no `DROP CONVERSION` statement in the SQL standard.

See Also

[ALTER CONVERSION](#), [CREATE CONVERSION](#)

DROP DATABASE

Removes a database.

Synopsis

```
DROP DATABASE [IF EXISTS] name
```

Description

`DROP DATABASE` drops a database. It removes the catalog entries for the database and deletes the directory containing the data. It can only be executed by the database owner. Also, it cannot be executed while you or anyone else are connected to the target database. (Connect to *template1* or any other database to issue this command.)

`DROP DATABASE` cannot be undone. Use it with care!

Parameters

IF EXISTS

Do not throw an error if the database does not exist. A notice is issued in this case.

name

The name of the database to remove.

Notes

`DROP DATABASE` cannot be executed inside a transaction block.

This command cannot be executed while connected to the target database. Thus, it might be more convenient to use the program `dropdb` instead, which is a wrapper around this command.

Examples

Drop the database named *testdb*:

```
DROP DATABASE testdb;
```

Compatibility

There is no `DROP DATABASE` statement in the SQL standard.

See Also

[ALTER DATABASE](#), [CREATE DATABASE](#)

DROP DOMAIN

Removes a domain.

Synopsis

```
DROP DOMAIN [IF EXISTS] name [, ...] [CASCADE | RESTRICT]
```

Description

`DROP DOMAIN` removes a previously defined domain. You must be the owner of a domain to drop it.

Parameters

IF EXISTS

Do not throw an error if the domain does not exist. A notice is issued in this case.

name

The name (optionally schema-qualified) of an existing domain.

CASCADE

Automatically drop objects that depend on the domain (such as table columns).

RESTRICT

Refuse to drop the domain if any objects depend on it. This is the default.

Examples

Drop the domain named *zipcode*:

```
DROP DOMAIN zipcode;
```

Compatibility

This command conforms to the SQL standard, except for the `IF EXISTS` option, which is a Greenplum Database extension.

See Also

[ALTER DOMAIN](#), [CREATE DOMAIN](#)

DROP EXTERNAL TABLE

Removes an external or web table definition.

Synopsis

```
DROP EXTERNAL [WEB] TABLE [IF EXISTS] name [CASCADE | RESTRICT]
```

Description

`DROP EXTERNAL TABLE` drops an existing external or web table definition from the database system. The external data sources or files are not deleted. To execute this command you must be the owner of the external table.

Parameters

WEB

Optional keyword for dropping external web tables.

IF EXISTS

Do not throw an error if the external table does not exist. A notice is issued in this case.

name

The name (optionally schema-qualified) of an existing external table.

CASCADE

Automatically drop objects that depend on the external table (such as views).

RESTRICT

Refuse to drop the external table if any objects depend on it. This is the default.

Examples

Remove the external table named *staging* if it exists:

```
DROP EXTERNAL TABLE IF EXISTS staging;
```

Compatibility

There is no `DROP EXTERNAL TABLE` statement in the SQL standard.

See Also

[CREATE EXTERNAL TABLE](#)

DROP FUNCTION

Removes a function.

Synopsis

```
DROP FUNCTION [IF EXISTS] name ( [ [argmode] [argname] argtype
[, ...] ] ) [CASCADE | RESTRICT]
```

Description

`DROP FUNCTION` removes the definition of an existing function. To execute this command the user must be the owner of the function. The argument types to the function must be specified, since several different functions may exist with the same name and different argument lists.

Parameters

IF EXISTS

Do not throw an error if the function does not exist. A notice is issued in this case.

name

The name (optionally schema-qualified) of an existing function.

argmode

The mode of an argument: either `IN`, `OUT`, or `INOUT`. If omitted, the default is `IN`.

Note that `DROP FUNCTION` does not actually pay any attention to `OUT` arguments, since only the input arguments are needed to determine the function's identity. So it is sufficient to list the `IN` and `INOUT` arguments.

argname

The name of an argument. Note that `DROP FUNCTION` does not actually pay any attention to argument names, since only the argument data types are needed to determine the function's identity.

argtype

The data type(s) of the function's arguments (optionally schema-qualified), if any.

CASCADE

Automatically drop objects that depend on the function (such as operators or triggers).

RESTRICT

Refuse to drop the function if any objects depend on it. This is the default.

Examples

Drop the square root function:

```
DROP FUNCTION sqrt(integer);
```

Compatibility

A `DROP FUNCTION` statement is defined in the SQL standard, but it is not compatible with this command.

See Also

[CREATE FUNCTION](#), [ALTER FUNCTION](#)

DROP GROUP

Removes a database role.

Synopsis

```
DROP GROUP [IF EXISTS] name [, ...]
```

Description

`DROP GROUP` is an obsolete command, though still accepted for backwards compatibility. Groups (and users) have been superseded by the more general concept of roles. See [DROP ROLE](#) for more information.

Parameters

IF EXISTS

Do not throw an error if the role does not exist. A notice is issued in this case.

name

The name of an existing role.

Compatibility

There is no `DROP GROUP` statement in the SQL standard.

See Also

[DROP ROLE](#)

DROP INDEX

Removes an index.

Synopsis

```
DROP INDEX [IF EXISTS] name [, ...] [CASCADE | RESTRICT]
```

Description

`DROP INDEX` drops an existing index from the database system. To execute this command you must be the owner of the index.

Parameters

IF EXISTS

Do not throw an error if the index does not exist. A notice is issued in this case.

name

The name (optionally schema-qualified) of an existing index.

CASCADE

Automatically drop objects that depend on the index.

RESTRICT

Refuse to drop the index if any objects depend on it. This is the default.

Examples

Remove the index *title_idx*:

```
DROP INDEX title_idx;
```

Compatibility

`DROP INDEX` is a Greenplum Database language extension. There are no provisions for indexes in the SQL standard.

See Also

[ALTER INDEX](#), [CREATE INDEX](#), [REINDEX](#)

DROP LANGUAGE

Removes a procedural language.

Synopsis

```
DROP [PROCEDURAL] LANGUAGE [IF EXISTS] name [CASCADE | RESTRICT]
```

Description

`DROP LANGUAGE` will remove the definition of the previously registered procedural language. You must be a superuser to drop a language.

Parameters

PROCEDURAL

Optional keyword - has no effect.

IF EXISTS

Do not throw an error if the language does not exist. A notice is issued in this case.

name

The name of an existing procedural language. For backward compatibility, the name may be enclosed by single quotes.

CASCADE

Automatically drop objects that depend on the language (such as functions written in that language).

RESTRICT

Refuse to drop the language if any objects depend on it. This is the default.

Examples

Remove the procedural language *plsample*:

```
DROP LANGUAGE plsample;
```

Compatibility

There is no `DROP LANGUAGE` statement in the SQL standard.

See Also

[ALTER LANGUAGE](#), [CREATE LANGUAGE](#)

DROP OPERATOR

Removes an operator.

Synopsis

```
DROP OPERATOR [IF EXISTS] name ( {lefttype | NONE} , {righttype
| NONE} ) [CASCADE | RESTRICT]
```

Description

DROP OPERATOR drops an existing operator from the database system. To execute this command you must be the owner of the operator.

Parameters

IF EXISTS

Do not throw an error if the operator does not exist. A notice is issued in this case.

name

The name (optionally schema-qualified) of an existing operator.

lefttype

The data type of the operator's left operand; write NONE if the operator has no left operand.

righttype

The data type of the operator's right operand; write NONE if the operator has no right operand.

CASCADE

Automatically drop objects that depend on the operator.

RESTRICT

Refuse to drop the operator if any objects depend on it. This is the default.

Examples

Remove the power operator a^b for type *integer*:

```
DROP OPERATOR ^ (integer, integer);
```

Remove the left unary bitwise complement operator $\sim b$ for type *bit*:

```
DROP OPERATOR ~ (none, bit);
```

Remove the right unary factorial operator $x!$ for type *bigint*:

```
DROP OPERATOR ! (bigint, none);
```

Compatibility

There is no `DROP OPERATOR` statement in the SQL standard.

See Also

`ALTER OPERATOR`, `CREATE OPERATOR`

DROP OPERATOR CLASS

Removes an operator class.

Synopsis

```
DROP OPERATOR CLASS [IF EXISTS] name USING index_method [CASCADE
| RESTRICT]
```

Description

`DROP OPERATOR` drops an existing operator class. To execute this command you must be the owner of the operator class.

Parameters

IF EXISTS

Do not throw an error if the operator class does not exist. A notice is issued in this case.

name

The name (optionally schema-qualified) of an existing operator class.

index_method

The name of the index access method the operator class is for.

CASCADE

Automatically drop objects that depend on the operator class.

RESTRICT

Refuse to drop the operator class if any objects depend on it. This is the default.

Examples

Remove the B-tree operator class *widget_ops*:

```
DROP OPERATOR CLASS widget_ops USING btree;
```

This command will not succeed if there are any existing indexes that use the operator class. Add `CASCADE` to drop such indexes along with the operator class.

Compatibility

There is no `DROP OPERATOR CLASS` statement in the SQL standard.

See Also

[ALTER OPERATOR CLASS](#), [CREATE OPERATOR CLASS](#)

DROP OWNED

Removes database objects owned by a database role.

Synopsis

```
DROP OWNED BY name [, ...] [CASCADE | RESTRICT]
```

Description

`DROP OWNED` drops all the objects in the current database that are owned by one of the specified roles. Any privileges granted to the given roles on objects in the current database will also be revoked.

Parameters

name

The name of a role whose objects will be dropped, and whose privileges will be revoked.

CASCADE

Automatically drop objects that depend on the affected objects.

RESTRICT

Refuse to drop the objects owned by a role if any other database objects depend on one of the affected objects. This is the default.

Notes

`DROP OWNED` is often used to prepare for the removal of one or more roles. Because `DROP OWNED` only affects the objects in the current database, it is usually necessary to execute this command in each database that contains objects owned by a role that is to be removed.

Using the `CASCADE` option may make the command recurse to objects owned by other users.

The `REASSIGN OWNED` command is an alternative that reassigns the ownership of all the database objects owned by one or more roles.

Examples

Remove any database objects owned by the role named *sally*:

```
DROP OWNED BY sally;
```

Compatibility

The `DROP OWNED` statement is a Greenplum Database extension.

See Also

REASSIGN OWNED, DROP ROLE

DROP RESOURCE QUEUE

Removes a resource queue.

Synopsis

```
DROP RESOURCE QUEUE queue_name
```

Description

This command removes a workload management resource queue from Greenplum Database. To drop a resource queue, the queue cannot have any roles assigned to it, nor can it have any statements waiting in the queue. Only a superuser can drop a resource queue.

Parameters

queue_name

The name of a resource queue to remove.

Notes

Use [ALTER ROLE](#) to remove a user from a resource queue.

To see all the currently active queries for all resource queues, perform the following query of the `pg_locks` table joined with the `pg_roles` and `pg_resqueue` tables:

```
SELECT rolname, rsqname, locktype, objid, transaction, pid,
       mode, granted FROM pg_roles, pg_resqueue, pg_locks WHERE
       pg_roles.rolresqueue=pg_locks.objid AND
       pg_locks.objid=pg_resqueue.oid;
```

To see the roles assigned to a resource queue, perform the following query of the `pg_roles` and `pg_resqueue` system catalog tables:

```
SELECT rolname, rsqname FROM pg_roles, pg_resqueue WHERE
       pg_roles.rolresqueue=pg_resqueue.oid;
```

Examples

Remove a role from a resource queue:

```
ALTER ROLE bob RESOURCE QUEUE NONE;
```

Remove the resource queue named *adhoc*:

```
DROP RESOURCE QUEUE adhoc;
```

Compatibility

The `DROP RESOURCE QUEUE` statement is a Greenplum Database extension.

See Also

`ALTER RESOURCE QUEUE`, `CREATE RESOURCE QUEUE`, `ALTER ROLE`

DROP ROLE

Removes a database role.

Synopsis

```
DROP ROLE [IF EXISTS] name [, ...]
```

Description

`DROP ROLE` removes the specified role(s). To drop a superuser role, you must be a superuser yourself. To drop non-superuser roles, you must have `CREATEROLE` privilege.

A role cannot be removed if it is still referenced in any database; an error will be raised if so. Before dropping the role, you must drop all the objects it owns (or reassign their ownership) and revoke any privileges the role has been granted. The `REASSIGN OWNED` and `DROP OWNED` commands can be useful for this purpose.

However, it is not necessary to remove role memberships involving the role; `DROP ROLE` automatically revokes any memberships of the target role in other roles, and of other roles in the target role. The other roles are not dropped nor otherwise affected.

Parameters

IF EXISTS

Do not throw an error if the role does not exist. A notice is issued in this case.

name

The name of the role to remove.

Examples

Remove the roles named *sally* and *bob*:

```
DROP ROLE sally, bob;
```

Compatibility

The SQL standard defines `DROP ROLE`, but it allows only one role to be dropped at a time, and it specifies different privilege requirements than Greenplum Database uses.

See Also

[REASSIGN OWNED](#), [DROP OWNED](#), [CREATE ROLE](#), [ALTER ROLE](#), [SET ROLE](#)

DROP RULE

Removes a rewrite rule.

Synopsis

```
DROP RULE [IF EXISTS] name ON relation [CASCADE | RESTRICT]
```

Description

`DROP RULE` drops a rewrite rule from a table or view.

Parameters

IF EXISTS

Do not throw an error if the rule does not exist. A notice is issued in this case.

name

The name of the rule to remove.

relation

The name (optionally schema-qualified) of the table or view that the rule applies to.

CASCADE

Automatically drop objects that depend on the rule.

RESTRICT

Refuse to drop the rule if any objects depend on it. This is the default.

Examples

Remove the rewrite rule `sales_2006` on the table `sales`:

```
DROP RULE sales_2006 ON sales;
```

Compatibility

There is no `DROP RULE` statement in the SQL standard.

See Also

[CREATE RULE](#)

DROP SCHEMA

Removes a schema.

Synopsis

```
DROP SCHEMA [IF EXISTS] name [, ...] [CASCADE | RESTRICT]
```

Description

`DROP SCHEMA` removes schemas from the database. A schema can only be dropped by its owner or a superuser. Note that the owner can drop the schema (and thereby all contained objects) even if he does not own some of the objects within the schema.

Parameters

IF EXISTS

Do not throw an error if the schema does not exist. A notice is issued in this case.

name

The name of the schema to remove.

CASCADE

Automatically drops any objects contained in the schema (tables, functions, etc.).

RESTRICT

Refuse to drop the schema if it contains any objects. This is the default.

Examples

Remove the schema *mystuff* from the database, along with everything it contains:

```
DROP SCHEMA mystuff CASCADE;
```

Compatibility

`DROP SCHEMA` is fully conforming with the SQL standard, except that the standard only allows one schema to be dropped per command. Also, the `IF EXISTS` option is a Greenplum Database extension.

See Also

[CREATE SCHEMA](#), [ALTER SCHEMA](#)

DROP SEQUENCE

Removes a sequence.

Synopsis

```
DROP SEQUENCE [IF EXISTS] name [, ...] [CASCADE | RESTRICT]
```

Description

`DROP SEQUENCE` removes a sequence generator table. You must own the sequence to drop it (or be a superuser).

Parameters

IF EXISTS

Do not throw an error if the sequence does not exist. A notice is issued in this case.

name

The name (optionally schema-qualified) of the sequence to remove.

CASCADE

Automatically drop objects that depend on the sequence.

RESTRICT

Refuse to drop the sequence if any objects depend on it. This is the default.

Examples

Remove the sequence *myserial*:

```
DROP SEQUENCE myserial;
```

Compatibility

`DROP SEQUENCE` is fully conforming with the SQL standard, except that the standard only allows one sequence to be dropped per command. Also, the `IF EXISTS` option is a Greenplum Database extension.

See Also

[ALTER SEQUENCE](#), [CREATE SEQUENCE](#)

DROP TABLE

Removes a table.

Synopsis

```
DROP TABLE [IF EXISTS] name [, ...] [CASCADE | RESTRICT]
```

Description

`DROP TABLE` removes tables from the database. Only its owner may drop a table. To empty a table of rows without removing the table definition, use `DELETE` or `TRUNCATE`.

`DROP TABLE` always removes any indexes, rules, triggers, and constraints that exist for the target table. However, to drop a table that is referenced by a view, `CASCADE` must be specified. `CASCADE` will remove a dependent view entirely.

Parameters

IF EXISTS

Do not throw an error if the table does not exist. A notice is issued in this case.

name

The name (optionally schema-qualified) of the table to remove.

CASCADE

Automatically drop objects that depend on the table (such as views).

RESTRICT

Refuse to drop the table if any objects depend on it. This is the default.

Examples

Remove the table *mytable*:

```
DROP TABLE mytable;
```

Compatibility

`DROP TABLE` is fully conforming with the SQL standard, except that the standard only allows one table to be dropped per command. Also, the `IF EXISTS` option is a Greenplum Database extension.

See Also

[CREATE TABLE](#), [ALTER TABLE](#), [TRUNCATE](#)

DROP TABLESPACE

Removes a tablespace.

Synopsis

```
DROP TABLESPACE [IF EXISTS] tablespacename
```

Description

DROP TABLESPACE removes a tablespace from the system.

A tablespace can only be dropped by its owner or a superuser. The tablespace must be empty of all database objects before it can be dropped. It is possible that objects in other databases may still reside in the tablespace even if no objects in the current database are using the tablespace.

Parameters

IF EXISTS

Do not throw an error if the tablespace does not exist. A notice is issued in this case.

tablespacename

The name of the tablespace to remove.

Examples

Remove the tablespace *mystuff*:

```
DROP TABLESPACE mystuff;
```

Compatibility

DROP TABLESPACE is a Greenplum Database extension.

See Also

[CREATE TABLESPACE](#), [ALTER TABLESPACE](#)

DROP TRIGGER

Removes a trigger.

Synopsis

```
DROP TRIGGER [IF EXISTS] name ON table [CASCADE | RESTRICT]
```

Description

`DROP TRIGGER` will remove an existing trigger definition. To execute this command, the current user must be the owner of the table for which the trigger is defined.

Parameters

IF EXISTS

Do not throw an error if the trigger does not exist. A notice is issued in this case.

name

The name of the trigger to remove.

table

The name (optionally schema-qualified) of the table for which the trigger is defined.

CASCADE

Automatically drop objects that depend on the trigger.

RESTRICT

Refuse to drop the trigger if any objects depend on it. This is the default.

Examples

Remove the trigger *sendmail* on table *expenses*;

```
DROP TRIGGER sendmail ON expenses;
```

Compatibility

The `DROP TRIGGER` statement in Greenplum Database is not compatible with the SQL standard. In the SQL standard, trigger names are not local to tables, so the command is simply `DROP TRIGGER name`.

See Also

[ALTER TRIGGER](#), [CREATE TRIGGER](#)

DROP TYPE

Removes a data type.

Synopsis

```
DROP TYPE [IF EXISTS] name [, ...] [CASCADE | RESTRICT]
```

Description

`DROP TYPE` will remove a user-defined data type. Only the owner of a type can remove it.

Parameters

IF EXISTS

Do not throw an error if the type does not exist. A notice is issued in this case.

name

The name (optionally schema-qualified) of the data type to remove.

CASCADE

Automatically drop objects that depend on the type (such as table columns, functions, operators).

RESTRICT

Refuse to drop the type if any objects depend on it. This is the default.

Examples

Remove the data type `box`;

```
DROP TYPE box;
```

Compatibility

This command is similar to the corresponding command in the SQL standard, apart from the `IF EXISTS` option, which is a Greenplum Database extension. But note that the `CREATE TYPE` command and the data type extension mechanisms in Greenplum Database differ from the SQL standard.

See Also

[ALTER TYPE](#), [CREATE TYPE](#)

DROP USER

Removes a database role.

Synopsis

```
DROP USER [IF EXISTS] name [, ...]
```

Description

`DROP USER` is an obsolete command, though still accepted for backwards compatibility. Groups (and users) have been superseded by the more general concept of roles. See [DROP ROLE](#) for more information.

Parameters

IF EXISTS

Do not throw an error if the role does not exist. A notice is issued in this case.

name

The name of an existing role.

Compatibility

There is no `DROP USER` statement in the SQL standard. The SQL standard leaves the definition of users to the implementation.

See Also

[DROP ROLE](#)

DROP VIEW

Removes a view.

Synopsis

```
DROP VIEW [IF EXISTS] name [, ...] [CASCADE | RESTRICT]
```

Description

`DROP VIEW` will remove an existing view. Only the owner of a view can remove it.

Parameters

IF EXISTS

Do not throw an error if the view does not exist. A notice is issued in this case.

name

The name (optionally schema-qualified) of the view to remove.

CASCADE

Automatically drop objects that depend on the view (such as other views).

RESTRICT

Refuse to drop the view if any objects depend on it. This is the default.

Examples

Remove the view *topten*;

```
DROP VIEW topten;
```

Compatibility

`DROP VIEW` is fully conforming with the SQL standard, except that the standard only allows one view to be dropped per command. Also, the `IF EXISTS` option is a Greenplum Database extension.

See Also

[CREATE VIEW](#)

END

Commits the current transaction.

Synopsis

```
END [WORK | TRANSACTION]
```

Description

`END` commits the current transaction. All changes made by the transaction become visible to others and are guaranteed to be durable if a crash occurs. This command is a Greenplum Database extension that is equivalent to `COMMIT`.

Parameters

WORK
TRANSACTION

Optional keywords. They have no effect.

Examples

Commit the current transaction:

```
END;
```

Compatibility

`END` is a Greenplum Database extension that provides functionality equivalent to `COMMIT`, which is specified in the SQL standard.

See Also

`BEGIN`, `ROLLBACK`, `COMMIT`

EXECUTE

Executes a prepared SQL statement.

Synopsis

```
EXECUTE name [ (parameter [, ...] ) ]
```

Description

`EXECUTE` is used to execute a previously prepared statement. Since prepared statements only exist for the duration of a session, the prepared statement must have been created by a `PREPARE` statement executed earlier in the current session.

If the `PREPARE` statement that created the statement specified some parameters, a compatible set of parameters must be passed to the `EXECUTE` statement, or else an error is raised. Note that (unlike functions) prepared statements are not overloaded based on the type or number of their parameters; the name of a prepared statement must be unique within a database session.

For more information on the creation and usage of prepared statements, see `PREPARE`.

Parameters

name

The name of the prepared statement to execute.

parameter

The actual value of a parameter to the prepared statement. This must be an expression yielding a value that is compatible with the data type of this parameter, as was determined when the prepared statement was created.

Examples

Create a prepared statement for an `INSERT` statement, and then execute it:

```
PREPARE fooplan (int, text, bool, numeric) AS INSERT INTO
foo VALUES($1, $2, $3, $4);
EXECUTE fooplan(1, 'Hunter Valley', 't', 200.00);
```

Compatibility

The SQL standard includes an `EXECUTE` statement, but it is only for use in embedded SQL. This version of the `EXECUTE` statement also uses a somewhat different syntax.

See Also

[DEALLOCATE](#), [PREPARE](#)

EXPLAIN

Shows the query plan of a statement.

Synopsis

```
EXPLAIN [ANALYZE] [VERBOSE] statement
```

Description

`EXPLAIN` displays the query plan that the Greenplum planner generates for the supplied statement. Query plans are a tree plan of nodes. Each node in the plan represents a single operation, such as table scan, join, aggregation or a sort.

Plans should be read from the bottom up as each node feeds rows into the node directly above it. The bottom nodes of a plan are usually table scan operations (sequential, index or bitmap index scans). If the query requires joins, aggregations, or sorts (or other operations on the raw rows) then there will be additional nodes above the scan nodes to perform these operations. The topmost plan nodes are usually the Greenplum Database motion nodes (redistribute, broadcast, or gather motions). These are the operations responsible for moving rows between the segment instances during query processing.

The output of `EXPLAIN` has one line for each node in the plan tree, showing the basic node type plus the following cost estimates that the planner made for the execution of that plan node:

- **cost** - measured in units of disk page fetches; that is, 1.0 equals one sequential disk page read. The first estimate is the start-up cost (cost of getting to the first row) and the second is the total cost (cost of getting all rows). Note that the total cost assumes that all rows will be retrieved, which may not always be the case (if using `LIMIT` for example).
- **rows** - the total number of rows output by this plan node. This is usually less than the actual number of rows processed or scanned by the plan node, reflecting the estimated selectivity of any `WHERE` clause conditions. Ideally the top-level nodes estimate will approximate the number of rows actually returned, updated, or deleted by the query.
- **width** - total bytes of all the rows output by this plan node.

It is important to note that the cost of an upper-level node includes the cost of all its child nodes. The topmost node of the plan has the estimated total execution cost for the plan. This is this number that the planner seeks to minimize. It is also important to realize that the cost only reflects things that the query planner cares about. In particular, the cost does not consider the time spent transmitting result rows to the client.

`EXPLAIN ANALYZE` causes the statement to be actually executed, not only planned. The `EXPLAIN ANALYZE` plan shows the actual results along with the planner's estimates. This is useful for seeing whether the planner's estimates are close to reality. In addition to the information shown in the `EXPLAIN` plan, `EXPLAIN ANALYZE` will show the following additional information:

- The total elapsed time (in milliseconds) that it took to run the query.
- The number of *workers* (segments) involved in a plan node operation. Only segments that return rows are counted.
- The maximum number of rows returned by the segment that produced the most rows for an operation. If multiple segments produce an equal number of rows, the one with the longest *time to end* is the one chosen.
- The segment id number of the segment that produced the most rows for an operation.
- For relevant operations, the *work_mem* used by the operation. If *work_mem* was not sufficient to perform the operation in memory, the plan will show how much data was spilled to disk and how many passes over the data were required for the lowest performing segment. For example:


```
Work_mem used: 64K bytes avg, 64K bytes max (seg0).
Work_mem wanted: 90K bytes avg, 90K bytes max (seg0) to abate workfile
I/O affecting 2 workers.
[seg0] pass 0: 488 groups made from 488 rows; 263 rows written to
workfile
[seg0] pass 1: 263 groups made from 263 rows
```
- The time (in milliseconds) it took to retrieve the first row from the segment that produced the most rows, and the total time taken to retrieve all rows from that segment. The *<time> to first row* may be omitted if it is the same as the *<time> to end*.

Important: Keep in mind that the statement is actually executed when `EXPLAIN ANALYZE` is used. Although `EXPLAIN ANALYZE` will discard any output that a `SELECT` would return, other side effects of the statement will happen as usual. If you wish to use `EXPLAIN ANALYZE` on a DML statement without letting the command affect your data, use this approach:

```
BEGIN;
EXPLAIN ANALYZE ...;
ROLLBACK;
```

Parameters

name

The name of the prepared statement to execute.

parameter

The actual value of a parameter to the prepared statement. This must be an expression yielding a value that is compatible with the data type of this parameter, as was determined when the prepared statement was created.

Notes

In order to allow the query planner to make reasonably informed decisions when optimizing queries, the `ANALYZE` statement should be run to record statistics about the distribution of data within the table. If you have not done this (or if the statistical

distribution of the data in the table has changed significantly since the last time `ANALYZE` was run), the estimated costs are unlikely to conform to the real properties of the query, and consequently an inferior query plan may be chosen.

See also, “[Query Profiling](#)” on page 148.

Examples

To illustrate how to read an `EXPLAIN` query plan, consider the following example for a very simple query:

```
EXPLAIN SELECT * FROM names WHERE name = 'Joelle';
          QUERY PLAN
-----
Gather Motion 2:1 (slice1) (cost=0.00..20.88 rows=1 width=13)
  -> Seq Scan on 'names' (cost=0.00..20.88 rows=1 width=13)
      Filter: name::text ~~ 'Joelle'::text
```

If we read the plan from the bottom up, the query planner starts by doing a sequential scan of the `names` table. Notice that the `WHERE` clause is being applied as a *filter* condition. This means that the scan operation checks the condition for each row it scans, and outputs only the ones that pass the condition.

The results of the scan operation are passed up to a *gather motion* operation. In Greenplum Database, a gather motion is when segments send rows up to the master. In this case we have 2 segment instances sending to 1 master instance (2:1). This operation is working on *slice1* of the parallel query execution plan. In Greenplum Database a query plan is divided into *slices* so that portions of the query plan can be worked on in parallel by the segments.

The estimated startup cost for this plan is 00.00 (no cost) and a total cost of 20.88 disk page fetches. The planner is estimating that this query will return one row.

Compatibility

There is no `EXPLAIN` statement defined in the SQL standard.

See Also

[ANALYZE](#)

FETCH

Retrieves rows from a query using a cursor.

Synopsis

```
FETCH [ forward_direction { FROM | IN } ] cursorname
```

where *forward_direction* can be empty or one of:

```
NEXT
FIRST
LAST
ABSOLUTE count
RELATIVE count
count
ALL
FORWARD
FORWARD count
FORWARD ALL
```

Description

FETCH retrieves rows using a previously-created cursor.

A cursor has an associated position, which is used by FETCH. The cursor position can be before the first row of the query result, on any particular row of the result, or after the last row of the result. When created, a cursor is positioned before the first row. After fetching some rows, the cursor is positioned on the row most recently retrieved. If FETCH runs off the end of the available rows then the cursor is left positioned after the last row. FETCH ALL will always leave the cursor positioned after the last row.

The forms NEXT, FIRST, LAST, ABSOLUTE, RELATIVE fetch a single row after moving the cursor appropriately. If there is no such row, an empty result is returned, and the cursor is left positioned before the first row or after the last row as appropriate.

The forms using FORWARD retrieve the indicated number of rows moving in the forward direction, leaving the cursor positioned on the last-returned row (or after all rows, if the count exceeds the number of rows available). Note that it is not possible to move a cursor position backwards in Greenplum Database, since scrollable cursors are not supported. You can only move a cursor forward in position using FETCH.

RELATIVE 0 and FORWARD 0 request fetching the current row without moving the cursor, that is, re-fetching the most recently fetched row. This will succeed unless the cursor is positioned before the first row or after the last row, in which case no row is returned.

Outputs

On successful completion, a FETCH command returns a command tag of the form

```
FETCH count
```

The count is the number of rows fetched (possibly zero). Note that in `psql`, the command tag will not actually be displayed, since `psql` displays the fetched rows instead.

Parameters

forward_direction

Defines the fetch direction and number of rows to fetch. Only forward fetches are allowed in Greenplum Database. It can be one of the following:

NEXT

Fetch the next row. This is the default if direction is omitted.

FIRST

Fetch the first row of the query (same as `ABSOLUTE 1`). Only allowed if it is the first `FETCH` operation using this cursor.

LAST

Fetch the last row of the query (same as `ABSOLUTE -1`).

ABSOLUTE count

Fetch the specified row of the query. Position after last row if count is out of range. Only allowed if the row specified by *count* moves the cursor position forward.

RELATIVE count

Fetch the specified row of the query *count* rows ahead of the current cursor position. `RELATIVE 0` re-fetches the current row, if any. Only allowed if *count* moves the cursor position forward.

count

Fetch the next *count* number of rows (same as `FORWARD count`).

ALL

Fetch all remaining rows (same as `FORWARD ALL`).

FORWARD

Fetch the next row (same as `NEXT`).

FORWARD count

Fetch the next *count* number of rows. `FORWARD 0` re-fetches the current row.

FORWARD ALL

Fetch all remaining rows.

cursorname

The name of an open cursor.

Notes

Greenplum Database does not support scrollable cursors, so you can only use `FETCH` to move the cursor position forward.

`ABSOLUTE` fetches are not any faster than navigating to the desired row with a relative move: the underlying implementation must traverse all the intermediate rows anyway.

Updating data via a cursor is currently not supported by Greenplum Database.

`DECLARE` is used to define a cursor. Use `MOVE` to change cursor position without retrieving data.

Examples

-- Start the transaction:

```
BEGIN;
```

-- Set up a cursor:

```
DECLARE mycursor CURSOR FOR SELECT * FROM films;
```

-- Fetch the first 5 rows in the cursor *mycursor*:

```
FETCH FORWARD 5 FROM mycursor;
```

code	title	did	date_prod	kind	len
BL101	The Third Man	101	1949-12-23	Drama	01:44
BL102	The African Queen	101	1951-08-11	Romantic	01:43
JL201	Une Femme est une Femme	102	1961-03-12	Romantic	01:25
P_301	Vertigo	103	1958-11-14	Action	02:08
P_302	Becket	103	1964-02-03	Drama	02:28

-- Close the cursor and end the transaction:

```
CLOSE mycursor;
COMMIT;
```

Compatibility

SQL standard allows cursors only in embedded SQL and in modules. Greenplum Database permits cursors to be used interactively.

The variant of `FETCH` described here returns the data as if it were a `SELECT` result rather than placing it in host variables. Other than this point, `FETCH` is fully upward-compatible with the SQL standard.

The `FETCH` forms involving `FORWARD`, as well as the forms `FETCH count` and `FETCH ALL`, in which `FORWARD` is implicit, are Greenplum Database extensions. `BACKWARD` is not supported.

The SQL standard allows only `FROM` preceding the cursor name; the option to use `IN` is an extension.

See Also

[DECLARE](#), [CLOSE](#), [MOVE](#)

GRANT

Defines access privileges.

Synopsis

```
GRANT { {SELECT | INSERT | UPDATE | DELETE | REFERENCES |
TRIGGER} [,...] | ALL [PRIVILEGES] }
    ON [TABLE] tablename [, ...]
    TO {rolename | PUBLIC} [, ...] [WITH GRANT OPTION]

GRANT { {USAGE | SELECT | UPDATE} [,...] | ALL [PRIVILEGES] }
    ON SEQUENCE sequencename [, ...]
    TO { rolename | PUBLIC } [, ...] [WITH GRANT OPTION]

GRANT { {CREATE | CONNECT | TEMPORARY | TEMP} [,...] | ALL
[PRIVILEGES] }
    ON DATABASE dbname [, ...]
    TO {rolename | PUBLIC} [, ...] [WITH GRANT OPTION]

GRANT { EXECUTE | ALL [PRIVILEGES] }
    ON FUNCTION funcname ( [ [argmode] [argname] argtype [, ...]
] ) [, ...]
    TO {rolename | PUBLIC} [, ...] [WITH GRANT OPTION]

GRANT { USAGE | ALL [PRIVILEGES] }
    ON LANGUAGE langname [, ...]
    TO {rolename | PUBLIC} [, ...] [WITH GRANT OPTION]

GRANT { {CREATE | USAGE} [,...] | ALL [PRIVILEGES] }
    ON SCHEMA schemaname [, ...]
    TO {rolename | PUBLIC} [, ...] [WITH GRANT OPTION]

GRANT { CREATE | ALL [PRIVILEGES] }
    ON TABLESPACE tablespacename [, ...]
    TO {rolename | PUBLIC} [, ...] [WITH GRANT OPTION]

GRANT parent_role [, ...]
    TO member_role [, ...] [WITH ADMIN OPTION]
```

Description

The GRANT command has two basic variants: one that grants privileges on a database object (table, view, sequence, database, function, procedural language, schema, or tablespace), and one that grants membership in a role.

GRANT on Database Objects

This variant of the GRANT command gives specific privileges on a database object to one or more roles. These privileges are added to those already granted, if any.

The key word `PUBLIC` indicates that the privileges are to be granted to all roles, including those that may be created later. `PUBLIC` may be thought of as an implicitly defined group-level role that always includes all roles. Any particular role will have the sum of privileges granted directly to it, privileges granted to any role it is presently a member of, and privileges granted to `PUBLIC`.

If `WITH GRANT OPTION` is specified, the recipient of the privilege may in turn grant it to others. Without a grant option, the recipient cannot do that. Grant options cannot be granted to `PUBLIC`.

There is no need to grant privileges to the owner of an object (usually the role that created it), as the owner has all privileges by default. The right to drop an object, or to alter its definition in any way is not described by a grantable privilege; it is inherent in the owner, and cannot be granted or revoked. The owner implicitly has all grant options for the object, too.

Depending on the type of object, the initial default privileges may include granting some privileges to `PUBLIC`. The default is no public access for tables, schemas, and tablespaces; `CONNECT` privilege and `TEMP` table creation privilege for databases; `EXECUTE` privilege for functions; and `USAGE` privilege for languages. The object owner may of course revoke these privileges.

Grant on Roles

This variant of the `GRANT` command grants membership in a role to one or more other roles. Membership in a role is significant because it conveys the privileges granted to a role to each of its members.

If `WITH ADMIN OPTION` is specified, the member may in turn grant membership in the role to others, and revoke membership in the role as well. Database superusers can grant or revoke membership in any role to anyone. Roles having `CREATEROLE` privilege can grant or revoke membership in any role that is not a superuser.

Unlike the case with privileges, membership in a role cannot be granted to `PUBLIC`.

Parameters

SELECT

Allows `SELECT` from any column of the specified table, view, or sequence. Also allows the use of `COPY TO`. For sequences, this privilege also allows the use of the `currval` function.

INSERT

Allows `INSERT` of a new row into the specified table. Also allows `COPY FROM`.

UPDATE

Allows `UPDATE` of any column of the specified table. `SELECT ... FOR UPDATE` and `SELECT ... FOR SHARE` also require this privilege (as well as the `SELECT` privilege). For sequences, this privilege allows the use of the `nextval` and `setval` functions.

DELETE

Allows `DELETE` of a row from the specified table.

REFERENCES

This keyword is accepted, although foreign key constraints are currently not supported in Greenplum Database. To create a foreign key constraint, it is necessary to have this privilege on both the referencing and referenced tables.

TRIGGER

Allows the creation of a trigger on the specified table.

CREATE

For databases, allows new schemas to be created within the database.

For schemas, allows new objects to be created within the schema. To rename an existing object, you must own the object and have this privilege for the containing schema.

For tablespaces, allows tables and indexes to be created within the tablespace, and allows databases to be created that have the tablespace as their default tablespace. (Note that revoking this privilege will not alter the placement of existing objects.)

CONNECT

Allows the user to connect to the specified database. This privilege is checked at connection startup (in addition to checking any restrictions imposed by `pg_hba.conf`).

**TEMPORARY
TEMP**

Allows temporary tables to be created while using the database.

EXECUTE

Allows the use of the specified function and the use of any operators that are implemented on top of the function. This is the only type of privilege that is applicable to functions. (This syntax works for aggregate functions, as well.)

USAGE

For procedural languages, allows the use of the specified language for the creation of functions in that language. This is the only type of privilege that is applicable to procedural languages.

For schemas, allows access to objects contained in the specified schema (assuming that the objects' own privilege requirements are also met). Essentially this allows the grantee to look up objects within the schema.

For sequences, this privilege allows the use of the `currval` and `nextval` functions.

ALL PRIVILEGES

Grant all of the available privileges at once. The `PRIVILEGES` key word is optional in Greenplum Database, though it is required by strict SQL.

PUBLIC

A special group-level role that denotes that the privileges are to be granted to all roles, including those that may be created later.

WITH GRANT OPTION

The recipient of the privilege may in turn grant it to others.

WITH ADMIN OPTION

The member of a role may in turn grant membership in the role to others.

Notes

Database superusers can access all objects regardless of object privilege settings. One exception to this rule is view objects. Access to tables referenced in the view is determined by permissions of the view owner not the current user (even if the current user is a superuser).

If a superuser chooses to issue a `GRANT` or `REVOKE` command, the command is performed as though it were issued by the owner of the affected object. In particular, privileges granted via such a command will appear to have been granted by the object owner. For role membership, the membership appears to have been granted by the containing role itself.

`GRANT` and `REVOKE` can also be done by a role that is not the owner of the affected object, but is a member of the role that owns the object, or is a member of a role that holds privileges `WITH GRANT OPTION` on the object. In this case the privileges will be recorded as having been granted by the role that actually owns the object or holds the privileges `WITH GRANT OPTION`.

Granting permission on a table does not automatically extend permissions to any sequences used by the table, including sequences tied to `SERIAL` columns. Permissions on a sequence must be set separately.

Greenplum Database does not support granting or revoking privileges for individual columns of a table. One possible workaround is to create a view having just the desired columns and then grant privileges to that view.

Use `psql`'s `\z` meta-command to obtain information about existing privileges for an object.

Examples

Grant insert privilege to all roles on table *mytable*:

```
GRANT INSERT ON mytable TO PUBLIC;
```

Grant all available privileges to role *sally* on the view *topten*. Note that while the above will indeed grant all privileges if executed by a superuser or the owner of *topten*, when executed by someone else it will only grant those permissions for which the granting role has grant options.

```
GRANT ALL PRIVILEGES ON topten TO sally;
```

Grant membership in role *admins* to user *joe*:

```
GRANT admins TO joe;
```

Compatibility

The `PRIVILEGES` key word in is required in the SQL standard, but optional in Greenplum Database. The SQL standard does not support setting the privileges on more than one object per command.

Greenplum Database allows an object owner to revoke his own ordinary privileges: for example, a table owner can make the table read-only to himself by revoking his own `INSERT`, `UPDATE`, and `DELETE` privileges. This is not possible according to the SQL standard. Greenplum Database treats the owner's privileges as having been granted by the owner to himself; therefore he can revoke them too. In the SQL standard, the owner's privileges are granted by an assumed *system* entity.

The SQL standard allows setting privileges for individual columns within a table.

The SQL standard provides for a `USAGE` privilege on other kinds of objects: character sets, collations, translations, domains.

Privileges on databases, tablespaces, schemas, and languages are Greenplum Database extensions.

See Also

[REVOKE](#)

INSERT

Creates new rows in a table.

Synopsis

```
INSERT INTO table [( column [, ...] )]
    {DEFAULT VALUES | VALUES ( {expression | DEFAULT} [, ...] )
    [, ...] | query}
```

Description

INSERT inserts new rows into a table. One can insert one or more rows specified by value expressions, or zero or more rows resulting from a query.

The target column names may be listed in any order. If no list of column names is given at all, the default is the columns of the table in their declared order. The values supplied by the VALUES clause or query are associated with the explicit or implicit column list left-to-right.

Each column not present in the explicit or implicit column list will be filled with a default value, either its declared default value or null if there is no default.

If the expression for any column is not of the correct data type, automatic type conversion will be attempted.

You must have INSERT privilege on a table in order to insert into it.

Outputs

On successful completion, an INSERT command returns a command tag of the form:

```
INSERT oid count
```

The *count* is the number of rows inserted. If count is exactly one, and the target table has OIDs, then *oid* is the OID assigned to the inserted row. Otherwise *oid* is zero.

Parameters

table

The name (optionally schema-qualified) of an existing table.

column

The name of a column in table. The column name can be qualified with a subfield name or array subscript, if needed. (Inserting into only some fields of a composite column leaves the other fields null.)

DEFAULT VALUES

All columns will be filled with their default values.

expression

An expression or value to assign to the corresponding column.

DEFAULT

The corresponding column will be filled with its default value.

query

A query (`SELECT` statement) that supplies the rows to be inserted. Refer to the `SELECT` statement for a description of the syntax.

Examples

Insert a single row into table *films*:

```
INSERT INTO films VALUES ('UA502', 'Bananas', 105,
    '1971-07-13', 'Comedy', '82 minutes');
```

In this example, the *length* column is omitted and therefore it will have the default value:

```
INSERT INTO films (code, title, did, date_prod, kind) VALUES
    ('T_601', 'Yojimbo', 106, '1961-06-16', 'Drama');
```

This example uses the `DEFAULT` clause for the *date_prod* column rather than specifying a value:

```
INSERT INTO films VALUES ('UA502', 'Bananas', 105, DEFAULT,
    'Comedy', '82 minutes');
```

To insert a row consisting entirely of default values:

```
INSERT INTO films DEFAULT VALUES;
```

To insert multiple rows using the multirow `VALUES` syntax:

```
INSERT INTO films (code, title, did, date_prod, kind) VALUES
    ('B6717', 'Tampopo', 110, '1985-02-10', 'Comedy'),
    ('HG120', 'The Dinner Game', 140, DEFAULT, 'Comedy');
```

This example inserts some rows into table *films* from a table *tmp_films* with the same column layout as *films*:

```
INSERT INTO films SELECT * FROM tmp_films WHERE date_prod <
    '2004-05-07';
```

Compatibility

`INSERT` conforms to the SQL standard. The case in which a column name list is omitted, but not all the columns are filled from the `VALUES` clause or query, is disallowed by the standard.

Possible limitations of the *query* clause are documented under `SELECT`.

See Also

`COPY`, `SELECT`, `CREATE EXTERNAL TABLE`

LOAD

Loads or reloads a shared library file.

Synopsis

```
LOAD 'filename'
```

Description

This command loads a shared library file into the Greenplum Database server address space. If the file had been loaded previously, it is first unloaded. This command is primarily useful to unload and reload a shared library file that has been changed since the server first loaded it. To make use of the shared library, function(s) in it need to be declared using the `CREATE FUNCTION` command.

The file name is specified in the same way as for shared library names in `CREATE FUNCTION`; in particular, one may rely on a search path and automatic addition of the system's standard shared library file name extension.

Note that in Greenplum Database the shared library file (`.so` file) must reside in the same path location on every host in the Greenplum Database array (masters, segments, and mirrors).

Only database superusers can load shared library files.

Parameters

filename

The path and file name of a shared library file. This file must exist in the same location on all hosts in your Greenplum Database array.

Examples

Load a shared library file:

```
LOAD '/usr/local/greenplum-db/lib/myfuncs.so';
```

Compatibility

`LOAD` is a Greenplum Database extension.

See Also

[CREATE FUNCTION](#)

LOCK

Locks a table.

Synopsis

```
LOCK [TABLE] name [, ...] [IN lockmode MODE] [NOWAIT]
```

where *lockmode* is one of:

```
ACCESS SHARE | ROW SHARE | ROW EXCLUSIVE | SHARE UPDATE  
EXCLUSIVE | SHARE | SHARE ROW EXCLUSIVE | EXCLUSIVE | ACCESS  
EXCLUSIVE
```

Description

`LOCK TABLE` obtains a table-level lock, waiting if necessary for any conflicting locks to be released. If `NOWAIT` is specified, `LOCK TABLE` does not wait to acquire the desired lock: if it cannot be acquired immediately, the command is aborted and an error is emitted. Once obtained, the lock is held for the remainder of the current transaction. There is no `UNLOCK TABLE` command; locks are always released at transaction end.

When acquiring locks automatically for commands that reference tables, Greenplum Database always uses the least restrictive lock mode possible. `LOCK TABLE` provides for cases when you might need more restrictive locking. For example, suppose an application runs a transaction at the *Read Committed* isolation level and needs to ensure that data in a table remains stable for the duration of the transaction. To achieve this you could obtain `SHARE` lock mode over the table before querying. This will prevent concurrent data changes and ensure subsequent reads of the table see a stable view of committed data, because `SHARE` lock mode conflicts with the `ROW EXCLUSIVE` lock acquired by writers, and your `LOCK TABLE name IN SHARE MODE` statement will wait until any concurrent holders of `ROW EXCLUSIVE` mode locks commit or roll back. Thus, once you obtain the lock, there are no uncommitted writes outstanding; furthermore none can begin until you release the lock.

To achieve a similar effect when running a transaction at the *Serializable* isolation level, you have to execute the `LOCK TABLE` statement before executing any `SELECT` or data modification statement. A serializable transaction's view of data will be frozen when its first `SELECT` or data modification statement begins. A `LOCK TABLE` later in the transaction will still prevent concurrent writes — but it won't ensure that what the transaction reads corresponds to the latest committed values.

If a transaction of this sort is going to change the data in the table, then it should use `SHARE ROW EXCLUSIVE` lock mode instead of `SHARE` mode. This ensures that only one transaction of this type runs at a time. Without this, a deadlock is possible: two transactions might both acquire `SHARE` mode, and then be unable to also acquire `ROW EXCLUSIVE` mode to actually perform their updates. Note that a transaction's own locks never conflict, so a transaction can acquire `ROW EXCLUSIVE` mode when it holds `SHARE` mode — but not if anyone else holds `SHARE` mode. To avoid deadlocks, make

sure all transactions acquire locks on the same objects in the same order, and if multiple lock modes are involved for a single object, then transactions should always acquire the most restrictive mode first.

Parameters

name

The name (optionally schema-qualified) of an existing table to lock.

If multiple tables are given, tables are locked one-by-one in the order specified in the `LOCK TABLE` command.

lockmode

The lock mode specifies which locks this lock conflicts with. If no lock mode is specified, then `ACCESS EXCLUSIVE`, the most restrictive mode, is used. Lock modes are as follows:

- **ACCESS SHARE** — Conflicts with the **ACCESS EXCLUSIVE** lock mode only. The commands `SELECT` and `ANALYZE` automatically acquire a lock of this mode on referenced tables. In general, any query that only reads a table and does not modify it will acquire this lock mode.
- **ROW SHARE** — Conflicts with the `EXCLUSIVE` and `ACCESS EXCLUSIVE` lock modes. The `SELECT FOR UPDATE` and `SELECT FOR SHARE` commands automatically acquire a lock of this mode on the target table(s) (in addition to `ACCESS SHARE` locks on any other tables that are referenced but not selected `FOR UPDATE/FOR SHARE`).
- **ROW EXCLUSIVE** — Conflicts with the `SHARE`, `SHARE ROW EXCLUSIVE`, `EXCLUSIVE`, and `ACCESS EXCLUSIVE` lock modes. The commands `INSERT` and `COPY` automatically acquire this lock mode on the target table (in addition to `ACCESS SHARE` locks on any other referenced tables).
- **SHARE UPDATE EXCLUSIVE** — Conflicts with the `SHARE UPDATE EXCLUSIVE`, `SHARE`, `SHARE ROW EXCLUSIVE`, `EXCLUSIVE`, and `ACCESS EXCLUSIVE` lock modes. This mode protects a table against concurrent schema changes and `VACUUM` runs. Acquired automatically by `VACUUM` (without `FULL`).
- **SHARE** — Conflicts with the `ROW EXCLUSIVE`, `SHARE UPDATE EXCLUSIVE`, `SHARE ROW EXCLUSIVE`, `EXCLUSIVE`, and `ACCESS EXCLUSIVE` lock modes. This mode protects a table against concurrent data changes. Acquired automatically by `CREATE INDEX`.
- **SHARE ROW EXCLUSIVE** — Conflicts with the `ROW EXCLUSIVE`, `SHARE UPDATE EXCLUSIVE`, `SHARE`, `SHARE ROW EXCLUSIVE`, `EXCLUSIVE`, and `ACCESS EXCLUSIVE` lock modes. This lock mode is not automatically acquired by any Greenplum Database command.

- **EXCLUSIVE** — Conflicts with the `ROW SHARE`, `ROW EXCLUSIVE`, `SHARE UPDATE EXCLUSIVE`, `SHARE`, `SHARE ROW EXCLUSIVE`, `EXCLUSIVE`, and `ACCESS EXCLUSIVE` lock modes. This mode allows only concurrent `ACCESS SHARE` locks, i.e., only reads from the table can proceed in parallel with a transaction holding this lock mode. This lock mode is automatically acquired for `UPDATE` and `DELETE` in Greenplum Database (which is more restrictive locking than in regular PostgreSQL).
- **ACCESS EXCLUSIVE** — Conflicts with locks of all modes (`ACCESS SHARE`, `ROW SHARE`, `ROW EXCLUSIVE`, `SHARE UPDATE EXCLUSIVE`, `SHARE`, `SHARE ROW EXCLUSIVE`, `EXCLUSIVE`, and `ACCESS EXCLUSIVE`). This mode guarantees that the holder is the only transaction accessing the table in any way. Acquired automatically by the `ALTER TABLE`, `DROP TABLE`, `REINDEX`, `CLUSTER`, and `VACUUM FULL` commands. This is also the default lock mode for `LOCK TABLE` statements that do not specify a mode explicitly.

NOWAIT

Specifies that `LOCK TABLE` should not wait for any conflicting locks to be released: if the specified lock(s) cannot be acquired immediately without waiting, the transaction is aborted.

Notes

`LOCK TABLE ... IN ACCESS SHARE MODE` requires `SELECT` privileges on the target table. All other forms of `LOCK` require `UPDATE` and/or `DELETE` privileges.

`LOCK TABLE` is useful only inside a transaction block (`BEGIN/COMMIT` pair), since the lock is dropped as soon as the transaction ends. A `LOCK TABLE` command appearing outside any transaction block forms a self-contained transaction, so the lock will be dropped as soon as it is obtained.

`LOCK TABLE` only deals with table-level locks, and so the mode names involving `ROW` are all misnomers. These mode names should generally be read as indicating the intention of the user to acquire row-level locks within the locked table. Also, `ROW EXCLUSIVE` mode is a sharable table lock. Keep in mind that all the lock modes have identical semantics so far as `LOCK TABLE` is concerned, differing only in the rules about which modes conflict with which. For information on how to acquire an actual row-level lock, see the `FOR UPDATE/FOR SHARE` clause in the [SELECT](#) reference documentation.

Examples

Obtain a `SHARE` lock on the `films` table when going to perform inserts into the `films_user_comments` table:

```
BEGIN WORK;
LOCK TABLE films IN SHARE MODE;
SELECT id FROM films
    WHERE name = 'Star Wars: Episode I - The Phantom Menace';
-- Do ROLLBACK if record was not returned
INSERT INTO films_user_comments VALUES
```

```
    (_id_, 'GREAT! I was waiting for it for so long!');  
COMMIT WORK;
```

Take a `SHARE ROW EXCLUSIVE` lock on a table when performing a delete operation:

```
BEGIN WORK;  
LOCK TABLE films IN SHARE ROW EXCLUSIVE MODE;  
DELETE FROM films_user_comments WHERE id IN  
    (SELECT id FROM films WHERE rating < 5);  
DELETE FROM films WHERE rating < 5;  
COMMIT WORK;
```

Compatibility

There is no `LOCK TABLE` in the SQL standard, which instead uses `SET TRANSACTION` to specify concurrency levels on transactions. Greenplum Database supports that too.

Except for `ACCESS SHARE`, `ACCESS EXCLUSIVE`, and `SHARE UPDATE EXCLUSIVE` lock modes, the Greenplum Database lock modes and the `LOCK TABLE` syntax are compatible with those present in Oracle.

See Also

[BEGIN](#), [SET TRANSACTION](#), [SELECT](#)

MOVE

Positions a cursor.

Synopsis

```
MOVE [ forward_direction {FROM | IN} ] cursorname
```

where *direction* can be empty or one of:

```
NEXT
FIRST
LAST
ABSOLUTE count
RELATIVE count
count
ALL
FORWARD
FORWARD count
FORWARD ALL
```

Description

MOVE repositions a cursor without retrieving any data. MOVE works exactly like the [FETCH](#) command, except it only positions the cursor and does not return rows.

Note that it is not possible to move a cursor position backwards in Greenplum Database, since scrollable cursors are not supported. You can only move a cursor forward in position using MOVE.

Outputs

On successful completion, a MOVE command returns a command tag of the form

```
MOVE count
```

The count is the number of rows that a [FETCH](#) command with the same parameters would have returned (possibly zero).

Parameters

forward_direction

See [FETCH](#) for more information.

cursorname

The name of an open cursor.

Examples

-- Start the transaction:

```
BEGIN;
```

-- Set up a cursor:

```
DECLARE mycursor CURSOR FOR SELECT * FROM films;
```

-- Move forward 5 rows in the cursor *mycursor*:

```
MOVE FORWARD 5 IN mycursor;
```

```
MOVE 5
```

--Fetch the next row after that (row 6):

```
FETCH 1 FROM mycursor;
```

```
code | title | did | date_prod | kind | len
```

```
-----+-----+-----+-----+-----+-----
```

```
P_303 | 48 Hrs | 103 | 1982-10-22 | Action | 01:37
```

```
(1 row)
```

-- Close the cursor and end the transaction:

```
CLOSE mycursor;
```

```
COMMIT;
```

Compatibility

There is no `MOVE` statement in the SQL standard.

See Also

[DECLARE](#), [FETCH](#), [CLOSE](#)

PREPARE

Prepare a statement for execution.

Synopsis

```
PREPARE name [ (datatype [, ...] ) ] AS statement
```

Description

`PREPARE` creates a prepared statement. A prepared statement is a server-side object that can be used to optimize performance. When the `PREPARE` statement is executed, the specified statement is parsed, rewritten, and planned. When an `EXECUTE` command is subsequently issued, the prepared statement need only be executed. Thus, the parsing, rewriting, and planning stages are only performed once, instead of every time the statement is executed.

Prepared statements can take parameters: values that are substituted into the statement when it is executed. When creating the prepared statement, refer to parameters by position, using `$1`, `$2`, etc. A corresponding list of parameter data types can optionally be specified. When a parameter's data type is not specified or is declared as unknown, the type is inferred from the context in which the parameter is used (if possible). When executing the statement, specify the actual values for these parameters in the `EXECUTE` statement.

Prepared statements only last for the duration of the current database session. When the session ends, the prepared statement is forgotten, so it must be recreated before being used again. This also means that a single prepared statement cannot be used by multiple simultaneous database clients; however, each client can create their own prepared statement to use. The prepared statement can be manually cleaned up using the `DEALLOCATE` command.

Prepared statements have the largest performance advantage when a single session is being used to execute a large number of similar statements. The performance difference will be particularly significant if the statements are complex to plan or rewrite, for example, if the query involves a join of many tables or requires the application of several rules. If the statement is relatively simple to plan and rewrite but relatively expensive to execute, the performance advantage of prepared statements will be less noticeable.

Parameters

name

An arbitrary name given to this particular prepared statement. It must be unique within a single session and is subsequently used to execute or deallocate a previously prepared statement.

datatype

The data type of a parameter to the prepared statement. If the data type of a particular parameter is unspecified or is specified as unknown, it will be inferred from the context in which the parameter is used. To refer to the parameters in the prepared statement itself, use \$1, \$2, etc.

statement

Any SELECT, INSERT, UPDATE, DELETE, or VALUES statement.

Notes

In some situations, the query plan produced for a prepared statement will be inferior to the query plan that would have been chosen if the statement had been submitted and executed normally. This is because when the statement is planned and the planner attempts to determine the optimal query plan, the actual values of any parameters specified in the statement are unavailable. Greenplum Database collects statistics on the distribution of data in the table, and can use constant values in a statement to make guesses about the likely result of executing the statement. Since this data is unavailable when planning prepared statements with parameters, the chosen plan may be suboptimal. To examine the query plan Greenplum Database has chosen for a prepared statement, use EXPLAIN.

For more information on query planning and the statistics collected by Greenplum Database for that purpose, see the ANALYZE documentation.

You can see all available prepared statements of a session by querying the *pg_prepared_statements* system view.

Examples

Create a prepared statement for an INSERT statement, and then execute it:

```
PREPARE fooplan (int, text, bool, numeric) AS INSERT INTO
foo VALUES ($1, $2, $3, $4);
EXECUTE fooplan(1, 'Hunter Valley', 't', 200.00);
```

Create a prepared statement for a SELECT statement, and then execute it. Note that the data type of the second parameter is not specified, so it is inferred from the context in which \$2 is used:

```
PREPARE usrrptplan (int) AS SELECT * FROM users u, logs l
WHERE u.usrid=$1 AND u.usrid=l.usrid AND l.date = $2;
EXECUTE usrrptplan(1, current_date);
```

Compatibility

The SQL standard includes a PREPARE statement, but it is only for use in embedded SQL. This version of the PREPARE statement also uses a somewhat different syntax.

See Also

`EXECUTE`, `DEALLOCATE`

REASSIGN OWNED

Changes the ownership of database objects owned by a database role.

Synopsis

```
REASSIGN OWNED BY old_role [, ...] TO new_role
```

Description

REASSIGN OWNED reassigns all the objects in the current database that are owned by *old_role* to *new_role*. Note that it does not change the ownership of the database itself.

Parameters

old_role

The name of a role. The ownership of all the objects in the current database owned by this role will be reassigned to *new_role*.

new_role

The name of the role that will be made the new owner of the affected objects.

Notes

REASSIGN OWNED is often used to prepare for the removal of one or more roles. Because REASSIGN OWNED only affects the objects in the current database, it is usually necessary to execute this command in each database that contains objects owned by a role that is to be removed.

The DROP OWNED command is an alternative that drops all the database objects owned by one or more roles.

The REASSIGN OWNED command does not affect the privileges granted to the old roles in objects that are not owned by them. Use DROP OWNED to revoke those privileges.

Examples

Reassign any database objects owned by the role named *sally* and *bob* to *admin*;

```
REASSIGN OWNED BY sally, bob TO admin;
```

Compatibility

The REASSIGN OWNED statement is a Greenplum Database extension.

See Also

[DROP OWNED](#), [DROP ROLE](#)

REINDEX

Rebuilds indexes.

Synopsis

```
REINDEX {INDEX | TABLE | DATABASE | SYSTEM} name
```

Description

`REINDEX` rebuilds an index using the data stored in the index's table, replacing the old copy of the index. There are several scenarios in which to use `REINDEX`:

- An index has become corrupted, and no longer contains valid data. Although in theory this should never happen, in practice indexes may become corrupted due to software bugs or hardware failures. `REINDEX` provides a recovery method.
- An index has become bloated, that it contains many empty or nearly-empty pages. This can occur with B-tree indexes in Greenplum Database under certain uncommon access patterns. `REINDEX` provides a way to reduce the space consumption of the index by writing a new version of the index without the dead pages.
- You have altered the fillfactor storage parameter for an index, and wish to ensure that the change has taken full effect.
- An index build with the `CONCURRENTLY` option failed, leaving an invalid index. Such indexes are useless but it can be convenient to use `REINDEX` to rebuild them. Note that `REINDEX` will not perform a concurrent build. To build the index without interfering with production you should drop the index and reissue the `CREATE INDEX CONCURRENTLY` command.

Parameters

INDEX

Recreate the specified index.

TABLE

Recreate all indexes of the specified table. If the table has a secondary TOAST table, that is reindexed as well.

DATABASE

Recreate all indexes within the current database. Indexes on shared system catalogs are skipped. This form of `REINDEX` cannot be executed inside a transaction block.

SYSTEM

Recreate all indexes on system catalogs within the current database. Indexes on user tables are not processed. Also, indexes on shared (global) system catalogs are skipped. This form of `REINDEX` cannot be executed inside a transaction block.

name

The name of the specific index, table, or database to be reindexed. Index and table names may be schema-qualified. Presently, `REINDEX DATABASE` and `REINDEX SYSTEM` can only reindex the current database, so their parameter must match the current database's name.

Notes

`REINDEX` is similar to a drop and recreate of the index in that the index contents are rebuilt from scratch. However, the locking considerations are rather different. `REINDEX` locks out writes but not reads of the index's parent table. It also takes an exclusive lock on the specific index being processed, which will block reads that attempt to use that index. In contrast, `DROP INDEX` momentarily takes exclusive lock on the parent table, blocking both writes and reads. The subsequent `CREATE INDEX` locks out writes but not reads; since the index is not there, no read will attempt to use it, meaning that there will be no blocking but reads may be forced into expensive sequential scans. Another important point is that the drop/create approach invalidates any cached query plans that use the index, while `REINDEX` does not.

Reindexing a single index or table requires being the owner of that index or table. Reindexing a database requires being the owner of the database (note that the owner can therefore rebuild indexes of tables owned by other users). Of course, superusers can always reindex anything.

If you suspect that shared global system catalog indexes are corrupted, they can only be reindexed in Greenplum utility mode. Contact Greenplum Customer Support for assistance in this situation. The typical symptom of a corrupt shared index is “index is not a btree” errors, or else the server crashes immediately at startup due to reliance on the corrupted indexes.

Examples

Rebuild a single index:

```
REINDEX INDEX my_index;
```

Rebuild all the indexes on the table *my_table*:

```
REINDEX TABLE my_table;
```

Compatibility

There is no `REINDEX` command in the SQL standard.

See Also

`CREATE INDEX`, `DROP INDEX`, `VACUUM`

RELEASE SAVEPOINT

Destroys a previously defined savepoint.

Synopsis

```
RELEASE [SAVEPOINT] savepoint_name
```

Description

RELEASE SAVEPOINT destroys a savepoint previously defined in the current transaction.

Destroying a savepoint makes it unavailable as a rollback point, but it has no other user visible behavior. It does not undo the effects of commands executed after the savepoint was established. (To do that, see [ROLLBACK TO SAVEPOINT](#).) Destroying a savepoint when it is no longer needed may allow the system to reclaim some resources earlier than transaction end.

RELEASE SAVEPOINT also destroys all savepoints that were established *after* the named savepoint was established.

Parameters

savepoint_name

The name of the savepoint to destroy.

Examples

To establish and later destroy a savepoint:

```
BEGIN;  
    INSERT INTO table1 VALUES (3);  
    SAVEPOINT my_savepoint;  
    INSERT INTO table1 VALUES (4);  
    RELEASE SAVEPOINT my_savepoint;  
COMMIT;
```

The above transaction will insert both 3 and 4.

Compatibility

This command conforms to the SQL standard. The standard specifies that the key word SAVEPOINT is mandatory, but Greenplum Database allows it to be omitted.

See Also

[BEGIN](#), [SAVEPOINT](#), [ROLLBACK TO SAVEPOINT](#), [COMMIT](#)

RESET

Restores the value of a system configuration parameter to the default value.

Synopsis

```
RESET configuration_parameter
```

```
RESET ALL
```

Description

RESET restores system configuration parameters to their default values. RESET is an alternative spelling for `SET configuration_parameter TO DEFAULT`.

The default value is defined as the value that the parameter would have had, had no SET ever been issued for it in the current session. The actual source of this value might be a compiled-in default, the master `postgresql.conf` configuration file, command-line options, or per-database or per-user default settings. See “[Server Configuration Parameters](#)” on page 755.

Parameters

configuration_parameter

The name of a system configuration parameter. See “[Server Configuration Parameters](#)” on page 755 for details.

ALL

Resets all settable configuration parameters to their default values.

Examples

Set the `work_mem` configuration parameter to its default value:

```
RESET work_mem;
```

Compatibility

RESET is a Greenplum Database extension.

See Also

[SET](#)

REVOKE

Removes access privileges.

Synopsis

```
REVOKE [GRANT OPTION FOR] { {SELECT | INSERT | UPDATE | DELETE
| REFERENCES | TRIGGER} [,...] | ALL [PRIVILEGES] }
ON [TABLE] tablename [, ...]
FROM {rolename | PUBLIC} [, ...]
[CASCADE | RESTRICT]
```

```
REVOKE [GRANT OPTION FOR] { {USAGE | SELECT | UPDATE} [,...]
| ALL [PRIVILEGES] }
ON SEQUENCE sequencename [, ...]
FROM { rolename | PUBLIC } [, ...]
[CASCADE | RESTRICT]
```

```
REVOKE [GRANT OPTION FOR] { {CREATE | CONNECT
| TEMPORARY | TEMP} [,...] | ALL [PRIVILEGES] }
ON DATABASE dbname [, ...]
FROM {rolename | PUBLIC} [, ...]
[CASCADE | RESTRICT]
```

```
REVOKE [GRANT OPTION FOR] {EXECUTE | ALL [PRIVILEGES]}
ON FUNCTION funcname ( [[argmode] [argname] argtype
[, ...]] ) [, ...]
FROM {rolename | PUBLIC} [, ...]
[CASCADE | RESTRICT]
```

```
REVOKE [GRANT OPTION FOR] {USAGE | ALL [PRIVILEGES]}
ON LANGUAGE langname [, ...]
FROM {rolename | PUBLIC} [, ...]
[ CASCADE | RESTRICT ]
```

```
REVOKE [GRANT OPTION FOR] { {CREATE | USAGE} [,...]
| ALL [PRIVILEGES] }
ON SCHEMA schemaname [, ...]
FROM {rolename | PUBLIC} [, ...]
[CASCADE | RESTRICT]
```

```
REVOKE [GRANT OPTION FOR] { CREATE | ALL [PRIVILEGES] }
ON TABLESPACE tablespacename [, ...]
FROM { rolename | PUBLIC } [, ...]
[CASCADE | RESTRICT]
```

```
REVOKE [ADMIN OPTION FOR] parent_role [, ...]
FROM member_role [, ...]
[CASCADE | RESTRICT]
```

Description

`REVOKE` command revokes previously granted privileges from one or more roles. The key word `PUBLIC` refers to the implicitly defined group of all roles.

See the description of the `GRANT` command for the meaning of the privilege types.

Note that any particular role will have the sum of privileges granted directly to it, privileges granted to any role it is presently a member of, and privileges granted to `PUBLIC`. Thus, for example, revoking `SELECT` privilege from `PUBLIC` does not necessarily mean that all roles have lost `SELECT` privilege on the object: those who have it granted directly or via another role will still have it.

If `GRANT OPTION FOR` is specified, only the grant option for the privilege is revoked, not the privilege itself. Otherwise, both the privilege and the grant option are revoked.

If a role holds a privilege with grant option and has granted it to other roles then the privileges held by those other roles are called dependent privileges. If the privilege or the grant option held by the first role is being revoked and dependent privileges exist, those dependent privileges are also revoked if `CASCADE` is specified, else the revoke action will fail. This recursive revocation only affects privileges that were granted through a chain of roles that is traceable to the role that is the subject of this `REVOKE` command. Thus, the affected roles may effectively keep the privilege if it was also granted through other roles.

When revoking membership in a role, `GRANT OPTION` is instead called `ADMIN OPTION`, but the behavior is similar.

Parameters

See `GRANT`.

Examples

Revoke insert privilege for the public on table *films*:

```
REVOKE INSERT ON films FROM PUBLIC;
```

Revoke all privileges from role *sally* on view *topten*. Note that this actually means revoke all privileges that the current role granted (if not a superuser).

```
REVOKE ALL PRIVILEGES ON topten FROM sally;
```

Revoke membership in role *admins* from user *joe*:

```
REVOKE admins FROM joe;
```

Compatibility

The compatibility notes of the `GRANT` command also apply to `REVOKE`.

One of `RESTRICT` or `CASCADE` is required according to the standard, but Greenplum Database assumes `RESTRICT` by default.

See Also

[GRANT](#)

ROLLBACK

Aborts the current transaction.

Synopsis

```
ROLLBACK [WORK | TRANSACTION]
```

Description

ROLLBACK rolls back the current transaction and causes all the updates made by the transaction to be discarded.

Parameters

WORK
TRANSACTION

Optional key words. They have no effect.

Notes

Use `COMMIT` to successfully end the current transaction.

Issuing `ROLLBACK` when not inside a transaction does no harm, but it will provoke a warning message.

Examples

To discard all changes made in the current transaction:

```
ROLLBACK;
```

Compatibility

The SQL standard only specifies the two forms `ROLLBACK` and `ROLLBACK WORK`. Otherwise, this command is fully conforming.

See Also

[BEGIN](#), [COMMIT](#), [SAVEPOINT](#), [ROLLBACK TO SAVEPOINT](#)

ROLLBACK TO SAVEPOINT

Rolls back the current transaction to a savepoint.

Synopsis

```
ROLLBACK [WORK | TRANSACTION] TO [SAVEPOINT] savepoint_name
```

Description

This command will roll back all commands that were executed after the savepoint was established. The savepoint remains valid and can be rolled back to again later, if needed.

ROLLBACK TO SAVEPOINT implicitly destroys all savepoints that were established after the named savepoint.

Parameters

WORK
TRANSACTION

Optional key words. They have no effect.

savepoint_name

The name of a savepoint to roll back to.

Notes

Use `RELEASE SAVEPOINT` to destroy a savepoint without discarding the effects of commands executed after it was established.

Specifying a savepoint name that has not been established is an error.

Cursors have somewhat non-transactional behavior with respect to savepoints. Any cursor that is opened inside a savepoint will be closed when the savepoint is rolled back. If a previously opened cursor is affected by a `FETCH` command inside a savepoint that is later rolled back, the cursor position remains at the position that `FETCH` left it pointing to (that is, `FETCH` is not rolled back). Closing a cursor is not undone by rolling back, either. A cursor whose execution causes a transaction to abort is put in a can't-execute state, so while the transaction can be restored using `ROLLBACK TO SAVEPOINT`, the cursor can no longer be used.

Examples

To undo the effects of the commands executed after *my_savepoint* was established:

```
ROLLBACK TO SAVEPOINT my_savepoint;
```

Cursor positions are not affected by a savepoint rollback:

```
BEGIN;
DECLARE foo CURSOR FOR SELECT 1 UNION SELECT 2;
```

```
SAVEPOINT foo;
FETCH 1 FROM foo;
column
-----
          1
ROLLBACK TO SAVEPOINT foo;
FETCH 1 FROM foo;
column
-----
          2
COMMIT;
```

Compatibility

The SQL standard specifies that the key word `SAVEPOINT` is mandatory, but Greenplum Database (and Oracle) allow it to be omitted. SQL allows only `WORK`, not `TRANSACTION`, as a noise word after `ROLLBACK`. Also, SQL has an optional clause `AND [NO] CHAIN` which is not currently supported by Greenplum Database. Otherwise, this command conforms to the SQL standard.

See Also

[BEGIN](#), [COMMIT](#), [SAVEPOINT](#), [RELEASE SAVEPOINT](#), [ROLLBACK](#)

SAVEPOINT

Defines a new savepoint within the current transaction.

Synopsis

```
SAVEPOINT savepoint_name
```

Description

SAVEPOINT establishes a new savepoint within the current transaction.

A savepoint is a special mark inside a transaction that allows all commands that are executed after it was established to be rolled back, restoring the transaction state to what it was at the time of the savepoint.

Parameters

savepoint_name

The name of the new savepoint.

Notes

Use `ROLLBACK TO SAVEPOINT` to rollback to a savepoint. Use `RELEASE SAVEPOINT` to destroy a savepoint, keeping the effects of commands executed after it was established.

Savepoints can only be established when inside a transaction block. There can be multiple savepoints defined within a transaction.

Examples

To establish a savepoint and later undo the effects of all commands executed after it was established:

```
BEGIN;  
    INSERT INTO table1 VALUES (1);  
    SAVEPOINT my_savepoint;  
    INSERT INTO table1 VALUES (2);  
    ROLLBACK TO SAVEPOINT my_savepoint;  
    INSERT INTO table1 VALUES (3);  
COMMIT;
```

The above transaction will insert the values 1 and 3, but not 2.

To establish and later destroy a savepoint:

```
BEGIN;  
    INSERT INTO table1 VALUES (3);
```

```
SAVEPOINT my_savepoint;  
INSERT INTO table1 VALUES (4);  
RELEASE SAVEPOINT my_savepoint;  
COMMIT;
```

The above transaction will insert both 3 and 4.

Compatibility

SQL requires a savepoint to be destroyed automatically when another savepoint with the same name is established. In Greenplum Database, the old savepoint is kept, though only the more recent one will be used when rolling back or releasing. (Releasing the newer savepoint will cause the older one to again become accessible to [ROLLBACK TO SAVEPOINT](#) and [RELEASE SAVEPOINT](#).) Otherwise, `SAVEPOINT` is fully SQL conforming.

See Also

[BEGIN](#), [COMMIT](#), [ROLLBACK](#), [RELEASE SAVEPOINT](#), [ROLLBACK TO SAVEPOINT](#)

SELECT

Retrieves rows from a table or view.

Synopsis

```

SELECT [ALL | DISTINCT [ON (expression [, ...])]]
      * | expression [[AS] output_name] [, ...]
[FROM from_item [, ...]]
[WHERE condition]
[GROUP BY grouping_element [, ...]]
[HAVING condition [, ...]]
[WINDOW window_name AS (window_specification)]
[{UNION | INTERSECT | EXCEPT} [ALL] select]
[ORDER BY expression [ASC | DESC | USING operator] [, ...]]
[LIMIT {count | ALL}]
[OFFSET start]
[FOR {UPDATE | SHARE} [OF table_name [, ...]] [NOWAIT] [...]]

```

where *grouping_element* can be one of:

```

()
expression
ROLLUP (expression [, ...])
CUBE (expression [, ...])
GROUPING SETS ((grouping_element [, ...]))

```

where *window_specification* can be:

```

[window_name]
[PARTITION BY expression [, ...]]
[ORDER BY expression [ASC | DESC | USING operator] [, ...]
  [{RANGE | ROWS}
    { UNBOUNDED PRECEDING
      | expression PRECEDING
      | CURRENT ROW
      | BETWEEN window_frame_bound AND window_frame_bound }]]

```

where *window_frame_bound* can be one of:

```

UNBOUNDED PRECEDING
expression PRECEDING
CURRENT ROW
expression FOLLOWING
UNBOUNDED FOLLOWING

```

where *from_item* can be one of:

```

[ONLY] table_name [[AS] alias [( column_alias [, ...] )]]
(select) [AS] alias [( column_alias [, ...] )]
function_name ( [argument [, ...]] ) [AS] alias
  [( column_alias [, ...]
    | column_definition [, ...] )]
function_name ( [argument [, ...]] ) AS
  ( column_definition [, ...] )

```

```

from_item [NATURAL] join_type from_item
        [ON join_condition | USING ( join_column [, ...] ) ]

```

Description

`SELECT` retrieves rows from zero or more tables. The general processing of `SELECT` is as follows:

1. All elements in the `FROM` list are computed. (Each element in the `FROM` list is a real or virtual table.) If more than one element is specified in the `FROM` list, they are cross-joined together.
2. If the `WHERE` clause is specified, all rows that do not satisfy the condition are eliminated from the output.
3. If the `GROUP BY` clause is specified, the output is divided into groups of rows that match on one or more of the defined grouping elements. If the `HAVING` clause is present, it eliminates groups that do not satisfy the given condition.
4. If a window expression is specified (and optional `WINDOW` clause), the output is organized according to the positional (row) or value-based (range) window frame.
5. `DISTINCT` eliminates duplicate rows from the result. `DISTINCT ON` eliminates rows that match on all the specified expressions. `ALL` (the default) will return all candidate rows, including duplicates.
6. The actual output rows are computed using the `SELECT` output expressions for each selected row.
7. Using the operators `UNION`, `INTERSECT`, and `EXCEPT`, the output of more than one `SELECT` statement can be combined to form a single result set. The `UNION` operator returns all rows that are in one or both of the result sets. The `INTERSECT` operator returns all rows that are strictly in both result sets. The `EXCEPT` operator returns the rows that are in the first result set but not in the second. In all three cases, duplicate rows are eliminated unless `ALL` is specified.
8. If the `ORDER BY` clause is specified, the returned rows are sorted in the specified order. If `ORDER BY` is not given, the rows are returned in whatever order the system finds fastest to produce.
9. If the `LIMIT` or `OFFSET` clause is specified, the `SELECT` statement only returns a subset of the result rows.
10. If `FOR UPDATE` or `FOR SHARE` is specified, the `SELECT` statement locks the entire table against concurrent updates.

You must have `SELECT` privilege on a table to read its values. The use of `FOR UPDATE` or `FOR SHARE` requires `UPDATE` privilege as well.

Parameters

The SELECT List

The `SELECT` list (between the key words `SELECT` and `FROM`) specifies expressions that form the output rows of the `SELECT` statement. The expressions can (and usually do) refer to columns computed in the `FROM` clause.

Using the clause `[AS] output_name`, another name can be specified for an output column. This name is primarily used to label the column for display. It can also be used to refer to the column's value in `ORDER BY` and `GROUP BY` clauses, but not in the `WHERE` or `HAVING` clauses; there you must write out the expression instead. The `AS` keyword is optional in most cases (such as when declaring an alias for column names, constants, function calls, and simple unary operator expressions). In cases where the declared alias is a reserved SQL keyword, the `output_name` must be enclosed in double quotes to avoid ambiguity.

An *expression* in the `SELECT` list can be a constant value, a column reference, an operator invocation, a function call, an aggregate expression, a window expression, a scalar subquery, and so on. See “SQL Value Expressions” on page 131 for more information. There are a number of constructs that can be classified as an expression but do not follow any general syntax rules. These generally have the semantics of a function or operator and are explained in “Function Calls” on page 134.

Instead of an expression, `*` can be written in the output list as a shorthand for all the columns of the selected rows. Also, one can write `table_name.*` as a shorthand for the columns coming from just that table.

The FROM Clause

The `FROM` clause specifies one or more source tables for the `SELECT`. If multiple sources are specified, the result is the Cartesian product (cross join) of all the sources. But usually qualification conditions are added to restrict the returned rows to a small subset of the Cartesian product. The `FROM` clause can contain the following elements:

table_name

The name (optionally schema-qualified) of an existing table or view. If `ONLY` is specified, only that table is scanned. If `ONLY` is not specified, the table and all its descendant tables (if any) are scanned.

alias

A substitute name for the `FROM` item containing the alias. An alias is used for brevity or to eliminate ambiguity for self-joins (where the same table is scanned multiple times). When an alias is provided, it completely hides the actual name of the table or function; for example given `FROM foo AS f`, the remainder of the `SELECT` must refer to this `FROM` item as `f` not `foo`. If an alias is written, a column alias list can also be written to provide substitute names for one or more columns of the table.

select

A sub-`SELECT` can appear in the `FROM` clause. This acts as though its output were created as a temporary table for the duration of this single `SELECT` command. Note that the sub-`SELECT` must be surrounded by parentheses, and an alias must be

provided for it. A `VALUES` command can also be used here. See “[Limited Correlated Subquery Syntax](#)” on page 536 for limitations of using correlated sub-selects in Greenplum Database.

function_name

Function calls can appear in the `FROM` clause. (This is especially useful for functions that return result sets, but any function can be used.) This acts as though its output were created as a temporary table for the duration of this single `SELECT` command. An alias may also be used. If an alias is written, a column alias list can also be written to provide substitute names for one or more attributes of the function’s composite return type. If the function has been defined as returning the record data type, then an alias or the key word `AS` must be present, followed by a column definition list in the form (`column_name data_type [, ...]`). The column definition list must match the actual number and types of columns returned by the function.

join_type

One of:

- `[INNER] JOIN`
- `LEFT [OUTER] JOIN`
- `RIGHT [OUTER] JOIN`
- `FULL [OUTER] JOIN`
- `CROSS JOIN`

For the `INNER` and `OUTER` join types, a join condition must be specified, namely exactly one of `NATURAL`, `ON join_condition`, or `USING (join_column [, ...])`. See below for the meaning. For `CROSS JOIN`, none of these clauses may appear.

A `JOIN` clause combines two `FROM` items. Use parentheses if necessary to determine the order of nesting. In the absence of parentheses, `JOINS` nest left-to-right. In any case `JOIN` binds more tightly than the commas separating `FROM` items.

`CROSS JOIN` and `INNER JOIN` produce a simple Cartesian product, the same result as you get from listing the two items at the top level of `FROM`, but restricted by the join condition (if any). `CROSS JOIN` is equivalent to `INNER JOIN ON (TRUE)`, that is, no rows are removed by qualification. These join types are just a notational convenience, since they do nothing you could not do with plain `FROM` and `WHERE`.

`LEFT OUTER JOIN` returns all rows in the qualified Cartesian product (i.e., all combined rows that pass its join condition), plus one copy of each row in the left-hand table for which there was no right-hand row that passed the join condition. This left-hand row is extended to the full width of the joined table by inserting null values for the right-hand columns. Note that only the `JOIN` clause’s own condition is considered while deciding which rows have matches. Outer conditions are applied afterwards.

Conversely, `RIGHT OUTER JOIN` returns all the joined rows, plus one row for each unmatched right-hand row (extended with nulls on the left). This is just a notational convenience, since you could convert it to a `LEFT OUTER JOIN` by switching the left and right inputs.

`FULL OUTER JOIN` returns all the joined rows, plus one row for each unmatched left-hand row (extended with nulls on the right), plus one row for each unmatched right-hand row (extended with nulls on the left).

ON *join_condition*

join_condition is an expression resulting in a value of type `boolean` (similar to a `WHERE` clause) that specifies which rows in a join are considered to match.

USING (*join_column* [, ...])

A clause of the form `USING (a, b, ...)` is shorthand for `ON left_table.a = right_table.a AND left_table.b = right_table.b` Also, `USING` implies that only one of each pair of equivalent columns will be included in the join output, not both.

NATURAL

`NATURAL` is shorthand for a `USING` list that mentions all columns in the two tables that have the same names.

The WHERE Clause

The optional `WHERE` clause has the general form:

```
WHERE condition
```

Where *condition* is any expression that evaluates to a result of type `boolean`. Any row that does not satisfy this condition will be eliminated from the output. A row satisfies the condition if it returns true when the actual row values are substituted for any variable references.

The GROUP BY Clause

The optional `GROUP BY` clause has the general form:

```
GROUP BY grouping_element [, ...]
```

where *grouping_element* can be one of:

```
(  
  expression  
  ROLLUP (expression [, ...])  
  CUBE (expression [, ...])  
  GROUPING SETS ((grouping_element [, ...]))
```

`GROUP BY` will condense into a single row all selected rows that share the same values for the grouped expressions. *expression* can be an input column name, or the name or ordinal number of an output column (`SELECT` list item), or an arbitrary expression formed from input-column values. In case of ambiguity, a `GROUP BY` name will be interpreted as an input-column name rather than an output column name.

Aggregate functions, if any are used, are computed across all rows making up each group, producing a separate value for each group (whereas without `GROUP BY`, an aggregate produces a single value computed across all the selected rows). When

`GROUP BY` is present, it is not valid for the `SELECT` list expressions to refer to ungrouped columns except within aggregate functions, since there would be more than one possible value to return for an ungrouped column.

Greenplum Database has the following additional OLAP grouping extensions (often referred to as *supergroups*):

ROLLUP

A `ROLLUP` grouping is an extension to the `GROUP BY` clause that creates aggregate subtotals that roll up from the most detailed level to a grand total, following a list of grouping columns (or expressions). `ROLLUP` takes an ordered list of grouping columns, calculates the standard aggregate values specified in the `GROUP BY` clause, then creates progressively higher-level subtotals, moving from right to left through the list. Finally, it creates a grand total. A `ROLLUP` grouping can be thought of as a series of grouping sets. For example:

```
GROUP BY ROLLUP (a,b,c)
```

is equivalent to:

```
GROUP BY GROUPING SETS( (a,b,c), (a,b), (a), () )
```

Notice that the n elements of a `ROLLUP` translate to $n+1$ grouping sets. Also, the order in which the grouping expressions are specified is significant in a `ROLLUP`.

CUBE

A `CUBE` grouping is an extension to the `GROUP BY` clause that creates subtotals for all of the possible combinations of the given list of grouping columns (or expressions). In terms of multidimensional analysis, `CUBE` generates all the subtotals that could be calculated for a data cube with the specified dimensions. For example:

```
GROUP BY CUBE (a,b,c)
```

is equivalent to:

```
GROUP BY GROUPING SETS( (a,b,c), (a,b), (a,c), (b,c), (a),
(b), (c), () )
```

Notice that n elements of a `CUBE` translate to 2^n grouping sets. Consider using `CUBE` in any situation requiring cross-tabular reports. `CUBE` is typically most suitable in queries that use columns from multiple dimensions rather than columns representing different levels of a single dimension. For instance, a commonly requested cross-tabulation might need subtotals for all the combinations of month, state, and product.

GROUPING SETS

You can selectively specify the set of groups that you want to create using a `GROUPING SETS` expression within a `GROUP BY` clause. This allows precise specification across multiple dimensions without computing a whole `ROLLUP` or `CUBE`. For example:

```
GROUP BY GROUPING SETS( (a,c), (a,b) )
```

If using the grouping extension clauses `ROLLUP`, `CUBE`, or `GROUPING SETS`, two challenges arise. First, how do you determine which result rows are subtotals, and then the exact level of aggregation for a given subtotal. Or, how do you differentiate between result rows that contain both stored `NULL` values and “NULL” values created by the `ROLLUP` or `CUBE`. Secondly, when duplicate grouping sets are specified in the `GROUP BY` clause, how do you determine which result rows are duplicates? There are two additional grouping functions you can use in the `SELECT` list to help with this:

- **grouping(column [, ...])** The `grouping` function can be applied to one or more grouping attributes to distinguish super-aggregated rows from regular grouped rows. This can be helpful in distinguishing a “NULL” representing the set of all values in a super-aggregated row from a `NULL` value in a regular row. Each argument in this function produces a bit — either 1 or 0, where 1 means the result row is super-aggregated, and 0 means the result row is from a regular grouping. The `grouping` function returns an integer by treating these bits as a binary number and then converting it to a base-10 integer.
- **group_id()** For grouping extension queries that contain duplicate grouping sets, the `group_id` function is used to identify duplicate rows in the output. All *unique* grouping set output rows will have a `group_id` value of 0. For each duplicate grouping set detected, the `group_id` function assigns a `group_id` number greater than 0. All output rows in a particular duplicate grouping set are identified by the same `group_id` number.

The WINDOW Clause

The `WINDOW` clause is used to define a window that can be used in the `OVER()` expression of a window function such as `rank` or `avg`. For example:

```
SELECT vendor, rank() OVER (mywindow) FROM sale
WINDOW mywindow AS (ORDER BY sum(prc*qty));
```

A `WINDOW` clause is has this general form:

```
WINDOW window_name AS (window_specification)
where window_specification can be:
[window_name]
[PARTITION BY expression [, ...]]
[ORDER BY expression [ASC | DESC | USING operator] [, ...]
  [{RANGE | ROWS}
   { UNBOUNDED PRECEDING
   | expression PRECEDING
   | CURRENT ROW
   | BETWEEN window_frame_bound AND window_frame_bound }]]
```

where `window_frame_bound` can be one of:

```
UNBOUNDED PRECEDING
expression PRECEDING
CURRENT ROW
expression FOLLOWING
UNBOUNDED FOLLOWING
```

window_name

Gives a name to the window specification.

PARTITION BY

The `PARTITION BY` clause organizes the result set into logical groups based on the unique values of the specified expression. When used with window functions, the functions are applied to each partition independently. For example, if you follow `PARTITION BY` with a column name, the result set is partitioned by the distinct values of that column. If omitted, the entire result set is considered one partition.

ORDER BY

The `ORDER BY` clause defines how to sort the rows in each partition of the result set. If omitted, rows are returned in whatever order is most efficient and may vary.

NOTE: Columns of data types that lack a coherent ordering, such as `time`, are not good candidates for use in the `ORDER BY` clause of a window specification. Time, with or without time zone, lacks a coherent ordering because addition and subtraction do not have the expected effects. For example, the following is not generally true: `x::time < x::time + '2 hour'::interval`

ROWS | RANGE

Use either a `ROWS` or `RANGE` clause to express the bounds of the window. The window bound can be one, many, or all rows of a partition. You can express the bound of the window either in terms of a range of data values offset from the value in the current row (`RANGE`), or in terms of the number of rows offset from the current row (`ROWS`). When using the `RANGE` clause, you must also use an `ORDER BY` clause. This is because the calculation performed to produce the window requires that the values be sorted. Additionally, the `ORDER BY` clause cannot contain more than one expression, and the expression must result in either a date or a numeric value. When using the `ROWS` or `RANGE` clauses, if you specify only a starting row, the current row is used as the last row in the window.

PRECEDING

The `PRECEDING` clause defines the first row of the window using the current row as a reference point. The starting row is expressed in terms of the number of rows preceding the current row. For example, in the case of `ROWS` framing, `5 PRECEDING` sets the window to start with the fifth row preceding the current row. In the case of `RANGE` framing, it sets the window to start with the first row whose ordering column value precedes that of the current row by 5 in the given order. If the specified order is ascending by date, this will be the first row within 5 days before the current row. `UNBOUNDED PRECEDING` sets the first row in the window to be the first row in the partition.

BETWEEN

The `BETWEEN` clause defines the first and last row of the window, using the current row as a reference point. First and last rows are expressed in terms of the number of rows preceding and following the current row, respectively. For example, `BETWEEN 3 PRECEDING AND 5 FOLLOWING` sets the window to start with the third row preceding the current row, and end with the fifth row following the current row. Use `BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED`

`FOLLOWING` to set the first and last rows in the window to be the first and last row in the partition, respectively. This is equivalent to the default behavior if no `ROW` or `RANGE` clause is specified.

FOLLOWING

The `FOLLOWING` clause defines the last row of the window using the current row as a reference point. The last row is expressed in terms of the number of rows following the current row. For example, in the case of `ROWS` framing, `5 FOLLOWING` sets the window to end with the fifth row following the current row. In the case of `RANGE` framing, it sets the window to end with the last row whose ordering column value follows that of the current row by 5 in the given order. If the specified order is ascending by date, this will be the last row within 5 days after the current row. Use `UNBOUNDED FOLLOWING` to set the last row in the window to be the last row in the partition.

If you do not specify a `ROW` or a `RANGE` clause, the window bound starts with the first row in the partition (`UNBOUNDED PRECEDING`) and ends with the current row (`CURRENT ROW`) if `ORDER BY` is used. If an `ORDER BY` is not specified, the window starts with the first row in the partition (`UNBOUNDED PRECEDING`) and ends with last row in the partition (`UNBOUNDED FOLLOWING`).

The HAVING Clause

The optional `HAVING` clause has the general form:

```
HAVING condition
```

Where *condition* is the same as specified for the `WHERE` clause. `HAVING` eliminates group rows that do not satisfy the condition. `HAVING` is different from `WHERE`: `WHERE` filters individual rows before the application of `GROUP BY`, while `HAVING` filters group rows created by `GROUP BY`. Each column referenced in *condition* must unambiguously reference a grouping column, unless the reference appears within an aggregate function.

The presence of `HAVING` turns a query into a grouped query even if there is no `GROUP BY` clause. This is the same as what happens when the query contains aggregate functions but no `GROUP BY` clause. All the selected rows are considered to form a single group, and the `SELECT` list and `HAVING` clause can only reference table columns from within aggregate functions. Such a query will emit a single row if the `HAVING` condition is true, zero rows if it is not true.

The UNION Clause

The `UNION` clause has this general form:

```
select_statement UNION [ALL] select_statement
```

Where *select_statement* is any `SELECT` statement without an `ORDER BY`, `LIMIT`, `FOR UPDATE`, or `FOR SHARE` clause. (`ORDER BY` and `LIMIT` can be attached to a subquery expression if it is enclosed in parentheses. Without parentheses, these clauses will be taken to apply to the result of the `UNION`, not to its right-hand input expression.)

The `UNION` operator computes the set union of the rows returned by the involved `SELECT` statements. A row is in the set union of two result sets if it appears in at least one of the result sets. The two `SELECT` statements that represent the direct operands of the `UNION` must produce the same number of columns, and corresponding columns must be of compatible data types.

The result of `UNION` does not contain any duplicate rows unless the `ALL` option is specified. `ALL` prevents elimination of duplicates. (Therefore, `UNION ALL` is usually significantly quicker than `UNION`; use `ALL` when you can.)

Multiple `UNION` operators in the same `SELECT` statement are evaluated left to right, unless otherwise indicated by parentheses.

Currently, `FOR UPDATE` and `FOR SHARE` may not be specified either for a `UNION` result or for any input of a `UNION`.

The `INTERSECT` Clause

The `INTERSECT` clause has this general form:

```
select_statement INTERSECT [ALL] select_statement
```

Where *select_statement* is any `SELECT` statement without an `ORDER BY`, `LIMIT`, `FOR UPDATE`, or `FOR SHARE` clause.

The `INTERSECT` operator computes the set intersection of the rows returned by the involved `SELECT` statements. A row is in the intersection of two result sets if it appears in both result sets.

The result of `INTERSECT` does not contain any duplicate rows unless the `ALL` option is specified. With `ALL`, a row that has m duplicates in the left table and n duplicates in the right table will appear $\min(m, n)$ times in the result set.

Multiple `INTERSECT` operators in the same `SELECT` statement are evaluated left to right, unless parentheses dictate otherwise. `INTERSECT` binds more tightly than `UNION`. That is, `A UNION B INTERSECT C` will be read as `A UNION (B INTERSECT C)`.

Currently, `FOR UPDATE` and `FOR SHARE` may not be specified either for an `INTERSECT` result or for any input of an `INTERSECT`.

The `EXCEPT` Clause

The `EXCEPT` clause has this general form:

```
select_statement EXCEPT [ALL] select_statement
```

Where *select_statement* is any `SELECT` statement without an `ORDER BY`, `LIMIT`, `FOR UPDATE`, or `FOR SHARE` clause.

The `EXCEPT` operator computes the set of rows that are in the result of the left `SELECT` statement but not in the result of the right one.

The result of `EXCEPT` does not contain any duplicate rows unless the `ALL` option is specified. With `ALL`, a row that has m duplicates in the left table and n duplicates in the right table will appear $\max(m-n, 0)$ times in the result set.

Multiple `EXCEPT` operators in the same `SELECT` statement are evaluated left to right, unless parentheses dictate otherwise. `EXCEPT` binds at the same level as `UNION`.

Currently, `FOR UPDATE` and `FOR SHARE` may not be specified either for an `EXCEPT` result or for any input of an `EXCEPT`.

The `ORDER BY` Clause

The optional `ORDER BY` clause has this general form:

```
ORDER BY expression [ASC | DESC | USING operator] [, ...]
```

Where *expression* can be the name or ordinal number of an output column (`SELECT` list item), or it can be an arbitrary expression formed from input-column values.

The `ORDER BY` clause causes the result rows to be sorted according to the specified expressions. If two rows are equal according to the left-most expression, they are compared according to the next expression and so on. If they are equal according to all specified expressions, they are returned in an implementation-dependent order.

The ordinal number refers to the ordinal (left-to-right) position of the result column. This feature makes it possible to define an ordering on the basis of a column that does not have a unique name. This is never absolutely necessary because it is always possible to assign a name to a result column using the `AS` clause.

It is also possible to use arbitrary expressions in the `ORDER BY` clause, including columns that do not appear in the `SELECT` result list. Thus the following statement is valid:

```
SELECT name FROM distributors ORDER BY code;
```

A limitation of this feature is that an `ORDER BY` clause applying to the result of a `UNION`, `INTERSECT`, or `EXCEPT` clause may only specify an output column name or number, not an expression.

If an `ORDER BY` expression is a simple name that matches both a result column name and an input column name, `ORDER BY` will interpret it as the result column name. This is the opposite of the choice that `GROUP BY` will make in the same situation. This inconsistency is made to be compatible with the SQL standard.

Optionally one may add the key word `ASC` (ascending) or `DESC` (descending) after any expression in the `ORDER BY` clause. If not specified, `ASC` is assumed by default. Alternatively, a specific ordering operator name may be specified in the `USING` clause. `ASC` is usually equivalent to `USING <` and `DESC` is usually equivalent to `USING >`. (But the creator of a user-defined data type can define exactly what the default sort ordering is, and it might correspond to operators with other names.)

The null value sorts higher than any other value. In other words, with ascending sort order, null values sort at the end, and with descending sort order, null values sort at the beginning.

Character-string data is sorted according to the locale-specific collation order that was established when the Greenplum Database system was initialized.

The `DISTINCT` Clause

If `DISTINCT` is specified, all duplicate rows are removed from the result set (one row is kept from each group of duplicates). `ALL` specifies the opposite: all rows are kept. `ALL` is the default.

`DISTINCT ON (expression [, ...])` keeps only the first row of each set of rows where the given expressions evaluate to equal. The `DISTINCT ON` expressions are interpreted using the same rules as for `ORDER BY`. Note that the ‘first row’ of each set is unpredictable unless `ORDER BY` is used to ensure that the desired row appears first. For example:

```
SELECT DISTINCT ON (location) location, time, report FROM
weather_reports ORDER BY location, time DESC;
```

retrieves the most recent weather report for each location. But if we had not used `ORDER BY` to force descending order of time values for each location, we would have gotten a report from an unpredictable time for each location.

The `DISTINCT ON` expression(s) must match the left-most `ORDER BY` expression(s). The `ORDER BY` clause will normally contain additional expression(s) that determine the desired precedence of rows within each `DISTINCT ON` group.

The LIMIT Clause

The `LIMIT` clause consists of two independent sub-clauses:

```
LIMIT {count | ALL}
OFFSET start
```

Where *count* specifies the maximum number of rows to return, while *start* specifies the number of rows to skip before starting to return rows. When both are specified, start rows are skipped before starting to count the count rows to be returned.

When using `LIMIT`, it is a good idea to use an `ORDER BY` clause that constrains the result rows into a unique order. Otherwise you will get an unpredictable subset of the query’s rows — you may be asking for the tenth through twentieth rows, but tenth through twentieth in what ordering? You don’t know what ordering unless you specify `ORDER BY`.

The query planner takes `LIMIT` into account when generating a query plan, so you are very likely to get different plans (yielding different row orders) depending on what you use for `LIMIT` and `OFFSET`. Thus, using different `LIMIT/OFFSET` values to select different subsets of a query result will give inconsistent results unless you enforce a predictable result ordering with `ORDER BY`. This is not a defect; it is an inherent consequence of the fact that SQL does not promise to deliver the results of a query in any particular order unless `ORDER BY` is used to constrain the order.

FOR UPDATE/FOR SHARE Clause

The `FOR UPDATE` clause has this form:

```
FOR UPDATE [OF table_name [, ...]] [NOWAIT]
```

The closely related `FOR SHARE` clause has this form:

```
FOR SHARE [OF table_name [, ...]] [NOWAIT]
```

`FOR UPDATE` causes the tables accessed by the `SELECT` statement to be locked as though for update. This prevents the table from being modified or deleted by other transactions until the current transaction ends. That is, other transactions that attempt `UPDATE`, `DELETE`, or `SELECT FOR UPDATE` of this table will be blocked until the current transaction ends. Also, if an `UPDATE`, `DELETE`, or `SELECT FOR UPDATE` from

another transaction has already locked a selected table, `SELECT FOR UPDATE` will wait for the other transaction to complete, and will then lock and return the updated table. Note that Greenplum Database takes a more restrictive lock for updates than does PostgreSQL.

To prevent the operation from waiting for other transactions to commit, use the `NOWAIT` option. `SELECT FOR UPDATE NOWAIT` reports an error, rather than waiting, if a selected row cannot be locked immediately. Note that `NOWAIT` applies only to the row-level lock(s) — the required `ROW SHARE` table-level lock is still taken in the ordinary way. You can use the `NOWAIT` option of `LOCK` if you need to acquire the table-level lock without waiting (see `LOCK`).

`FOR SHARE` behaves similarly, except that it acquires a shared rather than exclusive lock on the table. A shared lock blocks other transactions from performing `UPDATE`, `DELETE`, or `SELECT FOR UPDATE` on the table, but it does not prevent them from performing `SELECT FOR SHARE`.

If specific tables are named in `FOR UPDATE` or `FOR SHARE`, then only those tables are locked; any other tables used in the `SELECT` are simply read as usual. A `FOR UPDATE` or `FOR SHARE` clause without a table list affects all tables used in the command. If `FOR UPDATE` or `FOR SHARE` is applied to a view or subquery, it affects all tables used in the view or subquery.

Multiple `FOR UPDATE` and `FOR SHARE` clauses can be written if it is necessary to specify different locking behavior for different tables. If the same table is mentioned (or implicitly affected) by both `FOR UPDATE` and `FOR SHARE` clauses, then it is processed as `FOR UPDATE`. Similarly, a table is processed as `NOWAIT` if that is specified in any of the clauses affecting it.

Examples

To join the table *films* with the table *distributors*:

```
SELECT f.title, f.did, d.name, f.date_prod, f.kind FROM
distributors d, films f WHERE f.did = d.did
```

To sum the column *length* of all films and group the results by *kind*:

```
SELECT kind, sum(length) AS total FROM films GROUP BY kind;
```

To sum the column *length* of all films, group the results by *kind* and show those group totals that are less than 5 hours:

```
SELECT kind, sum(length) AS total FROM films GROUP BY kind
HAVING sum(length) < interval '5 hours';
```

Calculate the subtotals and grand totals of all sales for movie *kind* and *distributor*.

```
SELECT kind, distributor, sum(prc*qty) FROM sales
GROUP BY ROLLUP(kind, distributor)
ORDER BY 1,2,3;
```

Calculate the rank of movie distributors based on total sales:

```
SELECT distributor, sum(prc*qty),
rank() OVER (ORDER BY sum(prc*qty) DESC)
```

```
FROM sale
GROUP BY distributor ORDER BY 2 DESC;
```

The following two examples are identical ways of sorting the individual results according to the contents of the second column (*name*):

```
SELECT * FROM distributors ORDER BY name;
SELECT * FROM distributors ORDER BY 2;
```

The next example shows how to obtain the union of the tables *distributors* and *actors*, restricting the results to those that begin with the letter *W* in each table. Only distinct rows are wanted, so the key word `ALL` is omitted:

```
SELECT distributors.name FROM distributors WHERE
distributors.name LIKE 'W%' UNION SELECT actors.name FROM
actors WHERE actors.name LIKE 'W%';
```

This example shows how to use a function in the `FROM` clause, both with and without a column definition list:

```
CREATE FUNCTION distributors(int) RETURNS SETOF distributors
AS $$ SELECT * FROM distributors WHERE did = $1; $$ LANGUAGE
SQL;
SELECT * FROM distributors(111);
```

```
CREATE FUNCTION distributors_2(int) RETURNS SETOF record AS
$$ SELECT * FROM distributors WHERE did = $1; $$ LANGUAGE
SQL;
SELECT * FROM distributors_2(111) AS (dist_id int, dist_name
text);
```

Compatibility

The `SELECT` statement is compatible with the SQL standard, but there are some extensions and some missing features.

Omitted FROM Clauses

Greenplum Database allows one to omit the `FROM` clause. It has a straightforward use to compute the results of simple expressions. For example:

```
SELECT 2+2;
```

Some other SQL databases cannot do this except by introducing a dummy one-row table from which to do the `SELECT`.

Note that if a `FROM` clause is not specified, the query cannot reference any database tables. For compatibility with applications that rely on this behavior the `add_missing_from` configuration variable can be enabled.

The AS Key Word

In the SQL standard, the optional key word `AS` is just noise and can be omitted without affecting the meaning. The Greenplum Database parser requires this key word when renaming output columns because the type extensibility features lead to parsing ambiguities without it. `AS` is optional in `FROM` items, however.

Namespace Available to GROUP BY and ORDER BY

In the SQL-92 standard, an `ORDER BY` clause may only use result column names or numbers, while a `GROUP BY` clause may only use expressions based on input column names. Greenplum Database extends each of these clauses to allow the other choice as well (but it uses the standard's interpretation if there is ambiguity). Greenplum Database also allows both clauses to specify arbitrary expressions. Note that names appearing in an expression will always be taken as input-column names, not as result-column names.

SQL:1999 and later use a slightly different definition which is not entirely upward compatible with SQL-92. In most cases, however, Greenplum Database will interpret an `ORDER BY` or `GROUP BY` expression the same way SQL:1999 does.

Nonstandard Clauses

The clauses `DISTINCT ON`, `LIMIT`, and `OFFSET` are not defined in the SQL standard.

Limited Correlated Subquery Syntax

A correlated subquery is a nested `SELECT` statement that refers to a column from an outer `SELECT` statement. For example:

```
SELECT * FROM product WHERE exists (SELECT * FROM sale WHERE
    qty>0 AND pn = product.pn);
```

Subqueries that do not need any input from the parent query and produce just a single value or row are not subject to the limitations described in this section.

Greenplum Database (as does PostgreSQL) supports correlated subqueries by *flattening* (transforming the subquery into a join with the table(s) of the outer query). If a correlated subquery has none of the characteristics mentioned below that prevent flattening, then it will be transformed to a join and be executed as expected. Correlated subqueries with the following syntax cannot be flattened and will be rejected:

- *expression operator ANY/ALL (subquery) where expression contains a volatile function*
- `EXISTS` with aggregate in `HAVING`
- *(subquery) returning a single value or row*
- `ARRAY (subquery) returning an array of values or rows`
- Correlated subquery used in `WHERE` clause
- Correlated subquery used in `HAVING` clause
- Correlated subquery used in `OR` expression
- Correlated subquery used in *select-list*
- Set-returning function in *select-list* of subquery

- Aggregate function in subquery (other than `EXISTS`)
- `DISTINCT` in value, row or `ARRAY` subquery
- `GROUP BY` or `DISTINCT` in subquery (other than `EXISTS`)
- `LIMIT` in subquery (other than `EXISTS`)
- `OFFSET` in subquery
- Subquery without a `FROM` clause

Limited Use of `STABLE` and `VOLATILE` Functions

To prevent data from becoming out-of-sync across the segments in Greenplum Database, any function classified as `STABLE` or `VOLATILE` cannot be executed at the segment database level if it contains SQL or modifies the database in any way. See [CREATE FUNCTION](#) for more information.

See Also

[EXPLAIN](#)

SELECT INTO

Defines a new table from the results of a query.

Synopsis

```
SELECT [ALL | DISTINCT [ON ( expression [, ...] )]]
      * | expression [AS output_name] [, ...]
INTO [TEMPORARY | TEMP] [TABLE] new_table
     [FROM from_item [, ...]]
     [WHERE condition]
     [GROUP BY expression [, ...]]
     [HAVING condition [, ...]]
     [{UNION | INTERSECT | EXCEPT} [ALL] select]
     [ORDER BY expression [ASC | DESC | USING operator] [, ...]]
     [LIMIT {count | ALL}]
     [OFFSET start]
     [FOR {UPDATE | SHARE} [OF table_name [, ...]] [NOWAIT]
     [...]]
```

Description

`SELECT INTO` creates a new table and fills it with data computed by a query. The data is not returned to the client, as it is with a normal `SELECT`. The new table's columns have the names and data types associated with the output columns of the `SELECT`.

Parameters

The majority of parameters for `SELECT INTO` are the same as [SELECT](#).

TEMPORARY TEMP

If specified, the table is created as a temporary table.

new_table

The name (optionally schema-qualified) of the table to be created.

Examples

Create a new table `films_recent` consisting of only recent entries from the table `films`:

```
SELECT * INTO films_recent FROM films WHERE date_prod >=
'2006-01-01';
```

Compatibility

The SQL standard uses `SELECT INTO` to represent selecting values into scalar variables of a host program, rather than creating a new table. The Greenplum Database usage of `SELECT INTO` to represent table creation is historical. It is best to use `CREATE TABLE AS` for this purpose in new applications.

See Also

`SELECT`, `CREATE TABLE AS`

SET

Changes the value of a Greenplum Database configuration parameter.

Synopsis

```
SET [SESSION | LOCAL] configuration_parameter {TO | =} value |
'value' | DEFAULT}
```

```
SET [SESSION | LOCAL] TIME ZONE {timezone | LOCAL | DEFAULT}
```

Description

The `SET` command changes server configuration parameters. Any user-settable *global* or *master-only* Greenplum Database configuration parameter can be changed on-the-fly with `SET` (see “[Server Configuration Parameters](#)” on page 755 for details). `SET` only affects the value used by the current session.

If `SET` or `SET SESSION` is issued within a transaction that is later aborted, the effects of the `SET` command disappear when the transaction is rolled back. Once the surrounding transaction is committed, the effects will persist until the end of the session, unless overridden by another `SET`.

The effects of `SET LOCAL` last only till the end of the current transaction, whether committed or not. A special case is `SET` followed by `SET LOCAL` within a single transaction: the `SET LOCAL` value will be seen until the end of the transaction, but afterwards (if the transaction is committed) the `SET` value will take effect.

Parameters

SESSION

Specifies that the command takes effect for the current session. This is the default.

LOCAL

Specifies that the command takes effect for only the current transaction. After `COMMIT` or `ROLLBACK`, the session-level setting takes effect again. Note that `SET LOCAL` will appear to have no effect if it is executed outside of a transaction.

configuration_parameter

The name of a Greenplum Database configuration parameter. Only parameters classified as user-settable, global or master-only can be changed with `SET`. See “[Server Configuration Parameters](#)” on page 755 for details.

value

New value of parameter. Values can be specified as string constants, identifiers, numbers, or comma-separated lists of these. `DEFAULT` can be used to specify resetting the parameter to its default value.

TIME ZONE

`SET TIME ZONE` *value* is an alias for `SET timezone TO value`. The syntax `SET TIME ZONE` allows special syntax for the time zone specification. Here are examples of valid values:

```
'PST8PDT'
```

```
'Europe/Rome'
```

```
-7 (time zone 7 hours west from UTC)
```

```
INTERVAL '-08:00' HOUR TO MINUTE (time zone 8 hours west from UTC).
```

**LOCAL
DEFAULT**

Set the time zone to your local time zone (the one that the server's operating system defaults to). See the [Time zone section of the PostgreSQL documentation](#) for more information about time zones in Greenplum Database.

Examples

Set the schema search path:

```
SET search_path TO my_schema, public;
```

Set the style of date to traditional POSTGRES with “day before month” input convention:

```
SET datestyle TO postgres, dmy;
```

Set the time zone for San Mateo, California:

```
SET TIME ZONE 'PST8PDT';
```

Set the time zone for Italy:

```
SET TIME ZONE 'Europe/Rome';
```

Compatibility

`SET TIME ZONE` extends syntax defined in the SQL standard. The standard allows only numeric time zone offsets while Greenplum Database allows more flexible time-zone specifications. All other `SET` features are Greenplum Database extensions.

See Also

[RESET](#), [SHOW](#)

SET ROLE

Sets the current role identifier of the current session.

Synopsis

```
SET [SESSION | LOCAL] ROLE rolename
SET [SESSION | LOCAL] ROLE NONE
RESET ROLE
```

Description

This command sets the current role identifier of the current SQL-session context to be *rolename*. The role name may be written as either an identifier or a string literal. After `SET ROLE`, permissions checking for SQL commands is carried out as though the named role were the one that had logged in originally.

The specified *rolename* must be a role that the current session user is a member of. If the session user is a superuser, any role can be selected.

The `NONE` and `RESET` forms reset the current role identifier to be the current session role identifier. These forms may be executed by any user.

Parameters

SESSION

Specifies that the command takes effect for the current session. This is the default.

LOCAL

Specifies that the command takes effect for only the current transaction. After `COMMIT` or `ROLLBACK`, the session-level setting takes effect again. Note that `SET LOCAL` will appear to have no effect if it is executed outside of a transaction.

rolename

The name of a role to use for permissions checking in this session.

NONE

RESET

Reset the current role identifier to be the current session role identifier (that of the role used to log in).

Notes

Using this command, it is possible to either add privileges or restrict privileges. If the session user role has the `INHERITS` attribute, then it automatically has all the privileges of every role that it could `SET ROLE` to; in this case `SET ROLE` effectively drops all the privileges assigned directly to the session user and to the other roles it is a member of, leaving only the privileges available to the named role. On the other

hand, if the session user role has the `NOINHERITS` attribute, `SET ROLE` drops the privileges assigned directly to the session user and instead acquires the privileges available to the named role.

In particular, when a superuser chooses to `SET ROLE` to a non-superuser role, she loses her superuser privileges.

`SET ROLE` has effects comparable to `SET SESSION AUTHORIZATION`, but the privilege checks involved are quite different. Also, `SET SESSION AUTHORIZATION` determines which roles are allowable for later `SET ROLE` commands, whereas changing roles with `SET ROLE` does not change the set of roles allowed to a later `SET ROLE`.

Examples

```
SELECT SESSION_USER, CURRENT_USER;
 session_user | current_user
-----+-----
 peter       | peter
```

```
SET ROLE 'paul';
```

```
SELECT SESSION_USER, CURRENT_USER;
 session_user | current_user
-----+-----
 peter       | paul
```

Compatibility

Greenplum Database allows identifier syntax (*rolename*), while the SQL standard requires the role name to be written as a string literal. SQL does not allow this command during a transaction; Greenplum Database does not make this restriction. The `SESSION` and `LOCAL` modifiers are a Greenplum Database extension, as is the `RESET` syntax.

See Also

[SET SESSION AUTHORIZATION](#)

SET SESSION AUTHORIZATION

Sets the session role identifier and the current role identifier of the current session.

Synopsis

```
SET [SESSION | LOCAL] SESSION AUTHORIZATION rolename
SET [SESSION | LOCAL] SESSION AUTHORIZATION DEFAULT
RESET SESSION AUTHORIZATION
```

Description

This command sets the session role identifier and the current role identifier of the current SQL-session context to be *rolename*. The role name may be written as either an identifier or a string literal. Using this command, it is possible, for example, to temporarily become an unprivileged user and later switch back to being a superuser.

The session role identifier is initially set to be the (possibly authenticated) role name provided by the client. The current role identifier is normally equal to the session user identifier, but may change temporarily in the context of `setuid` functions and similar mechanisms; it can also be changed by `SET ROLE`. The current user identifier is relevant for permission checking.

The session user identifier may be changed only if the initial session user (the authenticated user) had the superuser privilege. Otherwise, the command is accepted only if it specifies the authenticated user name.

The `DEFAULT` and `RESET` forms reset the session and current user identifiers to be the originally authenticated user name. These forms may be executed by any user.

Parameters

SESSION

Specifies that the command takes effect for the current session. This is the default.

LOCAL

Specifies that the command takes effect for only the current transaction. After `COMMIT` or `ROLLBACK`, the session-level setting takes effect again. Note that `SET LOCAL` will appear to have no effect if it is executed outside of a transaction.

rolename

The name of the role to assume.

NONE

RESET

Reset the session and current role identifiers to be that of the role used to log in.

Examples

```
SELECT SESSION_USER, CURRENT_USER;
 session_user | current_user
-----+-----
peter        | peter
```

```
SET SESSION AUTHORIZATION 'paul';
```

```
SELECT SESSION_USER, CURRENT_USER;
 session_user | current_user
-----+-----
paul         | paul
```

Compatibility

The SQL standard allows some other expressions to appear in place of the literal *rolename*, but these options are not important in practice. Greenplum Database allows identifier syntax (“*rolename*”), which SQL does not. SQL does not allow this command during a transaction; Greenplum Database does not make this restriction. The `SESSION` and `LOCAL` modifiers are a Greenplum Database extension, as is the `RESET` syntax.

See Also

[SET ROLE](#)

SET TRANSACTION

Sets the characteristics of the current transaction.

Synopsis

```
SET TRANSACTION transaction_mode [, ...]
```

```
SET SESSION CHARACTERISTICS AS TRANSACTION transaction_mode [, ...]
```

where *transaction_mode* is one of:

```
ISOLATION LEVEL {SERIALIZABLE | REPEATABLE READ | READ
COMMITTED | READ UNCOMMITTED}
READ WRITE | READ ONLY
```

Description

The `SET TRANSACTION` command sets the characteristics of the current transaction. It has no effect on any subsequent transactions.

The available transaction characteristics are the transaction isolation level and the transaction access mode (read/write or read-only).

The isolation level of a transaction determines what data the transaction can see when other transactions are running concurrently.

- **READ COMMITTED** — A statement can only see rows committed before it began. This is the default.
- **SERIALIZABLE** — All statements of the current transaction can only see rows committed before the first query or data-modification statement was executed in this transaction.

The SQL standard defines two additional levels, `READ UNCOMMITTED` and `REPEATABLE READ`. In Greenplum Database `READ UNCOMMITTED` is treated as `READ COMMITTED`, while `REPEATABLE READ` is treated as `SERIALIZABLE`.

The transaction isolation level cannot be changed after the first query or data-modification statement (`SELECT`, `INSERT`, `DELETE`, `UPDATE`, `FETCH`, or `COPY`) of a transaction has been executed.

The transaction access mode determines whether the transaction is read/write or read-only. Read/write is the default. When a transaction is read-only, the following SQL commands are disallowed: `INSERT`, `UPDATE`, `DELETE`, and `COPY FROM` if the table they would write to is not a temporary table; all `CREATE`, `ALTER`, and `DROP` commands; `GRANT`, `REVOKE`, `TRUNCATE`; and `EXPLAIN ANALYZE` and `EXECUTE` if the command they would execute is among those listed. This is a high-level notion of read-only that does not prevent all writes to disk.

Parameters

SESSION CHARACTERISTICS

Sets the default transaction characteristics for subsequent transactions of a session.

SERIALIZABLE
REPEATABLE READ
READ COMMITTED
READ UNCOMMITTED

The SQL standard defines four transaction isolation levels: `READ COMMITTED`, `READ UNCOMMITTED`, `SERIALIZABLE`, and `REPEATABLE READ`. The default behavior is that a statement can only see rows committed before it began (`READ COMMITTED`). In Greenplum Database `READ UNCOMMITTED` is treated the same as `READ COMMITTED`. `SERIALIZABLE` is supported the same as `REPEATABLE READ` wherein all statements of the current transaction can only see rows committed before the first statement was executed in the transaction. `SERIALIZABLE` is the strictest transaction isolation. This level emulates serial transaction execution, as if transactions had been executed one after another, serially, rather than concurrently. Applications using this level must be prepared to retry transactions due to serialization failures.

READ WRITE
READ ONLY

Determines whether the transaction is read/write or read-only. Read/write is the default. When a transaction is read-only, the following SQL commands are disallowed: `INSERT`, `UPDATE`, `DELETE`, and `COPY FROM` if the table they would write to is not a temporary table; all `CREATE`, `ALTER`, and `DROP` commands; `GRANT`, `REVOKE`, `TRUNCATE`; and `EXPLAIN ANALYZE` and `EXECUTE` if the command they would execute is among those listed.

Notes

If `SET TRANSACTION` is executed without a prior `START TRANSACTION` or `BEGIN`, it will appear to have no effect.

It is possible to dispense with `SET TRANSACTION` by instead specifying the desired `transaction_modes` in `BEGIN` or `START TRANSACTION`.

The session default transaction modes can also be set by setting the configuration parameters `default_transaction_isolation` and `default_transaction_read_only`.

Examples

Set the transaction isolation level for the current transaction:

```
BEGIN;
SET TRANSACTION SERIALIZABLE;
```

Compatibility

Both commands are defined in the SQL standard. `SERIALIZABLE` is the default transaction isolation level in the standard. In Greenplum Database the default is `READ COMMITTED`. Because of lack of predicate locking, the `SERIALIZABLE` level is not truly serializable. Essentially, a predicate-locking system prevents phantom reads by restricting what is written, whereas a multi-version concurrency control model (MVCC) as used in Greenplum Database prevents them by restricting what is read.

In the SQL standard, there is one other transaction characteristic that can be set with these commands: the size of the diagnostics area. This concept is specific to embedded SQL, and therefore is not implemented in the Greenplum Database server.

The SQL standard requires commas between successive *transaction_modes*, but for historical reasons Greenplum Database allows the commas to be omitted.

See Also

[BEGIN](#), [LOCK](#)

SHOW

Shows the value of a system configuration parameter.

Synopsis

```
SHOW configuration_parameter
```

```
SHOW ALL
```

Description

SHOW will display the current settings of Greenplum Database system configuration parameters. These parameters can be set using the SET statement, or by editing the `postgresql.conf` configuration file of the Greenplum Database master. Note that some parameters viewable by SHOW are read-only — their values can be viewed but not set. See “[Server Configuration Parameters](#)” on page 755 for details.

Parameters

configuration_parameter

The name of a system configuration parameter. See “[Server Configuration Parameters](#)” on page 755.

ALL

Shows the current value of all configuration parameters.

Examples

Show the current setting of the parameter *search_path*:

```
SHOW search_path;
```

Show the current setting of all parameters:

```
SHOW ALL;
```

Compatibility

SHOW is a Greenplum Database extension.

See Also

[SET](#), [RESET](#)

START TRANSACTION

Starts a transaction block.

Synopsis

```
START TRANSACTION [SERIALIZABLE | REPEATABLE READ | READ
COMMITTED | READ UNCOMMITTED] [READ WRITE | READ ONLY]
```

Description

`START TRANSACTION` begins a new transaction block. If the isolation level or read/write mode is specified, the new transaction has those characteristics, as if `SET TRANSACTION` was executed. This is the same as the `BEGIN` command.

Parameters

SERIALIZABLE
REPEATABLE READ
READ COMMITTED
READ UNCOMMITTED

The SQL standard defines four transaction isolation levels: `READ COMMITTED`, `READ UNCOMMITTED`, `SERIALIZABLE`, and `REPEATABLE READ`. The default behavior is that a statement can only see rows committed before it began (`READ COMMITTED`). In Greenplum Database `READ UNCOMMITTED` is treated the same as `READ COMMITTED`. `SERIALIZABLE` is supported the same as `REPEATABLE READ` wherein all statements of the current transaction can only see rows committed before the first statement was executed in the transaction. `SERIALIZABLE` is the strictest transaction isolation. This level emulates serial transaction execution, as if transactions had been executed one after another, serially, rather than concurrently. Applications using this level must be prepared to retry transactions due to serialization failures.

READ WRITE
READ ONLY

Determines whether the transaction is read/write or read-only. Read/write is the default. When a transaction is read-only, the following SQL commands are disallowed: `INSERT`, `UPDATE`, `DELETE`, and `COPY FROM` if the table they would write to is not a temporary table; all `CREATE`, `ALTER`, and `DROP` commands; `GRANT`, `REVOKE`, `TRUNCATE`; and `EXPLAIN ANALYZE` and `EXECUTE` if the command they would execute is among those listed.

Examples

To begin a transaction block:

```
START TRANSACTION;
```

Compatibility

In the standard, it is not necessary to issue `START TRANSACTION` to start a transaction block: any SQL command implicitly begins a block. Greenplum Database behavior can be seen as implicitly issuing a `COMMIT` after each command that does not follow `START TRANSACTION` (or `BEGIN`), and it is therefore often called ‘autocommit’. Other relational database systems may offer an autocommit feature as a convenience.

The SQL standard requires commas between successive *transaction_modes*, but for historical reasons Greenplum Database allows the commas to be omitted.

See also the compatibility section of [SET TRANSACTION](#).

See Also

[BEGIN](#), [SET TRANSACTION](#)

TRUNCATE

Empties a table of all rows.

Synopsis

```
TRUNCATE [TABLE] name [, ...] [CASCADE | RESTRICT]
```

Description

TRUNCATE quickly removes all rows from a table or set of tables. It has the same effect as an unqualified DELETE on each table, but since it does not actually scan the tables it is faster. This is most useful on large tables.

Parameters

name

The name (optionally schema-qualified) of a table to be truncated.

CASCADE

Since this key word applies to foreign key references (which are not supported in Greenplum Database) it has no effect.

RESTRICT

Since this key word applies to foreign key references (which are not supported in Greenplum Database) it has no effect.

Notes

Only the owner of a table may TRUNCATE it.

TRUNCATE will not run any user-defined ON DELETE triggers that might exist for the tables.

TRUNCATE will not truncate any tables that inherit from the named table. Only the named table is truncated, not its child tables.

Examples

Empty the table films:

```
TRUNCATE films;
```

Compatibility

There is no TRUNCATE command in the SQL standard.

See Also

DELETE, DROP TABLE

UPDATE

Updates rows of a table.

Synopsis

```
UPDATE [ONLY] table [[AS] alias]
  SET {column = {expression | DEFAULT} |
      (column [, ...]) = ({expression | DEFAULT} [, ...])} [, ...]
  [FROM fromlist]
  [WHERE condition]
```

Description

UPDATE changes the values of the specified columns in all rows that satisfy the condition. Only the columns to be modified need be mentioned in the SET clause; columns not explicitly modified retain their previous values.

By default, UPDATE will update rows in the specified table and all its subtables. If you wish to only update the specific table mentioned, you must use the ONLY clause.

There are two ways to modify a table using information contained in other tables in the database: using sub-selects, or specifying additional tables in the FROM clause. Which technique is more appropriate depends on the specific circumstances. Note that if you do join two or more tables in an UPDATE command, those tables must have the same Greenplum distribution key column(s) and also be joined on the distribution key column(s).

You must have the UPDATE privilege on the table to update it, as well as the SELECT privilege to any table whose values are read in the expressions or condition.

Outputs

On successful completion, an UPDATE command returns a command tag of the form:

```
UPDATE count
```

Where *count* is the number of rows updated. If *count* is 0, no rows matched the condition (this is not considered an error).

Parameters

ONLY

If specified, update rows from the named table only. When not specified, any tables inheriting from the named table are also processed.

table

The name (optionally schema-qualified) of an existing table.

alias

A substitute name for the target table. When an alias is provided, it completely hides the actual name of the table. For example, given `UPDATE foo AS f`, the remainder of the `UPDATE` statement must refer to this table as `f` not `foo`.

column

The name of a column in table. The column name can be qualified with a subfield name or array subscript, if needed. Do not include the table's name in the specification of a target column.

expression

An expression to assign to the column. The expression may use the old values of this and other columns in the table.

DEFAULT

Set the column to its default value (which will be `NULL` if no specific default expression has been assigned to it).

fromlist

A list of table expressions, allowing columns from other tables to appear in the `WHERE` condition and the update expressions. This is similar to the list of tables that can be specified in the `FROM` clause of a `SELECT` statement. Note that the target table must not appear in the *fromlist*, unless you intend a self-join (in which case it must appear with an *alias* in the *fromlist*).

condition

An expression that returns a value of type boolean. Only rows for which this expression returns true will be updated.

output_expression

An expression to be computed and returned by the `UPDATE` command after each row is updated. The expression may use any column names of the table or table(s) listed in `FROM`. Write `*` to return all columns.

output_name

A name to use for a returned column.

Notes

`SET` is not allowed on the Greenplum distribution key columns of a table.

You cannot use `STABLE` or `VOLATILE` functions in an `UPDATE` statement if mirrors are enabled. This can potentially cause the primary segment and its mirror to become out-of-sync because the command is run first on the primary and then a second time on the mirror in the current Greenplum Database implementation.

When a `FROM` clause is present, what essentially happens is that the target table is joined to the tables mentioned in the from list, and each output row of the join represents an update operation for the target table. When using `FROM` you should

ensure that the join produces at most one output row for each row to be modified. In other words, a target row should not join to more than one row from the other table(s). If it does, then only one of the join rows will be used to update the target row, but which one will be used is not readily predictable.

Because of this indeterminacy, referencing other tables only within sub-selects is safer, though often harder to read and slower than using a join.

Note also that in Greenplum Database, any tables that are joined in an update operation must have the *same Greenplum distribution key* and be joined on the column(s) that make up the distribution key (equijoin).

Examples

Change the word *Drama* to *Dramatic* in the column *kind* of the table *films*:

```
UPDATE films SET kind = 'Dramatic' WHERE kind = 'Drama';
```

Adjust temperature entries and reset precipitation to its default value in one row of the table *weather*:

```
UPDATE weather SET temp_lo = temp_lo+1, temp_hi =
temp_lo+15, prcp = DEFAULT
WHERE city = 'San Francisco' AND date = '2006-07-03';
```

Use the alternative column-list syntax to do the same update:

```
UPDATE weather SET (temp_lo, temp_hi, prcp) = (temp_lo+1,
temp_lo+15, DEFAULT)
WHERE city = 'San Francisco' AND date = '2006-07-03';
```

Increment the sales count of the salesperson who manages the account for Acme Corporation, using the `FROM` clause syntax (assuming both tables being joined are distributed in Greenplum Database on the *id* column):

```
UPDATE employees SET sales_count = sales_count + 1 FROM
accounts
WHERE accounts.name = 'Acme Corporation'
AND employees.id = accounts.id;
```

Perform the same operation, using a sub-select in the `WHERE` clause:

```
UPDATE employees SET sales_count = sales_count + 1 WHERE id =
(SELECT id FROM accounts WHERE name = 'Acme Corporation');
```

Attempt to insert a new stock item along with the quantity of stock. If the item already exists, instead update the stock count of the existing item. To do this without failing the entire transaction, use savepoints.

```
BEGIN;
-- other operations
SAVEPOINT sp1;
INSERT INTO wines VALUES('Chateau Lafite 2003', '24');
-- Assume the above fails because of a unique key violation,
-- so now we issue these commands:
```

```

ROLLBACK TO sp1;
UPDATE wines SET stock = stock + 24 WHERE winename = 'Chateau
Lafite 2003';
-- continue with other operations, and eventually
COMMIT;

```

Compatibility

This command conforms to the SQL standard, except that the `FROM` clause is a Greenplum Database extension.

According to the standard, the column-list syntax should allow a list of columns to be assigned from a single row-valued expression, such as a sub-select:

```

UPDATE accounts SET (contact_last_name, contact_first_name) =
    (SELECT last_name, first_name FROM salesmen
     WHERE salesmen.id = accounts.sales_id);

```

This is not currently implemented — the source must be a list of independent expressions.

Some other database systems offer a `FROM` option in which the target table is supposed to be listed again within `FROM`. That is not how Greenplum Database interprets `FROM`. Be careful when porting applications that use this extension.

See Also

[DELETE](#), [SELECT](#), [INSERT](#)

VACUUM

Garbage-collects and optionally analyzes a database.

Synopsis

```
VACUUM [FULL] [FREEZE] [VERBOSE] [table]
VACUUM [FULL] [FREEZE] [VERBOSE] ANALYZE
      [table [(column [, ...] )]]
```

Description

`VACUUM` reclaims storage occupied by deleted tuples. In normal Greenplum Database operation, tuples that are deleted or obsoleted by an update are not physically removed from their table; they remain present on disk until a `VACUUM` is done. Therefore it is necessary to do `VACUUM` periodically, especially on frequently-updated tables.

With no parameter, `VACUUM` processes every table in the current database. With a parameter, `VACUUM` processes only that table.

`VACUUM ANALYZE` performs a `VACUUM` and then an `ANALYZE` for each selected table. This is a handy combination form for routine maintenance scripts. See [ANALYZE](#) for more details about its processing.

Plain `VACUUM` (without `FULL`) simply reclaims space and makes it available for re-use. This form of the command can operate in parallel with normal reading and writing of the table, as an exclusive lock is not obtained. `VACUUM FULL` does more extensive processing, including moving of tuples across blocks to try to compact the table to the minimum number of disk blocks. This form is much slower and requires an exclusive lock on each table while it is being processed.

Outputs

When `VERBOSE` is specified, `VACUUM` emits progress messages to indicate which table is currently being processed. Various statistics about the tables are printed as well.

Parameters

FULL

Selects a full vacuum, which may reclaim more space, but takes much longer and exclusively locks the table.

Warning: A `VACUUM FULL` is not recommended in Greenplum Database. See the “Notes” section.

FREEZE

Specifying `FREEZE` is equivalent to performing `VACUUM` with the `vacuum_freeze_min_age` server configuration parameter set to zero. The `FREEZE` option is deprecated and will be removed in a future release. Set the parameter in the master `postgresql.conf` file instead.

VERBOSE

Prints a detailed vacuum activity report for each table.

ANALYZE

Updates statistics used by the planner to determine the most efficient way to execute a query.

table

The name (optionally schema-qualified) of a specific table to vacuum. Defaults to all tables in the current database.

column

The name of a specific column to analyze. Defaults to all columns.

Notes

`VACUUM` cannot be executed inside a transaction block.

Greenplum recommends that active production databases be vacuumed frequently (at least nightly), in order to remove expired rows. After adding or deleting a large number of rows, it may be a good idea to issue a `VACUUM ANALYZE` command for the affected table. This will update the system catalogs with the results of all recent changes, and allow the Greenplum query planner to make better choices in planning queries.

`VACUUM` causes a substantial increase in I/O traffic, which can cause poor performance for other active sessions. Therefore, it is advisable to vacuum the database at low usage times.

Regular PostgreSQL has a separate optional server process called the *autovacuum daemon*, whose purpose is to automate the execution of `VACUUM` and `ANALYZE` commands. This feature is currently disabled in Greenplum Database.

Expired rows are held in what is called the *free space map*. The free space map must be sized large enough to cover the dead rows of all tables in your database. If not sized large enough, space occupied by dead rows that overflow the free space map cannot be reclaimed by a regular `VACUUM` command.

A `VACUUM FULL` will reclaim all expired row space, but is a very expensive operation and may take an unacceptably long time to finish on large, distributed Greenplum Database tables. If you do get into a situation where the free space map has overflowed, it may be more timely to recreate the table with a `CREATE TABLE AS` statement and drop the old table. A `VACUUM FULL` is not recommended in Greenplum Database.

It is best to size the free space map appropriately. The free space map is configured with the following server configuration parameters:

```
max_fsm_pages
max_fsm_relations
```

See “[About Concurrency Control in Greenplum Database](#)” on page 125 for more information.

Examples

Vacuum all tables in the current database:

```
VACUUM;
```

Vacuum a specific table only:

```
VACUUM mytable;
```

Vacuum all tables in the current database and collect statistics for the query planner:

```
VACUUM ANALYZE;
```

Compatibility

There is no `VACUUM` statement in the SQL standard.

See Also

[ANALYZE](#)

VALUES

Computes a set of rows.

Synopsis

```
VALUES ( expression [, ...] ) [, ...]
[ORDER BY sort_expression [ASC | DESC | USING operator] [, ...]]
[LIMIT {count | ALL}] [OFFSET start]
```

Description

VALUES computes a row value or set of row values specified by value expressions. It is most commonly used to generate a ‘constant table’ within a larger command, but it can be used on its own.

When more than one row is specified, all the rows must have the same number of elements. The data types of the resulting table’s columns are determined by combining the explicit or inferred types of the expressions appearing in that column, using the same rules as for UNION.

Within larger commands, VALUES is syntactically allowed anywhere that SELECT is. Because it is treated like a SELECT by the grammar, it is possible to use the ORDER BY, LIMIT, and OFFSET clauses with a VALUES command.

Parameters

expression

A constant or expression to compute and insert at the indicated place in the resulting table (set of rows). In a VALUES list appearing at the top level of an INSERT, an expression can be replaced by DEFAULT to indicate that the destination column’s default value should be inserted. DEFAULT cannot be used when VALUES appears in other contexts.

sort_expression

An expression or integer constant indicating how to sort the result rows. This expression may refer to the columns of the VALUES result as *column1*, *column2*, etc. For more details see [“The ORDER BY Clause”](#) on page 532.

operator

A sorting operator. For details see [“The ORDER BY Clause”](#) on page 532.

LIMIT *count*

OFFSET *start*

The maximum number of rows to return. For details see [“The LIMIT Clause”](#) on page 533.

Notes

VALUES lists with very large numbers of rows should be avoided, as you may encounter out-of-memory failures or poor performance. VALUES appearing within INSERT is a special case (because the desired column types are known from the INSERT's target table, and need not be inferred by scanning the VALUES list), so it can handle larger lists than are practical in other contexts.

Examples

A bare VALUES command:

```
VALUES (1, 'one'), (2, 'two'), (3, 'three');
```

This will return a table of two columns and three rows. It is effectively equivalent to:

```
SELECT 1 AS column1, 'one' AS column2
UNION ALL
SELECT 2, 'two'
UNION ALL
SELECT 3, 'three';
```

More usually, VALUES is used within a larger SQL command. The most common use is in INSERT:

```
INSERT INTO films (code, title, did, date_prod, kind)
VALUES ('T_601', 'Yojimbo', 106, '1961-06-16', 'Drama');
```

In the context of INSERT, entries of a VALUES list can be DEFAULT to indicate that the column default should be used here instead of specifying a value:

```
INSERT INTO films VALUES
('UA502', 'Bananas', 105, DEFAULT, 'Comedy', '82
minutes'),
('T_601', 'Yojimbo', 106, DEFAULT, 'Drama', DEFAULT);
```

VALUES can also be used where a sub-SELECT might be written, for example in a FROM clause:

```
SELECT f.* FROM films f, (VALUES('MGM', 'Horror'), ('UA',
'Sci-Fi')) AS t (studio, kind) WHERE f.studio = t.studio AND
f.kind = t.kind;
UPDATE employees SET salary = salary * v.increase FROM
(VALUES(1, 200000, 1.2), (2, 400000, 1.4)) AS v (depno,
target, increase) WHERE employees.depno = v.depno AND
employees.sales >= v.target;
```

Note that an AS clause is required when VALUES is used in a FROM clause, just as is true for SELECT. It is not required that the AS clause specify names for all the columns, but it is good practice to do so. The default column names for VALUES are column1, column2, etc. in Greenplum Database, but these names might be different in other database systems.

When `VALUES` is used in `INSERT`, the values are all automatically coerced to the data type of the corresponding destination column. When it is used in other contexts, it may be necessary to specify the correct data type. If the entries are all quoted literal constants, coercing the first is sufficient to determine the assumed type for all:

```
SELECT * FROM machines WHERE ip_address IN
  (VALUES ('192.168.0.1'::inet), ('192.168.0.10'),
   ('192.168.1.43'));
```

Note: For simple `IN` tests, it is better to rely on the list-of-scalars form of `IN` than to write a `VALUES` query as shown above. The list of scalars method requires less writing and is often more efficient.

Compatibility

`VALUES` conforms to the SQL standard, except that `LIMIT` and `OFFSET` are Greenplum Database extensions.

See Also

[INSERT](#), [SELECT](#)

B. Management Utility Reference

This appendix provides references for the command-line management utilities provided with Greenplum Database. Greenplum Database utilizes the standard PostgreSQL client and server programs, and also has additional management utilities to facilitate the administration of a distributed Greenplum Database DBMS.

The following Greenplum Database management utilities are located in `$GPHOME/bin`:

- `gp_dump`
- `gp_restore`
- `gpactivatestandby`
- `gpaddmirrors`
- `gpchecknet`
- `gpcheckos`
- `gpcheckperf`
- `gpcrondump`
- `gpdbrestore`
- `gpdeletesystem`
- `gpdetective`
- `gpexpand`
- `gpfdist`
- `gpinitstandby`
- `gpinitssystem`
- `gpload`
- `gplogfilter`
- `gpmapproduce`
- `gpmigrator`
- `gpbuildsystem`
- `gprecoverseg`
- `gpsizecalc`
- `gpskew`
- `gpscp`
- `gpssh`
- `gpssh-exkeys`
- `gpstart`
- `gpstate`
- `gpstop`

Backend Server Programs

The following server programs are also located in `$GPHOME/bin` of your Greenplum Database installation. These are the standard PostgreSQL server programs, which have been modified to handle the parallelism and distribution of a Greenplum Database system. Keep in mind that Greenplum Database is essentially several PostgreSQL database instances working together as a single DBMS, so Greenplum Database relies on PostgreSQL for its underlying functionality. Users and administrators do not access these programs directly, but do so through the Greenplum Database management tools and utilities.

Table B.1 Greenplum Database Backend Server Programs

Program Name	Description	Use Instead
<code>initdb</code>	This program is called by <code>gpinitssystem</code> when initializing a Greenplum Database array. It is used internally to create the individual segment instances and the master instance.	<code>gpinitssystem</code>
<code>ipcclean</code>	Not used in Greenplum Database	N/A
<code>gpsyncmaster</code>	This is the Greenplum program that starts the <code>gpsyncagent</code> process on the standby master host. Administrators do not call this program directly, but do so through the management scripts that initialize and/or activate a standby master for a Greenplum Database system. This process is responsible for keeping the standby master up to date with the primary master via a transaction log replication process.	<code>gpinitstandby</code> , <code>gpactivatestandby</code>
<code>pg_controldata</code>	Not used in Greenplum Database	<code>gpstate</code>
<code>pg_ctl</code>	This program is called by <code>gpstart</code> and <code>gpstop</code> when starting or stopping a Greenplum Database array. It is used internally to stop and start the individual segment instances and the master instance in parallel and with the correct options.	<code>gpstart</code> , <code>gpstop</code>
<code>pg_resetxlog</code>	Not used in Greenplum Database	N/A
<code>postgres</code>	The <code>postgres</code> executable is the actual PostgreSQL server process that processes queries.	The main <code>postgres</code> process (<code>postmaster</code>) creates other <code>postgres</code> processes as needed to handle client connections.
<code>postmaster</code>	<code>postmaster</code> starts the <code>postgres</code> database server listener process that accepts client connections. In Greenplum Database, a <code>postgres</code> database listener process runs on the Greenplum Master Instance and on each Segment Instance.	In Greenplum Database, you use <code>gpstart</code> and <code>gpstop</code> to start all <code>postmasters</code> (<code>postgres</code> processes) in the system at once in the correct order and with the correct options.

Management Utility Summary

gp_dump

Writes out a database to SQL script files, which can then be used to restore the database using `gp_restore`.

```
gp_dump [-a | -s] [-c] [-d] [-D] [-n schema] [-o] [-O] [-t table_name] [-x] [-h
hostname] [-p port] [-U username] [-W] [-i] [-v] [--gp-c]
[--gp-d=backup_directory] [--gp-r=reportfile] [--gp-s=a|p|i [dbid, ...]]
database_name
```

```
gp_dump -? | --help
```

```
gp_dump --version
```

```
-a | --data-only
```

```
-s | --schema-only
```

```
-c | --clean
```

```
-d | --inserts
```

```
-D | --column-inserts
```

```
-n schema | --schema=schema
```

```
-o | --oids
```

```
-O | --no-owner
```

```
-t table | --table=table
```

```
-x | --no-privileges | --no-acl
```

```
-h hostname | --host=hostname
```

```
-p port | --port=port
```

```
-U username | --username=user
```

```
-W (force password prompt)
```

```
-i | --ignore-version
```

```
-v | --verbose
```

```
--gp-c (use gzip)
```

```
--gp-d=directoryname
```

```
--gp-r=reportfile
```

```
--gp-s=a | p | i [dbid, ...] (backup all, primaries, or certain segments)
```

```
database_name
```

```
-? | --help (help)
```

```
--version (show utility version)
```

gp_restore

Restores Greenplum databases that were backed up using `gp_dump`.

```
gp_restore --gp-k=timestamp_key -d database_name [-i] [-v] [-a | -s] [-c] [-h
hostname] [-p port] [-U username] [-W] [--gp-c] [--gp-i] [--gp-d=directoryname]
```

```
[--gp-r=reportfile] [--gp-l=a|p|i[dbid, ...]]
gp_restore -? | -h | --help
gp_restore --version
--gp-k=timestamp_key
-d database_name | --dbname=dbname
-i | --ignore-version
-v | --verbose
-a | --data-only
-c | --clean
-s | --schema-only
-h hostname | --host=hostname
-p port | --port=port
-U username | --username=username
-W (force password prompt)
--gp-c (use gunzip)
--gp-i (ignore errors)
--gp-d=directoryname
--gp-r=reportfile
--gp-l=a | p | i [dbid, ...] (restore all, primaries, or certain segments)
-? | -h | --help (help)
--version (show utility version)
```

gpactivatestandby

Activates a standby master host and makes it the active master for the Greenplum Database system.

```
gpactivatestandby -d standby_master_datadir [-c new_standby_master] [-f] [-a]
[-q] [-l logfile_directory]
gpactivatestandby -? | -h | --help
gpactivatestandby -v
-d standby_master_datadir
-c new_standby_master_hostname
-f (force activation)
-a (do not prompt)
-q (no screen output)
-l logfile_directory
-? | -h | --help (help)
-v (show utility version)
```

gpaddmirrors

Adds mirror segments to a Greenplum Database system that was initially configured without mirroring.

```
gpaddmirrors [-d master_data_directory] -p port_offset [-m datadir_config_file]
```

```

[-s] [-a] [-B parallel_processes] [-l logfile_directory] [-D]
gpaddmirrors [-d master_data_directory] -m datadir_config_file [-p port_offset]
[-s] [-a] [-B parallel_processes] [-l logfile_directory] [-D]
gpaddmirrors [-d master_data_directory] -i mirror_config_file [-s] [-a] [-B
parallel_processes] [-l logfile_directory] [-D]
gpaddmirrors -o output_sample_mirror_config
gpaddmirrors -?
gpaddmirrors --version
-a (do not prompt)
-B parallel_processes
-d master_data_directory
-D (debug)
-i mirror_config_file
-l logfile_directory
-m datadir_config_file
-o output_sample_mirror_config
-p port_offset
-s (spread mirrors)
--version (show utility version)
-? (help)

```

gpchecknet

Verifies the network performance between the specified hosts.

```

gpchecknet -d test_directory { -f host_file | -h hostname [-h hostname ...] }
[-r n|N] [-D] [-v | -V]
gpchecknet -?
gpchecknet --version
-d temp_test_directory
-f host_file
-h hostname
-r n|N
-D (do not filter for multi-homed hosts)
-v (verbose)
-V (very verbose)
-? (help)
--version

```

gpcheckos

Checks the operating system environment of the specified hosts to make sure the configuration is

compatible with Greenplum Database.

```
gpcheckos { [-m] -f host_file | -h hostname [-h hostname ...] } [-v]
```

```
gpcheckos -?
```

```
gpcheckos --version
```

```
-f host_file
```

```
-h hostname
```

```
-m (check master environment)
```

```
-v (verbose)
```

```
-? (help)
```

```
--version
```

gpcheckperf

Verifies the baseline hardware performance of the specified hosts.

```
gpcheckperf -d test_directory [-d test_directory ...] {-f host_file | -h host-  
name [-h hostname ...]} [-r ds] [-B block_size] [-S file_size] [-D] [-v | -V]
```

```
gpcheckperf -?
```

```
gpcheckperf --version
```

```
-d test_directory
```

```
-f host_file
```

```
-h hostname
```

```
-r ds
```

```
-B block_size
```

```
-S file_size
```

```
-D (display per-host results)
```

```
-v (verbose)
```

```
-V (very verbose)
```

```
-? (help)
```

```
--version
```

gpcrondump

A wrapper utility for `gp_dump`, which can be called directly or from a crontab entry.

```
gpcrondump -x database_name
```

```
[-s schema | -t schema.table | -T schema.table[,...]]
```

```
[-p | -m | -w dbid[,...]] [-u backup_directory]
```

```
[-R post_dump_script] [-c | -o] [-z] [-r] [-f free_space_percent] [-b] [-i] [-j |
```

```
-k] [-g] [-G] [-C] [-d master_data_directory]
```

```
[-B parallel_processes] [-a] [-q]
```

```
[-y reportfile] [-l logfile_directory] [-D]
```

```
{ [-E encoding] [--inserts | --column-inserts] [--oids] [--no-owner |
```

```

--use-set-session-authorization] [--no-privileges] }
gpcrondump -?
gpcrondump -v
-a (do not prompt)
-b (bypass disk space check)
-B parallel_processes
-c (clear old dump files first)
-C (clean old catalog dumps)
--column-inserts
-d master_data_directory
-D (debug)
-E encoding
-f free_space_percent
-g (copy config files)
-G (dump global objects)
-i (ignore parameter check)
--inserts
-j (vacuum before dump)
-k (vacuum after dump)
-l logfile_directory
-m (mirror segments only)
--no-owner
--no-privileges
-o (clear old dump files only)
--oids
-p (primary segments only)
-q (no screen output)
-r (no rollback on failure)
-R post_dump_script
-s schema_name
-t schema.table_name
-T schema.table_name
-u backup_directory
--use-set-session-authorization
-v (show utility version)
-w dbid[,...] (backup certain segments only)
-x database_name
-y reportfile
-z (no compression)
-? (help)

```

gpdbrestore

A wrapper utility around `gp_restore`. Restores a database from a set of dump files generated by `gpcrondump`.

```
gpdbrestore { -t timestamp_key [-L] | -b YYYYMMDD | -R hostname:path_to_dumpset
| -s database_name } [-T schema.table [,...]] [-e] [-G] [-B parallel_processes]
[-d master_data_directory] [-a] [-q] [-l logfile_directory] [-D]
```

```
gpdbrestore -?
```

```
gpdbrestore -v
```

```
-a (do not prompt)
```

```
-b YYYYMMDD
```

```
-B parallel_processes
```

```
-d master_data_directory
```

```
-D (debug)
```

```
-e (drop target database before restore)
```

```
-G (restore global objects)
```

```
-l logfile_directory
```

```
-L (list tablenames in backup set)
```

```
-q (no screen output)
```

```
-R hostname:path_to_dumpset
```

```
-s database_name
```

```
-t timestamp_key
```

```
-T schema.table_name
```

```
-v (show utility version)
```

```
-? (help)
```

gpdeletesystem

Deletes a Greenplum Database system that was initialized using `gpinitssystem`.

```
gpdeletesystem -d master_data_directory [-B parallel_processes] [-f] [-l
logfile_directory] [-D]
```

```
gpdeletesystem -?
```

```
gpdeletesystem -v
```

```
-d data_directory
```

```
-B parallel_processes
```

```
-f (force)
```

```
-l logfile_directory
```

```
-D (debug)
```

```
-? (help)
```

```
-v (show utility version)
```

gpdetective

Collects diagnostic information from a running Greenplum Database system.

```
gpdetective [-h master_hostname] [-p master_port] [-U gp_superuser] [-P password]
```

```

[--start_date number_of_days | YYYY-MM-DD ]
gpdetective -?
gpdetective -v
-h master_hostname
-p master_port
-P password
[--start_date number_of_days | YYYY-MM-DD]
-U gp_superuser
-v (show utility version)
-? (help)

```

gpexpand

Expands an existing Greenplum Database across new hosts in the array.

```

gpexpand [-f hosts_file] [-D database_name]
gpexpand -i input_file [-D database_name] [-B batch_size] [-V]
gpexpand [-d duration[hh][:mm[:ss]] | [-e 'YYYY-MM-DD hh:mm:ss']]
[-a] [-n parallel_processes] [-D database_name]
[--verbose | -v]
gpexpand -r | --rollback [-D database_name]
gpexpand -c [-D database_name]
gpexpand -? | -h | --help
gpexpand --version
-h (help)
-i input_file
-f hosts_file
-D database_name
-d [hh][:mm[:ss]]
-e YYYY-MM-DD hh:mm:ss
-B batch_size
-n parallel_processes
-a (no analyze)
-c (clean)
-r | --rollback (roll back)
-V (no vacuum)
-? | -h | --help (help)
--verbose | -v
--version

```

gpfdist

Serves external table files to Greenplum Database segments.

```

gpfdist [-d directory] [-p http_port] [-l log_file] [-t timeout] [-m max_length]

```

```
[-v | -V]
gpfdist -?
gpfdist --version
-d directory
-l log_file
-p http_port
-t timeout
-m max_length
-? (help)
-v (verbose)
-V (very verbose)
--version
```

gpinitstandby

Adds and/or initializes a standby master host for a Greenplum Database system.

```
gpinitstandby -s standby_hostname [-r] [-n] [-a] [-q] [-l logfile_directory] [-B
parallel_processes] [-D]
gpinitstandby -?
gpinitstandby -v
-s standby_hostname
-r (remove standby master)
-n (resynchronize)
-a (do not prompt)
-q (no screen output)
-l logfile_directory
-B parallel_processes
-D (debug)
-? (help)
-v (show utility version)
```

gpinitssystem

Initializes a Greenplum Database system by using configuration parameters specified in the `gp_init_config` file.

```
gpinitssystem -c gp_init_config_file
[-h host_file] [-r | -B parallel_processes] [-p postgresql_conf_param_file] {[-s
standby_master_host] [-i]} [-o] [-m max_connections] [-b shared_buffer_size] [-e
```

```

superuser_password] [-n locale] [-S] [-a] [-q] [-l logfile_directory] [-D]
gpinitssystem -?
gpinitssystem -v
-a (do not prompt)
-b shared_buffer_size
-B parallel_processes
-c gp_init_config_file
-D (debug)
-e superuser_password
-h host_file
-i (do not start synchronization)
-l logfile_directory
-m max_connections
-n locale
-o (create master only)
-p postgresql_conf_param_file
-q (no screen output)
-r (serial mode)
-s standby_master_host
-S (spread mirror configuration)
-v (show utility version)
-? (help)

```

gpload

Runs a load job as defined in a YAML formatted control file.

```

gpload -f control_file [-l log_file] [-h hostname] [-p port] [-U username] [-d

```

```

database] [--gpfdist_timeout number_seconds] [-W] [[-v | -V] [-q]] [-D]
gpload -?
gpload --version
-f control_file
--gpfdist_timeout number_seconds
-l log_file
-v (verbose mode)
-V (very verbose mode)
-q (no screen output)
-D (debug mode)
-? (show help)
--version
-d database
-h hostname
-p port
-U username
-W (force password prompt)

```

gplogfilter

Searches through Greenplum Database log files for specified entries.

```

gplogfilter [timestamp_options] [pattern_options] [output_options]

```

```

 
gplogfilter --help
gplogfilter --version
-b datetime | --begin=datetime
-e datetime | --end=datetime
-d time | --duration=time
-c i[gnore]|r[espect] | --case=i[gnore]|r[espect]
-C '<string>' | --columns='<string>'
-f 'string' | --find='string'
-F 'string' | --nofind='string'
-m regex | --match=regex
-M regex | --nomatch=regex
-t | --trouble
-n integer | --tail=integer
-s offset [limit] | --slice=offset [limit]
-o output_file | --out=output_file
-z 0-9 | --zip=0-9
-a | --append
input_file
-u | --unzip
--help
--version

```

gmapreduce

Runs Greenplum MapReduce jobs as defined in a YAML specification document.

```

gmapreduce -f yaml_file [dbname [username]] [-k name=value | --key name=value]
[-h hostname | --host hostname] [-p port | --port port] [-U username | --username
username] [-W] [-v]

gmapreduce -V | --version

gmapreduce -h | --help

-f yaml_file
-? | --help
-V | --version
-v | --verbose
-k | --key name=value
-h host | --host host
-p port | --port port
-U username | --username username
-W | --password

```

gpmigrator

Upgrades an existing Greenplum Database 3.2.x.x system to 3.3.7.

```
gpmigrator old_GPHOME_path new_GPHOME_path
[-d master_data_directory] [-l logfile_directory]
[-q] [--debug] [-R]

gpmigrator --prepare [--master_port port --port_base port --mirror_port_base
port] old_GPHOME_path new_GPHOME_path
[-d master_data_directory] [-l logfile_directory] [-q]
[--debug]

gpmigrator --version | -v

gpmigrator --help | -h

old_GPHOME_path
new_GPHOME_path
--prepare
--master_port port
--port_base port
--mirror_port_base port
-R (revert)
-d master_data_directory
-l logfile_directory
-q (quiet mode)
--help | -h
--debug
--version | -v
```

gprebuildsystem

Rebuilds a Greenplum Database array using a backup file of the Greenplum Database schema and system catalog tables.

```
gprebuildsystem -f gp_catalog_sql_file -d temp_db_data_directory [-p
temp_db_port] [-c] [-C number_checkpoints] [-x] [-a] [-q]
```

```

[-l logfile_directory] [-D]
gprebuildsystem -?
gprebuildsystem -v
-f gp_catalog_sql_file
-d temp_db_data_directory
-c (prompt for changes to array configuration)
-C number_checkpoint_segments
-p temp_db_port
-s (spread mirror configuration)
-a (do not prompt)
-q (no screen output)
-x (do not use cdatabase file)
-l logfile_directory
-D (debug)
-? (help)
-v (show utility version)

```

gprecoverseg

Recovers a primary or mirror segment instance that has been marked as invalid (if mirroring is enabled).

```

gprecoverseg [-d master_data_directory] [-p new_recover_host | -s
pri_datadir,mir_datadir | -i recover_config_file]
[-B parallel_processes] [-F] [-z seg_data_dir:seg_hostname | -S seg_dbid] [-a]
[-q] [-l logfile_directory]
gprecoverseg -o output_sample_recover_config
gprecoverseg -?
gprecoverseg --version
-a (do not prompt)
-B parallel_processes
-d master_data_directory
-F (force shutdown)
-i recover_config_file
-l logfile_directory
-o output_sample_recover_config
-p new_recover_host
-q (no screen output)
-s pri_datadir,mir_datadir
-S seg_dbid (force recovery of a valid primary)
-z seg_data_dir:seg_hostname (force recovery of a valid primary)
--version (version)
-? (help)

```

gpscp

Copies files between multiple hosts at once.

```
gpscp { -f host_file | -h hostname [-h hostname ...] }
[-J character] [-v] [[user@]hostname:]file_to_copy [...] [[user@]host-
name:]copy_to_path
```

```
gpscp -?
```

```
gpscp --version
```

```
-f host_file
```

```
-h hostname
```

```
-J character
```

```
-v (verbose mode)
```

```
file_to_copy
```

```
copy_to_path
```

```
-? (help)
```

```
--version
```

gpsizecalc

Shows the disk space usage for a given database, table, or index.

```
gpsizecalc {-x database_name [-p | -m] | {-t schema.table_name [-h] [-i]
[-a database_name]}} [-s b|k|m|g|t] [-B parallel_processes] [-d data_directory]
[-f] [-l logfile_directory] [-D | -q]
```

```
gpsizecalc -?
```

```
gpsizecalc -v
```

```
-x database_name
```

```
-p (primary segments only)
```

```
-m (mirror segments only)
```

```
-t schema.table_name
```

```
-h (show one level of child table hierarchy)
```

```
-i (show index size)
```

```
-a database_name_for_table
```

```
-s b | k | m | g | t (show size in bytes, KB, MB, GB, or TB)
```

```
-B parallel_processes
```

```
-d master_data_directory
```

```
-f (per segment results)
```

```
-l logfile_directory
```

```
-D (show detailed output)
```

```
-q (no screen output)
```

```
-? (help)
```

```
-v (show utility version)
```

gpskew

Shows data distribution information for a Greenplum Database table.

```
gpskew -t schema.table_name [-w "WHERE_constraint"] [-h] [-r] [-f] [-m] [-a
database_name] [-B parallel_processes] [-c] [-d master_data_directory] [-l
logfile_directory] [-D]
```

```
gpskew -?
```

```
gpskew -v
```

```
-t schema.table_name
-w "WHERE_constraint"
-h (show child table hierarchy)
-r (show response times)
-f (show full output)
-m (mirror segments only)
-c (show distribution columns)
-a database_name
-B parallel_processes
-d master_data_directory
-l logfile_directory
-D (debug)
-? (help)
-v (show utility version)
```

gpssh

Provides ssh access to multiple hosts at once.

```
gpssh { -f host_file | -h hostname [-h hostname ...] } [-v] [-e] [bash_command]
```

```
gpssh -?
```

```
gpssh --version
```

```
-f host_file
-h hostname
-v (verbose mode)
-e (echo)
bash_command
-? (help)
--version
```

gpssh-exkeys

Exchanges SSH public keys between hosts.

```
gpssh-exkeys { -f host_file | -h hostname [-h hostname ...] | -e host_file -x
```

```

host_file}
gpssh-exkeys -?
gpssh-exkeys --version
-f host_file
-h hostname
-e existing_hosts_file
-x new_hosts_file
-? (help)
--version

```

gpstart

Starts a Greenplum Database system.

```

gpstart [-d master_data_directory] [-B parallel_processes] [-R] [-m] [-y] [-a]
[-l logfile_directory] [-v | -q]
gpstart --recover [-d master_data_directory] [-l logfile_directory] [-v | -q]
gpstart -? | -h | --help
gpstart --version
-a (do not prompt)
-B parallel_processes
-d master_data_directory
-l logfile_directory
-m (master only)
-q (no screen output)
-R (restricted mode)
-v (verbose output)
-y (do not start standby master)
-? | -h | --help (help)
--version (show utility version)

```

gpstate

Shows the status of a running Greenplum Database system.

```

gpstate [-d master_data_directory] [-B parallel_processes] [-s | -b | -Q] [-m]

```

```

[-c] [-p] [-i] [-f] [-q] [-l log_directory] [-D]
gpstate -t
gpstate -?
gpstate -v
-b (brief status)
-B parallel_processes
-c (show primary to mirror mappings)
-d master_data_directory
-D (debug)
-f (show standby master details)
-i (show Greenplum Database version)
-l logfile_directory
-m (list mirrors)
-p (show ports)
-q (no screen output)
-Q (quick status)
-s (detailed status)
-t (show default utility settings)
-v (show utility version)
-? (help)

```

gpstop

Stops or restarts a Greenplum Database system.

```
gpstop [-d master_data_directory] [-B parallel_processes]
```

```
[-M smart | fast | immediate] [-r] [-y] [-a] [-l logfile_directory] [-v | -q]  
gpstop -m [-d master_data_directory] [-y] [-l logfile_directory] [-C] [-v | -q]  
gpstop -u [-d master_data_directory] [-l logfile_directory] [-v | -q]  
gpstop --version  
gpstop -? | -h | --help  
-a (do not prompt)  
-B parallel_processes  
-d master_data_directory  
-l logfile_directory  
-m (master only)  
-M fast (fast shutdown - rollback)  
-M immediate (immediate shutdown - abort)  
-M smart (smart shutdown - warn)  
-q (no screen output)  
-r (restart)  
-u (reload master pg_hba.conf and postgresql.conf only)  
-v (verbose output)  
--version (show utility version)  
-y (do not stop standby master)  
-? | -h | --help (help)
```

gp_dump

Writes out a database to SQL script files, which can then be used to restore the database using `gp_restore`.

Synopsis

```
gp_dump [-a | -s] [-c] [-d] [-D] [-n schema] [-o] [-O] [-t
table_name] [-x] [-h hostname] [-p port] [-U username] [-W] [-i]
[-v] [--gp-c] [--gp-d=backup_directory] [--gp-r=reportfile]
[--gp-s=a|p|i[dbid, ...]] database_name
```

```
gp_dump -? | --help
```

```
gp_dump --version
```

Description

The `gp_dump` utility dumps the contents of a database into SQL script files, which can then be used to restore the database schema and user data at a later time using `gp_restore`. During a dump operation, users will still have full access to the database.

The functionality of `gp_dump` is analogous to PostgreSQL's `pg_dump` utility, which writes out (or dumps) the content of a database into a script file. The script file contains SQL commands that can be used to restore the databases, data, and global objects such as users, groups, and access permissions.

The functionality of `gp_dump` is modified to accommodate the distributed nature of a Greenplum database. Keep in mind that a database in Greenplum Database is actually comprised of several PostgreSQL instances (the master and all segments), each of which must be dumped individually. The `gp_dump` utility takes care of dumping all of the individual instances across the system.

The `gp_dump` utility performs the following actions and produces the following dump files by default:

On the master host

- Dumps the Greenplum Database system configuration tables into a SQL file in the master data directory. The default naming convention of this file is `gp_catalog_1_<dbid>_<timestamp>`. This file can be used by the `gprebuildsystem` utility to recover a Greenplum Database system in various situations.
- Dumps `CREATE DATABASE` SQL statements into a file in the master data directory. The default naming convention of this file is `gp_cdatabase_1_<dbid>_<timestamp>`. This statement can be run on the master instance to recreate the user database(s).
- Dumps the user database schema(s) into a SQL file in the master data directory. The default naming convention of this file is `gp_dump_1_<dbid>_<timestamp>`. This file is used by `gp_restore` to recreate the database schema(s).
- Creates a log file in the master data directory named `gp_dump_status_1_<dbid>_<timestamp>`.

- `gp_dump` launches a `gp_dump_agent` for each segment instance to be backed up. `gp_dump_agent` processes run on the segment hosts and report status back to the `gp_dump` process running on the master host.

On the segment hosts

- Dumps the user data for each segment instance into a SQL file in the segment instance's data directory. By default, only primary (or active) segment instances are backed up. The default naming convention of this file is `gp_dump_0_<dbid>_<timestamp>`. This file is used by `gp_restore` to recreate that particular segment of user data.
- Creates a log file in each segment instance's data directory named `gp_dump_status_0_<dbid>_<timestamp>`.

Note that the 14 digit timestamp is the number that uniquely identifies the backup job, and is part of the filename for each dump file created by a `gp_dump` operation. This timestamp must be passed to the `gp_restore` utility when restoring a Greenplum database.

Options

-a | --data-only

Dump only the data, not the schema (data definitions).

-s | --schema-only

Dump only the object definitions (schema), not data.

-c | --clean

Output commands to clean (drop) database objects prior to (the commands for) creating them.

-d | --inserts

Dump data as `INSERT` commands (rather than `COPY`). This will make restoration very slow; it is mainly useful for making dumps that can be loaded into non-PostgreSQL based databases. Note that the restore may fail altogether if you have rearranged column order. The `-D` option is safer, though even slower.

-D | --column-inserts

Dump data as `INSERT` commands with explicit column names (`INSERT INTO table (column, ...) VALUES ...`). This will make restoration very slow; it is mainly useful for making dumps that can be loaded into non-PostgreSQL based databases.

-n *schema* | --schema=*schema*

Dumps the contents of the named schema only. If this option is not specified, all non-system schemas in the target database will be dumped. You cannot backup system catalog schemas (such as `pg_catalog`) with `gp_dump`.

Caution: In this mode, `gp_dump` makes no attempt to dump any other database objects that objects in the selected schema may depend upon. Therefore, there is no guarantee that the results of a single-schema dump can be successfully restored by themselves into a clean database.

-o | --oids

Dump object identifiers (OIDs) as part of the data for every table. Use of OIDs is not recommended in Greenplum Database, so this option should not be used if restoring data to another Greenplum Database installation.

-O | --no-owner

Do not output commands to set ownership of objects to match the original database. By default, `gp_dump` issues `ALTER OWNER` or `SET SESSION AUTHORIZATION` statements to set ownership of created database objects. These statements will fail when the script is run unless it is started by a superuser (or the same user that owns all of the objects in the script). To make a script that can be restored by any user, but will give that user ownership of all the objects, specify `-O`.

-t *table* | --table=*table*

Dump data for the named table only. It is possible for there to be multiple tables with the same name in different schemas; if that is the case, all matching tables will be dumped. Specify both `--schema` and `--table` to select just one table.

Caution: In this mode, `gp_dump` makes no attempt to dump any other database objects that the selected table may depend upon. Therefore, there is no guarantee that the results of a single-table dump can be successfully restored by themselves into a clean database.

-x | --no-privileges | --no-acl

Prevents the dumping of access privileges (`GRANT/REVOKE` commands).

-h *hostname* | --host=*hostname*

The host name of the Greenplum Database master host. If not provided, the value of `$PGHOST` or the local host is used.

-p *port* | --port=*port*

The Greenplum Database master port. If not provided, the value of `$PGPORT` or the port number provided at compile time is used.

-U *username* | --username=*user*

The database super user account name, for example `gpadmin`. If not provided, the value of `$PGUSER` or the current OS user name is used.

-W (force password prompt)

Forces a password prompt. This will happen automatically if the server requires password authentication.

-i | --ignore-version

Ignores a version mismatch between `gp_dump` and the database server.

-v | --verbose

Specifies verbose mode. This will cause `gp_dump` to output detailed object comments and start/stop times to the dump file, and progress messages to standard error.

--gp-c (use gzip)

Use `gzip` for inline compression.

--gp-d=directoryname

Specifies the relative or absolute path where the backup files will be placed on each host. If this is a relative path, it is considered to be relative to the data directory. If the path does not exist, it will be created, if possible. If not specified, defaults to the data directory of each instance to be backed up. Using this option may be desirable if each segment host has multiple segment instances — it will create the dump files in a centralized location.

--gp-r=reportfile

Specifies the full path name where the backup job log file will be placed on the master host. If not specified, defaults to the master data directory or if running remotely, the current working directory.

--gp-s=a | p | i [dbid, ...] (backup all, primaries, or certain segments)

Specifies the set of segment instances to back up - (a)ll, (p)rimaries only or (i)ndividual segment instances only (followed by a comma-separated list of the segments `dbid` in brackets). The default is to backup only the primary (or active) segment instances.

database_name

Required. The name of the database you want to dump. If not specified, the value of `$PGDATABASE` will be used. The database name must be stated last after all other options have been specified.

-? | --help (help)

Displays the online help.

--version (show utility version)

Displays the version of this utility.

Examples

Back up a database:

```
gp_dump gpdb
```

Back up a database, and create dump files in a centralized location on all hosts:

```
gp_dump --gp-d=/home/gpadmin/backups gpdb
```

Back up a particular schema only:

```
gp_dump -n myschema mydatabase
```

Back up a single segment instance only (by noting the *dbid* of the segment instance):

```
gp_dump --gp-s=i[5] gpdb
```

See Also

[gp_restore](#), [gpdrestore](#), [gprebuildsystem](#), [gpcrondump](#), [pg_dump](#)

gp_restore

Restores Greenplum databases that were backed up using `gp_dump`.

Synopsis

```
gp_restore --gp-k=timestamp_key -d database_name [-i] [-v] [-a |
-s] [-c] [-h hostname] [-p port] [-U username] [-W] [--gp-c]
[--gp-i] [--gp-d=directoryname] [--gp-r=reportfile]
[--gp-l=a|p|i[dbid, ...]]
```

```
gp_restore -? | -h | --help
```

```
gp_restore --version
```

Description

The `gp_restore` utility recreates the data definitions (schema) and user data in a Greenplum database using the script files created by an `gp_dump` operation. The use of this utility assumes:

1. You have backup files created by an `gp_dump` operation.
2. Your Greenplum Database system up and running.
3. Your Greenplum Database system has the exact same number of segment instances (primary and mirror) as the system that was backed up using `gp_dump`.
4. (optional) The `gp_restore` utility uses the information in the Greenplum system catalog tables to determine the hosts, ports, and data directories for the segment instances it is restoring. If you want to change any of this information (for example, move the system to a different array of hosts) you must use the `gprebuildsystem` and `gprebuildseg` scripts to reconfigure your array before restoring.
5. The databases you are restoring have been created in the system.
6. If you used the options `-s` (schema only) , `-a` (data only), `--gp-c` (compressed), `--gp-d` (alternate dump file location) when performing the `gp_dump` operation, you must specify these options when doing the `gp_restore` as well.

The functionality of `gp_restore` is analogous to PostgreSQL's `pg_restore` utility, which restores a database from files created by the database backup process. It issues the commands necessary to reconstruct the database to the state it was in at the time it was saved.

The functionality of `gp_restore` is modified to accommodate the distributed nature of a Greenplum database, and to use files created by an `gp_dump` operation. Keep in mind that a database in Greenplum is actually comprised of several PostgreSQL database instances (the master and all segments), each of which must be restored individually. The `gp_restore` utility takes care of populating each segment in the system with its own distinct portion of data.

The `gp_restore` utility performs the following actions:

On the master host

- Creates the user database schema(s) using the `gp_dump_1_<dbid>_<timestamp>` SQL file created by `gp_dump`.
- Creates a log file in the master data directory named `gp_restore_status_1_<dbid>_<timestamp>`.
- `gp_restore` launches a `gp_restore_agent` for each segment instance to be restored. `gp_restore_agent` processes run on the segment hosts and report status back to the `gp_restore` process running on the master host.

On the segment hosts

- Restores the user data for each segment instance using the `gp_dump_0_<dbid>_<timestamp>` files created by `gp_dump`. Each segment instance on a host (primary and mirror instances) are restored.
- Creates a log file for each segment instance named `gp_restore_status_0_<dbid>_<timestamp>`.

Note that the 14 digit timestamp is the number that uniquely identifies the backup job to be restored, and is part of the filename for each dump file created by a `gp_dump` operation. This timestamp must be passed to the `gp_restore` utility when restoring a Greenplum database.

Options**--gp-k=*timestamp_key***

Required. The 14 digit timestamp key that uniquely identifies the backup set of data to restore. This timestamp can be found in the `gp_dump` log file output, as well as at the end of the dump files created by a `gp_dump` operation. It is of the form `YYYYMMDDHHMMSS`.

-d *database_name* | --dbname=*dbname*

Required. The name of the database to connect to in order to restore the user data. The database(s) you are restoring must exist, `gp_restore` does not create the database.

-i | --ignore-version

Ignores a version mismatch between `gp_restore` and the database server.

-v | --verbose

Specifies verbose mode.

-a | --data-only

Restore only the data, not the schema (data definitions).

-c | --clean

Clean (drop) database objects before recreating them.

-s | --schema-only

Restores only the schema (data definitions), no user data is restored.

-h *hostname* | --host=*hostname*

The host name of the Greenplum master host. If not provided, the value of `PGHOST` or the local host is used.

-p *port* | --port=*port*

The Greenplum master port. If not provided, the value of `PGPORT` or the port number provided at compile time is used.

-U *username* | --username=*username*

The database super user account name, for example `gpadmin`. If not provided, the value of `PGUSER` or the current OS user name is used.

-W (force password prompt)

Forces a password prompt. This will happen automatically if the server requires password authentication.

--gp-c (use gunzip)

Use `gunzip` for inline decompression.

--gp-i (ignore errors)

Specifies that processing should ignore any errors that occur. Use this option to continue restore processing on errors.

--gp-d=*directoryname*

Specifies the relative or absolute path to backup files on the hosts. If this is a relative path, it is considered to be relative to the data directory. If not specified, defaults to the data directory of each instance being restored. Use this option if you created your backup files in an alternate location when running `gp_dump`.

--gp-r=*reportfile*

Specifies the full path name where the restore job report file will be placed on the master host. If not specified, defaults to the master data directory.

--gp-l=*a* | *p* | *i* [*dbid*, ...] (restore all, primaries, or certain segments)

Specifies whether to check for backup files on (a)ll segment instances, only (p)rimarily segment instances, or only (i)ndividual segment instances (followed by a comma-separated list of the segments *dbid* in brackets). The default is to check for primary segment backup files only, and then recreate the corresponding mirrors.

-? | -h | --help (help)

Displays the online help.

--version (show utility version)

Displays the version of this utility.

Examples

Restore an Greenplum database using backup files created by `gp_dump`:

```
gp_restore --gp-k=2005103112453 -d gpdb
```

Restore a single segment instance only (by noting the *dbid* of the segment instance):

```
gp_restore --gp-k=2005103112453 -d gpdb --gp-s=i[5]
```

See Also

[gp_dump](#), [gprebuildsystem](#), [pg_restore](#), [gpdbrestore](#)

gpactivatstandby

Activates a standby master host and makes it the active master for the Greenplum Database system.

Synopsis

```
gpactivatstandby -d standby_master_datadir
[-c new_standby_master] [-f] [-a] [-q] [-l logfile_directory]

gpactivatstandby -? | -h | --help

gpactivatstandby -v
```

Description

The `gpactivatstandby` utility activates a backup master host and brings it into operation as the active master instance for a Greenplum Database system. You must run this utility from the master host you are activating, not the failed master host you are disabling. Running this utility assumes you have a backup master host configured for the system (see `gpinitstandby`).

The utility will perform the following steps:

- Stop the synchronization process (`gpsyncagent`) on the backup master
- Update the system catalog tables of the backup master using the logs
- Activate the backup master to be the new active master for the system
- (optional) Make the host specified with the `-c` option the new standby master host
- Restart the Greenplum Database system with the new master host

A backup Greenplum master host serves as a ‘warm standby’ in the event of the primary Greenplum master host becoming unoperational. The backup master is kept up to date by a transaction log replication process (`gpsyncagent`), which runs on the backup master host and keeps the data between the primary and backup master hosts synchronized.

If the primary master fails, the log replication process is shutdown, and the backup master can be activated in its place by using the `gpactivatstandby` utility. Upon activation of the backup master, the replicated logs are used to reconstruct the state of the Greenplum master host at the time of the last successfully committed transaction. To specify a new standby master host after making your current standby master active, use the `-c` option.

In order to use `gpactivatstandby` to activate a new primary master host, the master host that was previously serving as the primary master cannot be running. The utility checks for a `postmaster.pid` file in the data directory of the disabled master host, and if it finds it there, it will assume the old master host is still active. In some cases, you may need to remove the `postmaster.pid` file from the disabled master host data directory before running `gpactivatstandby` (for example, if the disabled master host process was terminated unexpectedly).

After activating a standby master, run `ANALYZE` to update the database query statistics. For example:

```
psql dbname -c 'ANALYZE;'
```

Options

-d *standby_master_datadir*

Required. The absolute path of the data directory for the master host you are activating.

-c *new_standby_master_hostname*

Optional. After you activate your standby master you may want to specify another host to be the new standby, otherwise your Greenplum Database system will no longer have a standby master configured. Use this option to specify the hostname of the new standby master host. You can also use `gpinitstandby` at a later time to configure a new standby master host.

-f (force activation)

Use this option to force activation of the backup master host when the synchronization process (`gpsyncagent`) is not running. Only use this option if you are sure that the backup and primary master hosts are consistent, and you know the `gpsyncagent` process is not running on the backup master host. This option may be useful if you have just initialized a new backup master using `gpinitstandby`, and want to activate it immediately.

-a (do not prompt)

Do not prompt the user for confirmation.

-q (no screen output)

Run in quiet mode. Command output is not displayed on the screen, but is still written to the log file.

-l *logfile_directory*

The directory to write the log file. Defaults to `~/gpAdminLogs`.

-? | -h | --help (help)

Displays the online help.

-v (show utility version)

Displays the version, status, last updated date, and check sum of this utility.

Examples

Activate the backup master host and make it the active master instance for a Greenplum Database system (run from backup master host you are activating):

```
gpactivatestandby -d /gpdata
```

Activate the backup master host and at the same time configure another host to be your new standby master:

```
gpactivatestandby -d /gpdata -c new_standby_hostname
```

See Also

[gpinitssystem](#), [gpinitstandby](#)

gpaddmirrors

Adds mirror segments to a Greenplum Database system that was initially configured without mirroring.

Synopsis

```
gpaddmirrors [-d master_data_directory] -p port_offset [-m datadir_config_file] [-s] [-a] [-B parallel_processes] [-l logfile_directory] [-D]
```

```
gpaddmirrors [-d master_data_directory] -m datadir_config_file [-p port_offset] [-s] [-a] [-B parallel_processes] [-l logfile_directory] [-D]
```

```
gpaddmirrors [-d master_data_directory] -i mirror_config_file [-s] [-a] [-B parallel_processes] [-l logfile_directory] [-D]
```

Note: `-p` may be used alone or in combination with `-m`, but may not be used in combination with `-i`.

```
gpaddmirrors -o output_sample_mirror_config
```

```
gpaddmirrors -?
```

```
gpaddmirrors --version
```

Description

The `gpaddmirrors` utility configures mirror segment instances for an existing Greenplum Database system that was initially configured with primary segment instances only. The utility will shut down your Greenplum Database system while it creates the mirrors, and then restart it with mirroring enabled.

Creating the Mirror Data Directories

By default, the utility will prompt you for the data storage location(s) where it will create the mirror segment data directories. The utility will create a unique data directory for each mirror segment instance in this location using a predefined naming convention. There must be the same number of data directories declared for mirror segment instances as for primary segment instances. It is OK to specify the same directory name multiple times if you want your mirror data directories created in the same location, or you can enter a different data location for each mirror. Enter the absolute path. For example:

```
Enter mirror segment data directory location 1 of 2 > /gpdb/mirror
```

```
Enter mirror segment data directory location 2 of 2 > /gpdb/mirror
```

OR

```
Enter mirror segment data directory location 1 of 2 > /gpdb/m1
```

```
Enter mirror segment data directory location 2 of 2 > /gpdb/m2
```

Alternatively, you can run the `gpaddmirrors` utility and supply a configuration file containing the mirror host names and data directory locations (`-i` option). This is useful if you want your mirror segments on a completely different set of hosts than your primary segments. The format of the mirror configuration file is:

```
mirror[<content_id_number>]=<hostname>:<port>:<datadir_location>
```

The `content_id_number` is used to match a primary segment to its corresponding mirror segment. This corresponds to the `content` attribute in the `gp_configuration` system catalog table. The `hostname` is the host name of the segment host you are adding. If using a multi-NIC configuration, use the per-interface host names configured for the segment host. You must supply the exact same number of mirror segments as you have primary segments.

You must make sure that the user who runs `gpaddmirrors` (for example, the `gpadmin` user) has permissions to write to the data directory locations specified. You may want to create these directories on the segment hosts and `chown` them to the appropriate user before running `gpaddmirrors`.

Options

-a (do not prompt)

Run in quiet mode - do not prompt the user to enter information. Must supply a configuration file with `-m` if this option is used.

-B parallel_processes

The number of mirror creation processes to start in parallel. If not specified, the utility will start up to 10 parallel processes depending on how many mirror segment instances it needs to build.

-d master_data_directory

Optional. The master data directory. If not specified, the value set for `$MASTER_DATA_DIRECTORY` will be used.

-D (debug)

Sets logging output to debug level.

-i mirror_config_file

A configuration file containing one line for each mirror segment you want to create. You must have the same number of mirror hosts and segments per host as you do for your primary configuration. If you have a multi-NIC configuration on your segment and mirror hosts, use the per-interface host names in this file. Each line in the file specifies the data `content` identifier (as per the `gp_configuration` catalog table), mirror host name, mirror database listener port, and mirror data directory location in the format of:

```
mirror[<content_id>]=<hostname>:<port>:<datadir_location>
```

-l logfile_directory

The directory to write the log file. Defaults to `~/gpAdminLogs`.

-m *datadir_config_file*

A configuration file containing a list of data directory locations. The number of locations specified must correspond to the number of segments per host. Each line in the file specifies a data directory location. For example:

```
/gpdata/m1
/gpdata/m2
/gpdata/m3
/gpdata/m4
```

-o *output_sample_mirror_config*

If you are not sure how to lay out the mirror configuration file used by the **-m** option, you can run `gpaddmirrors` with this option to generate a sample mirror configuration file based on your primary segment configuration. You can then edit this file to change the host names to alternate mirror hosts if necessary.

-p *port_offset*

This number is used to calculate the ports used for mirror segments. This number is added to whatever number you have chosen for your primary segment port base.

The default offset is 1000. So for example, if your primary segments use a port base of 50000, then mirrors will use a port base of 51000 by default. If a host has multiple mirror segment instances, the base port number will be incremented by one for each additional mirror segment instance started on that host.

-s (*spread mirrors*)

Spreads the mirror segments across the available hosts. The default is to group a set of mirror segments together on an alternate host from their primary segment set.

Mirror spreading will place each mirror on a different host within the Greenplum Database array. Spreading is only allowed if there is a sufficient number of hosts in the array (number of hosts is greater than or equal to the number of segment instances).

--version (*show utility version*)

Displays the version, status, last updated date, and check sum of this utility.

-? (*help*)

Displays the online help.

Examples

Add mirroring to an existing Greenplum Database system using the same set of hosts as your primary data. Calculate the mirror port by adding 100 to the current primary segment port numbers:

```
gpaddmirrors -p 100
```

Add mirroring to an existing Greenplum Database system using a different set of hosts from your primary data:

```
gpaddmirrors -m mirror_config_file
```

Where the *mirror_config_file* looks something like this:

```
mirror[0]=sdw2:50100:/gpdata/m1
mirror[1]=sdw2:50101:/gpdata/m2
mirror[2]=sdw1:50100:/gpdata/m1
mirror[3]=sdw1:50101:/gpdata/m2
```

Output a sample mirror configuration file to use with `gpaddmirrors -m:`

```
gpaddmirrors -o /home/gpadmin/sample_mirror_config
```

See Also

[gpinitssystem](#), [gpinitstandby](#), [gpactivatestandby](#)

gpchecknet

Verifies the network performance between the specified hosts.

Synopsis

```
gpchecknet -d test_directory { -f host_file | -h hostname [-h
hostname ...] } [-r n|N] [-D] [-v | -V]
```

```
gpchecknet -?
```

```
gpchecknet --version
```

Description

The `gpchecknet` utility tests network performance between the hosts in your Greenplum Database array by running the `netperf` benchmark program. `gpchecknet` runs the `netperf TCP_STREAM` test, which transfers a 15 second stream of data between the hosts included in the test. By default, pairs of hosts are tested in parallel and the minimum, maximum, average and median network transfer rates are reported in megabytes (MB) per second. If this summary network transfer rate is slower than expected (less than 100 MB/s), you can run the network test serially using the `-r n` option to obtain per-host results. Serial mode is also recommended if you are testing an uneven number of hosts.

To specify the hosts to test, use the `-f` option to specify a file containing a list of host names, or use the `-h` option to name single host names on the command-line. At least one host name (`-h`) or a host file (`-f`) is required.

You must also specify a temporary test directory (with `-d`). The user who runs `gpchecknet` must have write access to the specified test directory on all remote hosts.

Before using `gpchecknet`, you must have a trusted host setup between the hosts involved in the network test. You can use the utility `gpssh-exkeys` to update the known host files and exchange public keys between hosts if you have not done so already.

Note that `gpchecknet` calls to `gpssh` and `gpscp`, so these Greenplum utilities must also be in your `$PATH`.

Options

-d *temp_test_directory*

Specifies a temporary test directory on the host(s) where the `netperf` test files will be copied. You must have write access to this directory on all hosts involved in the test.

-f *host_file*

Specifies the name of a file that contains a list of hosts that will participate in the network test. The host name is required, and you can optionally specify an alternate user name and/or SSH port number per host. The syntax of the host file is one host per line as follows:

```
[username@]hostname[:ssh_port]
```

-h *hostname*

Specifies a single host name that will participate in the network test. You can use the `-h` option multiple times to specify multiple host names.

-r *n|N*

Specifies how to run the network test: test pairs of hosts in parallel (`N`) or each host sequentially (`n`). By default the network performance test is run in parallel (`N`), and is accurate only if you have an even number of hosts. If the parallel network test indicates a performance problem, running the sequential network test (`n`) can help pinpoint the particular host(s) that are causing the problem. Note that you should use the sequential network test (`n`) if testing an uneven number of hosts to get truly accurate results.

-D (do not filter for multi-homed hosts)

The default behavior is to run the network test once per physical host, even if the host has multiple configured host names. When the `-D` option is used, the test is run once for each host name supplied with the `-f` or `-h` options, even if they resolve to the same physical host.

-v (verbose)

Verbose mode shows progress and status messages of the network tests as they are run.

-V (very verbose)

Very verbose mode shows all output messages generated by this utility.

-? (help)

Displays the online help.

--version

Displays the version of this utility.

Examples

Run the parallel network performance test on all the hosts in the file *host_file* using the test directory of *gpdb*:

```
gpchecknet -f host_file -d /gpdb
```

Run the serial network performance test on hosts *sdw1*, *sdw2*, and *sdw3* using the test directory of *gpdb*:

```
gpchecknet -h sdw1 -h sdw2 -h sdw3 -r n -d /gpdb
```

See Also

[gpssh](#), [gpscp](#), [gpcheckperf](#), [gpcheckos](#)

gpcheckkos

Checks the operating system environment of the specified hosts to make sure the configuration is compatible with Greenplum Database.

Synopsis

```
gpcheckkos { [-m] -f host_file | -h hostname [-h hostname ...] }
[-v]
```

```
gpcheckkos -?
```

```
gpcheckkos --version
```

Description

The `gpcheckkos` utility checks the operating system environment of the specified hosts to make sure it complies with the requirements for running Greenplum Database. This utility assumes that the hosts involved in the check have the same hardware configuration. If your master host has a different hardware configuration, it should be checked separately from the segment hosts. `gpcheckkos` performs the following checks:

- **Software Binaries** — Makes sure that each host has the same Greenplum Database server binaries installed by comparing the checksum.
- **Software Version** — Makes sure that the Greenplum Database software version installed on each host is the same.
- **Python Version** (master only) — Makes sure that Python 2.3.5 is installed on each host. This exact version of Python is bundled with your Greenplum Database installation for use with the management utilities. This test is only run if you specify the `-m` option.
- **Greenplum Path** (master only) — Makes sure that `$GPHOME` is set on the specified hosts. This test is only run if you specify the `-m` option.
- **System Clocks** — Makes sure the system clocks are synchronized on all hosts
- **OS Platform** — Makes sure that all hosts have a supported OS platform installed and that all hosts have the same OS platform version installed.
- **OS Settings** — Makes sure that the recommended Greenplum Database system settings are correct in `/etc/sysctl.conf`
- **OS User Limits** — Makes sure that the recommended Greenplum Database user limits are set correctly for the current user.

Options

-f *host_file*

Specifies the name of a file that contains a list of hosts that will participate in the OS verification. The host name is required, and you can optionally specify an alternate user name and/or SSH port number per host. The syntax of the host file is one host per line as follows:

```
[username@] hostname[:ssh_port]
```

-h *hostname*

Specifies a single host name that will participate in the performance test. You can use the `-h` option multiple times to specify multiple host names. The syntax of the host name is:

```
[username@] hostname[:ssh_port]
```

-m (check master environment)

If provided, `gpcheckkos` will also check the specified hosts to make sure the `$GPHOME` environment variable is set and that the correct version of Python is installed. These checks are only relevant for the master host (or standby master).

-v (verbose)

Verbose mode shows progress and status messages of the OS verification tests as they are run.

-? (help)

Displays the online help.

--version

Displays the version of this utility.

Examples

Verify the OS environment on all the hosts in the file `seg_host_file`:

```
gpcheckkos -f seg_host_file
```

Verify the OS environment on the hosts named `seg1` and `seg2` and run in verbose mode:

```
gpcheckkos -h seg1 -h seg2 -v
```

Verify the OS environment on the master host only:

```
gpcheckkos -m -h mdw1
```

See Also

[gpcheckperf](#), [gpchecknet](#)

gpcheckperf

Verifies the baseline hardware performance of the specified hosts.

Synopsis

```
gpcheckperf -d test_directory [-d test_directory ...]
{-f host_file | -h hostname [-h hostname ...]} [-r ds] [-B
block_size] [-S file_size] [-D] [-v | -V]
```

```
gpcheckperf -?
```

```
gpcheckperf --version
```

Description

The `gpcheckperf` utility starts a session on the specified hosts and runs the following performance tests:

- **Disk I/O Test (`dd test`)** — To test the sequential throughput performance of a logical disk or file system, the utility uses the `dd` command, which is a standard UNIX utility. It times how long it takes to write and read a large file to and from disk and calculates your disk I/O performance in megabytes (MB) per second. By default, the file size that is used for the test is calculated at two times the total RAM on the host. This ensures that the test is truly testing disk I/O and not using the memory cache.
- **Memory Bandwidth Test (`stream`)** — To test memory bandwidth, the utility uses the `STREAM` benchmark program to measure sustainable memory bandwidth (in MB/s). This tests that your system is not limited in performance by the memory bandwidth of the system in relation to the computational performance of the CPU. In applications where the data set is large (as in Greenplum Database), low memory bandwidth is a major performance issue. If memory bandwidth is significantly lower than the theoretical bandwidth of the CPU, then it can cause the CPU to spend significant amounts of time waiting for data to arrive from system memory.

To specify the hosts to test, use the `-f` option to specify a file containing a list of host names, or use the `-h` option to name single host names on the command-line. When using `gpcheckperf` in a network divided into subnets, you must use multiple host files and run the utility multiple times to test each subnet separately. For example, if the network has four subnets and four network interfaces per host, run `gpcheckperf` four times with four separate host files.

You must also specify at least one test directory (with `-d`). The user who runs `gpcheckperf` must have write access to the specified test directories on all remote hosts. For the disk I/O test, the test directories should correspond to your segment data directories (primary and/or mirrors). For the memory bandwidth test, one test directory is still required to copy over the test program files.

Before using `gpcheckperf`, you must have a trusted host setup between the hosts involved in the performance test. You can use the utility `gpssh-exkeys` to update the known host files and exchange public keys between hosts if you have not done so already.

Note that `gpcheckperf` calls to `gpssh` and `gpscp`, so these Greenplum utilities must also be in your `$PATH`.

Options

-d *test_directory*

Specifies a single directory to test on the host(s). You must have write access to the test directory on all hosts involved in the performance test. You can use the `-d` option multiple times to specify multiple test directories (for example, to test disk I/O of your primary and mirror data directories).

-f *host_file*

Specifies the name of a file that contains a list of hosts that will participate in the performance test. The host name is required, and you can optionally specify an alternate user name and/or SSH port number per host. The syntax of the host file is one host per line as follows:

```
[username@]hostname[:ssh_port]
```

A given host file cannot contain mixed entries for hosts from separate subnets. When using `gpcheckperf` in a network divided into subnets, you must use multiple host files, (one per subnet) and run the utility multiple times to test each subnet separately.

-h *hostname*

Specifies a single host name that will participate in the performance test. You can use the `-h` option multiple times to specify multiple host names.

-r *ds*

Specifies which performance tests to run: disk I/O (`d`), stream test (`s`), or both of these tests (`ds`). The default is to run both tests (`ds`).

-B *block_size*

Specifies the block size to be used for disk I/O test. The default is 32KB, which is the same as the Greenplum Database page size. You can specify sizing in KB, MB, or GB.

-S *file_size*

Specifies the total file size to be used for the disk I/O test for all directories specified with `-d`. *file_size* should equal two times total RAM on the host. If not specified, the default is calculated at two times the total RAM on the host where `gpcheckperf` is executed. This ensures that the test is truly testing disk I/O and not using the memory cache. You can specify sizing in KB, MB, or GB.

-D (display per-host results)

Reports performance results for each host. The default is to report results for just the hosts with the minimum and maximum performance, as well as the total and average performance of all hosts.

-v (verbose)

Verbose mode shows progress and status messages of the performance tests as they are run.

-V (very verbose)

Very verbose mode shows all output messages generated by this utility.

-? (help)

Displays the online help.

--version

Displays the version of this utility.

Examples

Run the disk I/O and memory bandwidth tests on all the hosts in the file *host_file* using the test directory of */data/p1* and */data/m1*:

```
gpcheckperf -f host_file -d /data/p1 -d /data/m1
```

Run only the disk I/O test on the hosts named *seg1* and *seg2* using the test directory of */dbfast1*. Show individual host results and run in verbose mode:

```
gpcheckperf -h seg1 -h seg2 -d /dbfast1 -r d -D -v
```

See Also

[gpssh](#), [gpscp](#), [gpchecknet](#), [gpcheckos](#)

gpcrondump

A wrapper utility for `gp_dump`, which can be called directly or from a `crontab` entry.

Synopsis

```
gpcrondump -x database_name
[-s schema | -t schema.table | -T schema.table[,...]]
[-p | -m | -w dbid[,...]] [-u backup_directory]
[-R post_dump_script] [-c | -o] [-z] [-r] [-f free_space_percent]
[-b] [-i] [-j | -k] [-g] [-G] [-C] [-d master_data_directory]
[-B parallel_processes] [-a] [-q]
[-y reportfile] [-l logfile_directory] [-D]
{ [-E encoding] [--inserts | --column-inserts] [--oids]
[--no-owner | --use-set-session-authorization]
[--no-privileges] }
```

```
gpcrondump -?
```

```
gpcrondump -v
```

Description

`gpcrondump` is a wrapper utility for `gp_dump`. By default, dump files are created in their respective master and segment data directories in a directory named `db_dumps/YYYYMMDD`. The data dump files are compressed by default using `gzip`.

`gpcrondump` allows you to schedule routine backups of a Greenplum database using `cron` (a scheduling utility for UNIX operating systems). Cron jobs that call `gpcrondump` should be scheduled on the master host.

Email Notifications

To have `gpcrondump` send out status email notifications, you must place a file named `mail_contacts` in the home directory of the Greenplum superuser (`gpadmin`) or in the same directory as the `gpcrondump` utility (`$GPHOME/bin`). This file should contain one email address per line. `gpcrondump` will issue a warning if it cannot locate a `mail_contacts` file in either location. If both locations have a `mail_contacts` file, then the one in `$HOME` takes precedence.

Options

-a (do not prompt)

Do not prompt the user for confirmation.

-b (bypass disk space check)

Bypass disk space check. The default is to check for available disk space.

-B *parallel_processes*

The number of segments to check in parallel for pre/post-dump validation. If not specified, the utility will start up to 60 parallel processes depending on how many segment instances it needs to dump.

-c (clear old dump files first)

Clear out old dump files before doing the dump. The default is not to clear out old dump files. This will remove all old dump directories in the `db_dumps` directory, except for the dump directory of the current date.

-C (clean old catalog dumps)

Clean out old catalog schema dump files prior to create.

--column-inserts

Dump data as `INSERT` commands with column names.

-d *master_data_directory*

Optional. The master host data directory. If not specified, the value set for `$MASTER_DATA_DIRECTORY` will be used.

-D (debug)

Sets logging level to debug.

-E *encoding*

Character set encoding of dumped data. Defaults to the encoding of the database being dumped. See “[Character Set Support](#)” for the list of supported character sets.

-f *free_space_percent*

Percentage of free disk space to reserve when calculating the available free disk space for the dump operation. The default is 10.

-g (copy config files)

Secure a copy of the master and segment configuration files `postgresql.conf`, `pg_ident.conf`, and `pg_hba.conf`. These configuration files are dumped in the master or segment data directory to `db_dumps/YYYYMMDD/config_files_<timestamp>.tar`

-G (dump global objects)

Use `pg_dumpall` to dump global objects such as roles and tablespaces. Global objects are dumped in the master data directory to `db_dumps/YYYYMMDD/gp_global_1_1_<timestamp>`.

-i (ignore parameter check)

Ignore the initial parameter check phase.

--inserts

Dump data as `INSERT`, rather than `COPY` commands.

- j (vacuum before dump)**
Run `VACUUM` before the dump starts.
- k (vacuum after dump)**
Run `VACUUM` after the dump has completed successfully.
- l *logfile_directory***
The directory to write the log file. Defaults to `~/gpAdminLogs`.
- m (mirror segments only)**
Optional. Dump the mirror segment instances only. The default is to dump both primary and mirror segment instances. If a mirror is down, the dump will reconfigure itself and dump the active primary segment for the downed mirror.
- no-owner**
Do not output commands to set object ownership.
- no-privileges**
Do not output commands to set object privileges (`GRANT/REVOKE` commands).
- o (clear old dump files only)**
Clear out old dump files only, but do not run a dump. This will remove the oldest dump directory except not the current date's dump directory. All dump sets within that directory will be removed.
- oids**
Include object identifiers (oid) in dump data.
- p (primary segments only)**
Optional. Dump the primary segment instances only. The default is to dump both primary and mirror segment instances. If a primary is down, the dump will reconfigure itself and dump the active mirror segment for the downed primary.
- q (no screen output)**
Run in quiet mode. Command output is not displayed on the screen, but is still written to the log file.
- r (no rollback on failure)**
Do not rollback the dump files (delete a partial dump) if a failure is detected. The default is to rollback (delete partial dump files).
- R *post_dump_script***
The absolute path of a script to run after a successful dump operation. For example, you might want a script that moves completed dump files to a backup host. This script must reside in the same location on the master and all segment hosts.

-s *schema_name*

Optional. Dump only the named schema in the named database.

-t *schema.table_name*

Dump only the named table in this database.

-T *schema.table_name*

A comma-separated list of table names to *exclude* from the database dump.

-u *backup_directory*

Specifies the absolute path where the backup files will be placed on each host. If the path does not exist, it will be created, if possible. If not specified, defaults to the data directory of each instance to be backed up. Using this option may be desirable if each segment host has multiple segment instances as it will create the dump files in a centralized location rather than the segment data directories.

--use-set-session-authorization

Use `SET SESSION AUTHORIZATION` commands instead of `ALTER OWNER` commands to set object ownership.

-v (show utility version)

Displays the version, status, last updated date, and check sum of this utility.

-w *dbid[,...]* (backup certain segments only)

Specifies a set of segment instances to back up as a comma-separated list of the segments' *dbid*. The master is added to the list automatically. The default is to backup only the primary (or active) segment instances.

-x *database_name*

Required. The name of the Greenplum database to dump.

-y *reportfile*

Specifies the full path name where the backup job log file will be placed on the master host. If not specified, defaults to the master data directory or if running remotely, the current working directory.

-z (no compression)

Do not use compression. Default is to compress the dump files.

-? (help)

Displays the online help.

Examples

Call `gpcrondump` directly and dump *mydatabase* (and global objects):

```
gpcrondump -x mydatabase -c -g -G
```

A Linux `crontab` entry that runs a backup of the *sales* database (and global objects) nightly at one past midnight:

```
SHELL=/bin/bash
GPHOME=/usr/local/greenplum-db-3.3.7.x
MASTER_DATA_DIRECTORY=/data/gpdb_p1/gp-1
PATH=$PATH:$GPHOME/bin
01 0 * * * gpadmin gpccrondump -x sales -c -g -G -a -q >>
gp_salesdump.log
```

A Solaris `crontab` entry that runs a backup of the *sales* database (and global objects) nightly at one past midnight (no line breaks):

```
01 0 * * * SHELL=/bin/bash
GPHOME=/usr/local/greenplum-db-3.3.7.x PATH=$PATH:$GPHOME/bin
HOME=/export/home/gpadmin
MASTER_DATA_DIRECTORY=/data/gpdb_p1/gp-1
/usr/local/greenplum-db/bin/gpccrondump -x sales -c -g -G -a
-q >> gp_salesdump.log
```

See Also

[gp_dump](#), [gpdbrestore](#)

gpdbrestore

A wrapper utility around `gp_restore`. Restores a database from a set of dump files generated by `gpcrondump`.

Synopsis

```
gpdbrestore { -t timestamp_key [-L] | -b YYYYMMDD |
-R hostname:path_to_dumpset | -s database_name }
[-T schema.table [,...]] [-e] [-G] [-B parallel_processes]
[-d master_data_directory] [-a] [-q] [-l logfile_directory] [-D]
```

```
gpdbrestore -?
```

```
gpdbrestore -v
```

Description

`gpdbrestore` is a wrapper around `gp_restore`, which provides some convenience and flexibility in restoring from a set of backup files created by `gpcrondump`. This utility provides the following additional functionality on top of `gp_restore`:

- Automatically reconfigures for compression
- Validates the number of dump files are correct (For primary only, mirror only, primary and mirror, or a subset consisting of some mirror and primary segment dump files)
- If a failed segment is detected and if `gp_fault_action = continue`, restores to active segment instances
- Do not need to know the complete timestamp key (`-t`) of the backup set to restore. Additional options are provided to instead give just a date (`-b`), backup set directory location (`-R`), or database name (`-s`) to restore
- The `-R` option allows the ability to restore from a backup set located on a host outside of the Greenplum Database array (archive host). Ensures that the correct dump file goes to the correct segment instance.
- Identifies the database name automatically from the backup set.
- Allows you to restore particular tables only (`-T` option) instead of the entire database.
- Can restore global objects such as roles and tablespaces (`-G` option).
- Detects if the backup set is primary segments only or primary and mirror segments and passes the appropriate options to `gp_restore`.
- Allows you to drop the target database before a restore in a single operation.

Options

-a (do not prompt)

Do not prompt the user for confirmation.

-b YYYYMMDD

Looks for dump files in the segment data directories on the Greenplum Database array of hosts in `db_dumps/YYYYMMDD`

-B parallel_processes

The number of segments to check in parallel for pre/post-restore validation. If not specified, the utility will start up to 60 parallel processes depending on how many segment instances it needs to restore.

-d master_data_directory

Optional. The master host data directory. If not specified, the value set for `$MASTER_DATA_DIRECTORY` will be used.

-D (debug)

Sets logging level to debug.

-e (drop target database before restore)

Drops the target database before doing the restore and then recreates it.

-G (restore global objects)

Restores global objects such as roles and tablespaces if the global object dump file `db_dumps/<date>/gp_global_1_1_<timestamp>` is found in the master data directory.

-l logfile_directory

The directory to write the log file. Defaults to `~/gpAdminLogs`.

-L (list tablenames in backup set)

When used with the `-t` option, lists the table names that exist in the named backup set and exits. Does not do a restore.

-q (no screen output)

Run in quiet mode. Command output is not displayed on the screen, but is still written to the log file.

-R hostname:path_to_dumpset

Allows you to provide a hostname and full path to a set of dump files. The host does not have to be in the Greenplum Database array of hosts, but must be accessible from the Greenplum master.

-s database_name

Looks for latest set of dump files for the given database name in the segment data directories `db_dumps` directory on the Greenplum Database array of hosts.

-t *timestamp_key*

The 14 digit timestamp key that uniquely identifies a backup set of data to restore. It is of the form *YYYYMMDDHHMMSS*. Looks for dump files matching this timestamp key in the segment data directories *db_dumps* directory on the Greenplum Database array of hosts.

-T *schema.table_name*

A comma-separated list of specific table names to restore. The named table(s) must exist in the backup set of the database being restored.

-v (show utility version)

Displays the version, status, last updated date, and check sum of this utility.

-? (help)

Displays the online help.

Examples

Restore the *sales* database from the latest backup files generated by *gpcrondump* (assumes backup files are in the segment data directories in *db_dumps*)

```
gpdbrestore -s sales
```

Restore a database from backup files that reside on an archive host outside the Greenplum Database array (command issued on the Greenplum master host):

```
gpdbrestore -R archivehostname:/data_p1/db_dumps/20080214
```

Restore global objects only (roles and tablespaces):

```
gpdbrestore -G
```

See Also

[gpcrondump](#), [gp_restore](#)

gpdeletesystem

Deletes a Greenplum Database system that was initialized using [gpinitssystem](#).

Synopsis

```
gpdeletesystem -d master_data_directory [-B parallel_processes]
[-f] [-l logfile_directory] [-D]
```

```
gpdeletesystem -?
```

```
gpdeletesystem -v
```

Description

The `gpdeletesystem` utility will perform the following actions:

- Stop all `postgres` processes (the segment instances and master instance).
- Deletes all data directories.

Before running `gpdeletesystem`:

- Move any backup files (created by [gp_dump](#)) out of the master and segment data directories.
- Make sure that Greenplum Database is running.
- If you are currently in a segment data directory, change directory to another location. The utility fails with an error when run from within a segment data directory.

This utility will not uninstall the Greenplum Database software.

Options

-d *data_directory*

Required. The master host data directory.

-B *parallel_processes*

The number of segments to delete in parallel. If not specified, the utility will start up to 60 parallel processes depending on how many segment instances it needs to delete.

-f (**force**)

Force a delete even if backup files are found in the data directories. The default is to not delete Greenplum Database instances if backup files are present.

-l *logfile_directory*

The directory to write the log file. Defaults to `~/gpAdminLogs`.

-D (**debug**)

Sets logging level to debug.

-? (help)

Displays the online help.

-v (show utility version)

Displays the version, status, last updated date, and check sum of this utility.

Examples

Delete a Greenplum Database system:

```
gpdeletesystem -d /gpdata/gp-1
```

Delete a Greenplum Database system even if backup files are present:

```
gpdeletesystem -d /gpdata/gp-1 -f
```

See Also

[gpinitssystem](#), [gp_dump](#)

gpdetective

Collects diagnostic information from a running Greenplum Database system.

Synopsis

```
gpdetective [-h master_hostname] [-p master_port] [-U
gp_superuser] [-P password] [--start_date number_of_days |
YYYY-MM-DD ]
```

```
gpdetective -?
```

```
gpdetective -v
```

Description

The `gpdetective` utility collects information from a running Greenplum Database system and creates a bzip2-compressed tar output file. This output file can then be sent to Greenplum Customer Support to help with the diagnosis of Greenplum Database errors or system failures. The `gpdetective` utility runs the following diagnostic tests:

- Runs `gpstate` to check the system status
- Runs `gpcheckos` to make sure the recommended OS settings are set on all hosts
- Runs `pg_dumpall` to capture the schema DDL for all databases
- Runs `gpcheckcat` and `gpcheckdb` to check the system catalog tables for inconsistencies

And secures the following information:

- The master and segment `postgresql.conf` configuration file
- The master and segment server log files (provided that the log files are located in the default location)
- Core files (if any)
- Array configuration information
- Schema information

A bzip2-compressed tar output file containing this information is created in the current directory with a file name of `gpdetective<timestamp>.tar.bz2`.

Options

-h *master_hostname*

The host name of the machine on which the Greenplum master database server is running. If not specified, reads from the environment variable `PGHOST` or defaults to `localhost`.

-p *master_port*

The TCP port on which the Greenplum master database server is listening for connections. If not specified, reads from the environment variable `PGPORT` or defaults to 5432.

-P *password*

If Greenplum Database is configured to use password authentication, you must also supply the database superuser password. If not specified, reads from `~/ .pgpass` if it exists.

[--start_date *number_of_days* | *YYYY-MM-DD*]

Sets the start date for the diagnostic information collected. Can specify either the number of days prior, or an explicit past date.

-U *gp_superuser*

The Greenplum database superuser role name to connect as (typically `gpadmin`). If not specified, reads from the environment variable `PGUSER` or defaults to the current system user name.

-v (show utility version)

Displays the version of this utility.

-? (help)

Displays the utility usage and syntax.

Examples

Collect the diagnostic information for the past two days of activity in a Greenplum Database system:

```
gpdetective --start_date=2
```

Collect diagnostic information for a Greenplum Database system and supply the required connection information for the master host:

```
gpdetective -h mdw -p 54320 -U gpadmin -P mypassword
```

See Also

[gpstate](#), [gpcheckos](#), [pg_dumpall](#)

gpexpand

Expands an existing Greenplum Database across new hosts in the array.

Synopsis

```
gpexpand [-f hosts_file] [-D database_name]
gpexpand -i input_file [-D database_name] [-B batch_size] [-V]
gpexpand [-d duration[hh][:mm[:ss]] | [-e 'YYYY-MM-DD hh:mm:ss']]
[-a] [-n parallel_processes] [-D database_name]
[--verbose | -v]
gpexpand -r | --rollback [-D database_name]
gpexpand -c [-D database_name]
gpexpand -? | -h | --help
gpexpand --version
```

Prerequisites

- You are logged in as the Greenplum Database superuser (`gpadmin`).
- The new segment hosts have been installed and configured as per the existing segment hosts. This involves:
 - Configuring the hardware and OS
 - Installing the Greenplum software
 - Creating the `gpadmin` user account
 - Exchanging SSH keys.
- Enough disk space on your segment hosts to temporarily hold a copy of your largest table.

Description

The `gpexpand` utility performs array expansion in two phases: initialization, and redistribution.

In the initialization phase, `gpexpand` runs with an input file that specifies data directories, `dbid` values, and other characteristics of the new segments. You can create the input file manually, or by following the prompts in an interactive interview. In either case, you begin initialization by running `gpexpand -i input_file`.

If you choose to create the input file using the interactive interview, you can optionally specify a file containing a list of expansion hosts. If your platform or command shell limits the length of the list of hostnames that you can type when prompted in the interview, specifying the hosts with `-f` may be mandatory.

In addition to initializing the segments, the initialization phase performs these actions:

- Creates an expansion schema to store the status of the expansion operation, including detailed status for tables.

- Nulls the distribution policy for all tables. The original distribution policies are later restored in the reorganization phase.

To begin the redistribution phase, you must run `gpexpand` with either the `-d` (duration) or `-e` (end time) options. Until the specified end time or duration is reached, the utility will redistribute tables in the expansion schema. Each table is reorganized using `ALTER TABLE` commands to rebalance the tables across new segments, and to set tables to their original distribution policy. If `gpexpand` completes the reorganization of all tables before the specified duration, it displays a success message and ends.

Options

-h (help)

Displays the online help.

-i *input_file*

Specify the expansion configuration file. For the required format and contents for this file, see “Expansion Input File Format” on page 216.

-f *hosts_file*

Specifies the name of a file that contains a list of new hosts for system expansion. Each line of the file must contain a single host name.

This file can contain hostnames with or without interfaces specified. The `gpexpand` utility handles either case, adding interface numbers to end of the hostname if the original nodes are configured with multiple interfaces.

-D *database_name*

Specifies the database in which to create the expansion schema and tables. If this option is not given, the setting for the environment variable `PGDATABASE` is used. The database templates `template1` and `template0` cannot be used.

-d [*hh*] [:*mm*] [:*ss*]]

Duration of the expansion session from beginning to end.

-e *YYYY-MM-DD hh:mm:ss*

Ending date and time for the expansion session.

-B *batch_size*

Batch size of remote commands to send to a given host before making a one-second pause. Default is 8. Valid values are 1-128.

The `gpexpand` utility issues a number of setup commands that may exceed the host’s maximum threshold for authenticated connections as defined by `MaxStartups` in the SSH daemon configuration. The one-second pause allows authentications to be completed before `gpexpand` issues any more commands.

The default value does not normally need to be changed. However, it may be necessary to reduce the maximum number of commands if `gpexpand` fails with connection errors such as `'ssh_exchange_identification: Connection closed by remote host.'`

-n *parallel_processes*

The number of tables to redistribute simultaneously. Valid values are 1 - 16.

Each table redistribution process requires two database connections: one to alter the table, and another to update the table's status in the expansion schema. Before increasing `-n`, check the current value of the server configuration parameter `max_connections` and make sure the maximum connection limit is not exceeded.

-a (no analyze)

At the completion of each table redistribution operation, `gpexpand` runs `ANALYZE` to update the table statistics. To prevent statistics collection, specify `-a`.

-c (clean)

Remove the expansion schema.

-r | --rollback (roll back)

Roll back a failed expansion setup operation. If the rollback command fails, attempt again using the `-D` option to specify the database that contains the expansion schema for the operation that you want to roll back.

-V (no vacuum)

Do not vacuum catalog tables before creating schema copy.

-? | -h | --help (help)

Displays the online help.

--verbose | -v

Verbose debugging output. With this option, the utility will output all DDL and DML used to expand the database.

--version

Display the utility's version number and exit.

Examples

Run `gpexpand` with an input file to initialize new segments and create the expansion schema in the default database:

```
$ gpexpand -i input_file
```

Run `gpexpand` for sixty hours maximum duration to redistribute tables to new segments:

```
$ gpexpand -d 60:00:00
```

See Also

[gpssh-exkeys](#)

gpfdist

Serves external table files to Greenplum Database segments.

Synopsis

```
gpfdist [-d directory] [-p http_port] [-l log_file] [-t timeout]
[-m max_length] [-v | -V]
```

```
gpfdist -?
```

```
gpfdist --version
```

Description

`gpfdist` is a utility that serves external table files to all Greenplum Database segments in parallel. The benefit of using `gpfdist` over other file protocols is that you are guaranteed maximum parallelism while reading from external tables, thereby offering the best performance as well as easier administration of external tables.

If files are compressed using `gzip` or `bzip2` (have a `.gz` or `.bz2` file extension), `gpfdist` will uncompress the files automatically provided that `gunzip` or `bunzip2` is in your path.

In order for `gpfdist` to be used to serve external table files, the external table definition (as defined by `CREATE EXTERNAL TABLE`) must specify the directory name that contains the external table files using the `gpfdist://` protocol.

When Greenplum Database users select from an external table, `gpfdist` parses and serves the data files evenly to all the segment instances in the Greenplum Database system.

Most likely, you will want to run `gpfdist` on your ETL machines rather than the hosts where Greenplum Database is installed. To install `gpfdist` on another host, simply copy the utility over to that host and add `gpfdist` to your `$PATH`.

Options

-d *directory*

The directory from which `gpfdist` will serve files. If not specified, defaults to the current directory.

-l *log_file*

The fully qualified path and log file name where standard output messages are to be logged.

-p *http_port*

The HTTP port on which `gpfdist` will serve files. Defaults to 8080.

-t *timeout*

Sets the time allowed for Greenplum Database to establish a connection to a `gpfdist` process. Default is 5 seconds. Allowed values are 2 to 30 seconds. May need to be increased on systems with a lot of network traffic.

-m *max_length*

Sets the maximum allowed data row length in bytes. Default is 32768. Should be used when user data includes very wide rows, i.e when "line too long" error message is received. Should not be used otherwise as it increases resource allocation. Valid range is 32K to 1MB.

-? (help**)**

Displays the online help.

-v (verbose**)**

Verbose mode shows progress and status messages.

-V (very verbose**)**

Verbose mode shows all output messages generated by this utility.

--version

Displays the version of this utility.

Examples

Serve files from a specified directory using port 8081 (and start `gpfdist` in the background):

```
gpfdist -d /var/load_files -p 8081 &
```

Start `gpfdist` in the background and redirect output and errors to a log file:

```
gpfdist -d /var/load_files -p 8081 -l /home/gpadmin/log &
```

To stop `gpfdist` when it is running in the background:

--First find its process id:

```
ps ax | grep gpfdist
```

OR on Solaris

```
ps -ef | grep gpfdist
```

--Then kill the process, for example:

```
kill 3456
```

See Also

CREATE EXTERNAL TABLE

gpinitstandby

Adds and/or initializes a standby master host for a Greenplum Database system.

Synopsis

```
gpinitstandby -s standby_hostname [-r] [-n] [-a] [-q] [-l
logfile_directory] [-B parallel_processes] [-D]
```

```
gpinitstandby -?
```

```
gpinitstandby -v
```

Description

The `gpinitstandby` utility adds a backup master host to your Greenplum Database system. If your system has an existing backup master host configured, use the `-r` option to remove it before adding the new standby master host. Before running this utility, make sure that the Greenplum Database software is installed on the backup master host and that you have exchanged SSH keys between hosts. See “[Installing Greenplum Database on the Master Host](#)” on page 40 for instructions. This utility should be run on the currently active *primary* master host.

The utility will perform the following steps:

- Shutdown your Greenplum Database system
- Update the Greenplum Database system catalog to remove the existing backup master host information (if the `-r` option is supplied)
- Update the Greenplum Database system catalog to add the new backup master host information (use the `-n` option to skip this step)
- Edit the `pg_hba.conf` files of the segment instances to allow access from the newly added standby master.
- Setup the backup master instance on the alternate master host
- Start the synchronization process
- Restart your Greenplum Database system

A backup master host serves as a ‘warm standby’ in the event of the primary master host becoming unoperational. The backup master is kept up to date by a transaction log replication process (`gpsyncagent`), which runs on the backup master host and keeps the data between the primary and backup master hosts synchronized. If the primary master fails, the log replication process is shutdown, and the backup master can be activated in its place by using the `gpactivatestandby` utility. Upon activation of the backup master, the replicated logs are used to reconstruct the state of the master host at the time of the last successfully committed transaction.

Options

-s *standby_hostname*

Required. The host name of the standby master host.

-r (remove standby master)

Removes the currently configured standby master host from your Greenplum Database system.

-n (resynchronize)

Use this option if you already have a standby master configured, and just want to resynchronize the data between the primary and backup master host. The Greenplum system catalog tables will not be updated.

-a (do not prompt)

Do not prompt the user for confirmation.

-q (no screen output)

Run in quiet mode. Command output is not displayed on the screen, but is still written to the log file.

-l logfile_directory

The directory to write the log file. Defaults to ~/gpAdminLogs.

-B parallel_processes

The number of segment `pg_hba.conf` files to edit in parallel. If not specified, the utility will start up to 40 parallel processes depending on how many segment instances it needs to alter.

-D (debug)

Sets logging level to debug.

-? (help)

Displays the online help.

-v (show utility version)

Displays the version, status, last updated date, and check sum of this utility.

Examples

Add a backup master host to your Greenplum Database system and start the synchronization process:

```
gpinitstandby -s host09
```

Remove the existing backup master and add a backup master on a different host:

```
gpinitstandby -s host09 -r
```

Start an existing backup master host and synchronize the data with the primary master host - do not add a new Greenplum backup master host to the system catalog:

```
gpinitstandby -s host09 -n
```

See Also

[gpinitssystem](#), [gpaddmirrors](#), [gpactivatestandby](#)

gpinitssystem

Initializes a Greenplum Database system by using configuration parameters specified in the `gp_init_config` file.

Synopsis

```
gpinitssystem -c gp_init_config_file
[-h host_file] [-r | -B parallel_processes]
[-p postgresql_conf_param_file] {[-s standby_master_host] [-i]}
[-o] [-m max_connections] [-b shared_buffer_size] [-e
superuser_password] [-n locale] [-S] [-a] [-q] [-l
logfile_directory] [-D]

gpinitssystem -?

gpinitssystem -v
```

Description

The `gpinitssystem` utility will create a Greenplum Database instance using the values defined in a Greenplum Database configuration file. See [“Initialization Configuration File Reference”](#) on page 788 for more information about this configuration file.

Before running this utility, make sure that you have installed the Greenplum Database software on all the hosts in the array. See [“Installing the Greenplum Software”](#) on page 40.

The functionality of `gpinitssystem` is analogous to PostgreSQL’s `initdb` utility, which creates the data storage directories, generates the shared catalog tables and configuration files, and creates a *template* database, which is a template used to create other databases.

In a Greenplum Database DBMS, each database instance (the master and all segments) must be initialized across all of the hosts in the system in such a way that they can all work together as a unified DBMS. The `gpinitssystem` utility takes care of initializing the database on the master and on each segment instance, and configuring the system as a whole.

Before running `gpinitssystem`, you must set the `$GPHOME` environment variable to point to the location of your Greenplum Database server installation on the master host.

This utility performs the following tasks:

- Verifies that the parameters in the configuration file are correct.
- Ensures that a connection can be established to each host. If a host cannot be reached, the utility will exit.
- Displays the configuration that will be used and prompts the user for confirmation.
- Initializes the master instance.
- Initializes the standby master instance (if specified).

- Initializes the primary segment instances.
- Initializes the mirror segment instances (if mirroring is configured).
- Configures the Greenplum Database system and checks for errors.
- Starts the Greenplum Database system.

Options

-a (do not prompt)

Do not prompt the user for confirmation.

-b *shared_buffer_size*

Sets the amount of memory a Greenplum server instance uses for shared memory buffers. You can specify sizing in kilobytes (kB), megabytes (MB) or gigabytes (GB). The default is 125MB.

-B *parallel_processes*

The number of segments to create in parallel. If not specified, the utility will start up to 4 parallel processes at a time.

-c *gp_init_config_file*

Required. The full path and filename of the configuration file, which contains all of the defined parameters to configure and initialize a new array. See “[Initialization Configuration File Reference](#)” on page 788 for a description of this file.

-D (debug)

Sets log output level to debug.

-e *superuser_password*

The password to set for the Greenplum Database super user. Defaults to `gparray`. You can always change the super user password at a later time using the `ALTER ROLE` command. Client connections over the network require a password login for the database super user account (for example, the `gpadmin` user).

-h *host_file*

Optional. The full path and filename of a file that contains the host names of all the segment hosts in the array. If not specified on the command line, you can specify the host file using the `MACHINE_LIST_FILE` parameter in the `gp_init_config` file.

-i (do not start synchronization)

If a standby master host is configured, this will not start the synchronization process between the primary and backup master hosts. The default is to start the synchronization process.

-l *logfile_directory*

The directory to write the log file. Defaults to `~/gpAdminLogs`.

-m *max_connections*

Sets the maximum number of client connections allowed to the master. The default is 25.

-n *locale*

Sets the default locale (character set encoding) for the Greenplum Database system. Defaults to `en_US.utf8` (UNICODE). You must make sure that the locale you specify exists on your operating system (`locale -a` will show which locales are available). See “[Character Set Support](#)” on page 52 for more information.

-o (create master only)

Creates a master instance only. Does not configure any segment hosts.

-p *postgresql_conf_param_file*

Optional. The name of a file that contains global `postgresql.conf` parameter settings that you want to set for Greenplum Database. These settings will be used when the individual master and segment instances are initialized. For more information about the default server configuration parameters for Greenplum Database, see “[Server Configuration Parameters](#)” on page 755.

-q (no screen output)

Run in quiet mode. Command output is not displayed on the screen, but is still written to the log file.

-r (serial mode)

Runs in serial mode. The default is to build segments in parallel.

-s *standby_master_host*

Optional. If you wish to configure a backup master host, specify the host name using this option. You must have the Greenplum Database software installed and configured on the backup master host.

-S (spread mirror configuration)

If mirroring parameters specified, spreads the mirror segments across the available hosts. The default is to group the set of mirror segments together on an alternate host from their primary segment set. Mirror spreading will place each mirror on a different host within the Greenplum Database array. Spreading is only allowed if there is a sufficient number of hosts in the array (number of hosts is greater than the number of segment instances).

-v (show utility version)

Displays the version, status, last updated date, and check sum of this utility.

-? (help)

Displays the online help.

Examples

Initialize a Greenplum Database array:

```
gpinitssystem -c gp_init_config
```

Initialize a Greenplum Database array and set the superuser remote password:

```
gpinitssystem -c gp_init_config -e mypassword
```

Initialize a Greenplum Database array with an optional standby master host:

```
gpinitssystem -c gp_init_config -s host09
```

Display the online help for the `gpinitssystem` utility:

```
gpinitssystem -?
```

See Also

[gprebuildsystem](#), [gpdeletesystem](#)

gpload

Runs a load job as defined in a YAML formatted control file.

Synopsis

```
gpload -f control_file [-l log_file] [-h hostname] [-p port] [-U username] [-d database] [--gpfdist_timeout number_seconds] [-W]
[[-v | -V] [-q]] [-D]
```

```
gpload -?
```

```
gpload --version
```

Prerequisites

The client machine where `gpload` is executed must have the following:

- Python 2.5.1 or later, `pg8000` (the Python interface to PostgreSQL), and `pyyaml`. Note that Python 2.6.2 and the required Python libraries are included with the Greenplum Database server installation, so if you have Greenplum Database installed on the machine where `gpload` is running, you do not need a separate Python installation.
- The `gpfdist` parallel file distribution program installed and in your `$PATH`. This program is located in `$GPHOME/bin` of your Greenplum Database server installation.
- Network access to and from all hosts in your Greenplum Database array (master and segments).
- Network access to and from the hosts where the data to be loaded resides (ETL servers).

Description

`gpload` is a data loading utility that acts as an interface to Greenplum Database's external table parallel loading feature. Using a load specification defined in a YAML formatted control file, `gpload` executes a load by invoking the Greenplum parallel file server (`gpfdist`), creating an external table definition based on the source data defined, and executing an `INSERT`, `UPDATE` or `MERGE` operation to load the source data into the target table in the database.

Options

-f control_file

Required. A YAML file that contains the load specification details. See “[Control File Format](#)” on page 635.

--gpfdist_timeout number_seconds

Sets the timeout in seconds for the `gpfdist` parallel file distribution program to send a response. May need to be increased on busy networks.

-l *log_file*

Specifies where to write the log file. Defaults to
~/gpAdminLogs/gpload_YYYYMMDD. See also, “Log File Format” on page 641.

-v (verbose mode)

Show verbose output of the load steps as they are executed.

-V (very verbose mode)

Shows very verbose output.

-q (no screen output)

Run in quiet mode. Command output is not displayed on the screen, but is still written to the log file.

-D (debug mode)

Check for error conditions, but do not execute the load.

-? (show help)

Show help, then exit.

--version

Show the version of this utility, then exit.

Connection Options**-d *database***

The database to load into. If not specified, reads from the load control file, the environment variable \$PGDATABASE or defaults to the current system user name.

-h *hostname*

Specifies the host name of the machine on which the Greenplum master database server is running. If not specified, reads from the load control file, the environment variable \$PGHOST or defaults to localhost.

-p *port*

Specifies the TCP port on which the Greenplum master database server is listening for connections. If not specified, reads from the load control file, the environment variable \$PGPORT or defaults to 5432.

-U *username*

The database role name to connect as. If not specified, reads from the load control file, the environment variable \$PGUSER or defaults to the current system user name.

-W (force password prompt)

Force a password prompt. If not specified, reads the password from the environment variable \$PGPASSWORD or from a password file specified by \$PGPASSFILE or in ~/.pgpass. If these are not set, then gpload will prompt for a password even if -W is not supplied.

Control File Format

The `gpload` control file uses the [YAML 1.1](#) document format and then implements its own schema for defining the various steps of a Greenplum Database load operation. The control file must be a valid YAML document.

The `gpload` program processes the control file document in order and uses indentation (spaces) to determine the document hierarchy and the relationships of the sections to one another. The use of white space is significant. White space should not be used simply for formatting purposes, and tabs should not be used at all.

The basic structure of a load control file is:

```

---
VERSION: 1.0.0.1
DATABASE: db_name
USER: db_username
HOST: master_hostname
PORT: master_port
GLOAD:
  INPUT:
    - SOURCE:
        LOCAL_HOSTNAME:
          - hostname_or_ip
        PORT: http_port
        | PORT_RANGE: [start_port_range, end_port_range]
        FILE:
          - /path/to/input_file
    - COLUMNS:
        - field_name: data_type
    - FORMAT: text | csv
    - DELIMITER: 'delimiter_character'
    - ESCAPE: 'escape_character' | 'OFF'
    - NULL_AS: 'null_string'
    - QUOTE: 'csv_quote_character'
    - HEADER: true | false
    - ENCODING: database_encoding
    - ERROR_LIMIT: integer
    - ERROR_TABLE: schema.table_name
  OUTPUT:
    - TABLE: schema.table_name
    - MODE: insert | update | merge

```

```

- MATCH_COLUMNS:
    - target_column_name
- UPDATE_COLUMNS:
    - target_column_name
- UPDATE_CONDITION: 'boolean_condition'
- MAPPING:
    target_column_name: source_column_name | 'expression'
SQL:
- BEFORE: "sql_command"
- AFTER: "sql_command"

```

VERSION

Optional. The version of the `gpload` control file schema. The current version is 1.0.0.1.

DATABASE

Optional. Specifies which database in Greenplum to connect to. If not specified, defaults to `$PGDATABASE` if set or the current system user name. You can also specify the database on the command line using the `-d` option.

USER

Optional. Specifies which database role to use to connect. If not specified, defaults to the current user or `$PGUSER` if set. You can also specify the database role on the command line using the `-U` option.

If the user running `gpload` is not a Greenplum superuser, then the server configuration parameter `gp_external_grant_privileges` must be set to `on` in order for the load to be processed.

HOST

Optional. Specifies Greenplum master host name. If not specified, defaults to `localhost` or `$PGHOST` if set. You can also specify the master host name on the command line using the `-h` option.

PORT

Optional. Specifies Greenplum master port. If not specified, defaults to 5432 or `$PGPORT` if set. You can also specify the master port on the command line using the `-p` option.

GPLOAD

Required. Begins the load specification section. A `GPLOAD` specification must have an `INPUT` and an `OUTPUT` section defined.

INPUT

Required. Defines the location and the format of the input data to be loaded. `gpload` will start one or more instances of the `gpfdist` file distribution program on the current host and create the required external table definition(s) in Greenplum Database that point to the source data. Note that the host from which you run `gpload` must be accessible over the network by all Greenplum hosts (master and segments).

SOURCE

Required. The `SOURCE` block of an `INPUT` specification defines the location of a source file. An `INPUT` section can have more than one `SOURCE` block defined. Each `SOURCE` block defined corresponds to one instance of the `gpfdist` file distribution program that will be started on the local machine. Each `SOURCE` block defined must have a `FILE` specification.

See “Using the Greenplum File Server (`gpfdist`)” on page 156 for more information about single and multiple `gpfdist` instances.

LOCAL_HOSTNAME

Optional. Specifies the host name or IP address of the local machine on which `gpload` is running. If this machine is configured with multiple network interface cards (NICs), you can specify the host name or IP of each individual NIC to allow network traffic to use all NICs simultaneously. The default is to use the local machine’s primary host name or IP only.

PORT

Optional. Specifies the specific port number that the `gpfdist` file distribution program should use. You can also supply a `PORT_RANGE` to select an available port from the specified range. If both `PORT` and `PORT_RANGE` are defined, then `PORT` takes precedence. If neither `PORT` or `PORT_RANGE` are defined, the default is to select an available port between 8000 and 9000.

If multiple host names are declared in `LOCAL_HOSTNAME`, this port number is used for all hosts. This configuration is desired if you want to use all NICs to load the same file or set of files in a given directory location.

PORT_RANGE

Optional. Can be used instead of `PORT` to supply a range of port numbers from which `gpload` can choose an available port for this instance of the `gpfdist` file distribution program.

FILE

Required. Specifies the location of a file, named pipe, or directory location on the local file system that contains data to be loaded. You can declare more than one file so long as the data is of the same format in all files specified.

If the files are compressed using `gzip` or `bzip2` (have a `.gz` or `.bz2` file extension), the files will be uncompressed automatically (provided that `gunzip` or `bunzip2` is in your path).

When specifying which source files to load, you can use the wildcard character (`*`) or other C-style pattern matching to denote multiple files. The files specified are assumed to be relative to the current directory from which `gpload` is executed (or you can declare an absolute path).

COLUMNS

Optional. Specifies the schema of the source data file(s) in the format of *field_name: data_type*. The `DELIMITER` character in the source file is what separates two data value fields (columns). A row is determined by a line feed character (`0x0a`).

If the input `COLUMNS` are not specified, then the schema of the output `TABLE` is implied, meaning that the source data must have the same column order, number of columns, and data format as the target table.

The default source-to-target mapping is based on a match of column names as defined in this section and the column names in the target `TABLE`. This default mapping can be overridden using the `MAPPING` section.

FORMAT

Optional. Specifies the format of the source data file(s) - either plain text (`TEXT`) or comma separated values (`CSV`) format. Defaults to `TEXT` if not specified. For more information about the format of the source data, see “[Formatting of Input Data](#)” on page 155.

DELIMITER

Optional. Specifies a single ASCII character that separates columns within each row (line) of data. The default is a tab character in `TEXT` mode, a comma in `CSV` mode.

ESCAPE

Optional. Specifies the single character that is used for C escape sequences (such as `\n`, `\t`, `\100`, and so on) and for escaping data characters that might otherwise be taken as row or column delimiters. Make sure to choose an escape character that is not used anywhere in your actual column data. The default escape character is a `\` (backslash), however it is possible to specify any other character to represent an escape. It is also possible to disable escaping by specifying the value `'OFF'` as the escape value. This is very useful for data such as web log data that has many embedded backslashes that are not intended to be escapes.

NULL_AS

Optional. Specifies the string that represents a null value. The default is `\N` (backslash-N) in `TEXT` mode, and an empty value with no quotations in `CSV` mode. You might prefer an empty string even in `TEXT` mode for cases where you do not want to distinguish nulls from empty strings. Any source data item that matches this string will be considered a null value.

QUOTE

Required when `FORMAT` is `CSV`. Specifies the quotation character for `CSV` mode. The default is double-quote (`"`).

HEADER

Optional. For `CSV` formatted data, specifies that the first line in the data file(s) is a header row (contains the names of the columns) and should not be included as data to be loaded. If using multiple source files, all files must have a header row. The default is to assume that the input files do not have a header row.

ENCODING

Optional. Character set encoding of the source data. Specify a string constant (such as `'SQL_ASCII'`), an integer encoding number, or `'DEFAULT'` to use the default client encoding. See “[Character Set Support](#)” on page 52. If not specified, the default client encoding is used.

ERROR_LIMIT

Optional. Enables single row error isolation mode for this load operation. When enabled, input rows that have format errors will be discarded provided that the error limit count is not reached on any Greenplum segment instance during input processing. If the error limit is not reached, all good rows will be loaded and any error rows will either be discarded or logged to the table specified in `ERROR_TABLE`. The default is to abort the load operation on the first error encountered. Note that single row error isolation only applies to data rows with format errors; for example, extra or missing attributes, attributes of a wrong data type, or invalid client encoding sequences. Constraint errors, such as primary key violations, will still cause the load operation to abort if encountered. See “[Defining External Tables in Single Row Error Isolation Mode](#)” on page 159 for more information.

ERROR_TABLE

Optional when `ERROR_LIMIT` is declared. Specifies an error table where rows with formatting errors will be logged when running in single row error isolation mode. You can then examine this error table to see error rows that were not loaded (if any). If the `error_table` specified already exists, it will be used. If it does not exist, it will be automatically generated. See “[Investigating Load Errors](#)” on page 160 for more information about error tables.

OUTPUT

Required. Defines the target table and final data column values that are to be loaded into the database.

TABLE

Required. The name of the target table to load into.

MODE

Optional. Defaults to INSERT if not specified. There are three available load modes:

INSERT - Loads data into the target table using the following method:

```
INSERT INTO target_table SELECT * FROM input_data;
```

UPDATE - Updates the `UPDATE_COLUMNS` of the target table where the rows have `MATCH_COLUMNS` attribute values equal to those of the input data, and the optional `UPDATE_CONDITION` is true. UPDATE is not supported on tables with a random distribution policy.

MERGE - Inserts new rows and updates the `UPDATE_COLUMNS` of existing rows where `MATCH_COLUMNS` attribute values are equal to those of the input data, and the optional `UPDATE_CONDITION` is true. New rows are identified when the `MATCH_COLUMNS` value in the source data does not have a corresponding value in the existing data of the target table. If there are multiple new `MATCH_COLUMNS` values that are the same, only one new row for that value will be inserted (use `UPDATE_CONDITION` to filter out the rows you want to discard). MERGE is not supported on tables with a random distribution policy.

MATCH_COLUMNS

Required if `MODE` is UPDATE or MERGE. Specifies the column(s) to use as the join condition for the update. The attribute value in the specified target column(s) must be equal to that of the corresponding source data column(s) in order for the row to be updated in the target table. The `MATCH_COLUMNS` declared must be the same as the Greenplum distribution key columns for the table.

UPDATE_COLUMNS

Required if `MODE` is UPDATE or MERGE. Specifies the column(s) to update for the rows that meet the `MATCH_COLUMNS` criteria and the optional `UPDATE_CONDITION`. Update columns cannot be columns that are used for the Greenplum distribution key for the table.

UPDATE_CONDITION

Optional. Specifies a Boolean condition (similar to what you would declare in a WHERE clause) that must be met in order for a row in the target table to be updated (or inserted in the case of a MERGE).

MAPPING

Optional. If a mapping is specified, it overrides the default source-to-target column mapping. The default source-to-target mapping is based on a match of column names as defined in the source `COLUMNS` section and the column names of the target `TABLE`. A mapping is specified as either:

```
target_column_name: source_column_name
```

or

```
target_column_name: 'expression'
```

Where *expression* is any expression that you would specify in the `SELECT` list of a query, such as a constant value, a column reference, an operator invocation, a function call, and so on.

SQL

Optional. Defines SQL commands to run before and/or after the load operation. You can specify multiple `BEFORE` and/or `AFTER` commands. List commands in the order of desired execution.

BEFORE

Optional. An SQL command to run before the load operation starts. Enclose commands in quotes.

AFTER

Optional. An SQL command to run after the load operation completes. Enclose commands in quotes.

Log File Format

Log files output by `gpload` have the following format:

```
timestamp|level|message
```

Where *timestamp* takes the form: YYYY-MM-DD HH:MM:SS, *level* is one of `DEBUG`, `LOG`, `INFO`, `ERROR`, and *message* is a normal text message.

Some `INFO` messages that may be of interest in the log files are (where # corresponds to the actual number of seconds, units of data, or failed rows):

```
INFO|running time: #.## seconds
INFO|transferred #.# kB of #.# kB.
INFO|gpload succeeded
INFO|gpload succeeded with warnings
INFO|gpload failed
INFO|1 bad row
INFO|# bad rows
```

Examples

Run a load job as defined in *my_load.yml*:

```
gpload -f my_load.yml
```

Example load control file:

```

---
VERSION: 1.0.0.1
DATABASE: ops
USER: gpadmin
HOST: mdw-1
PORT: 5432
Gpload:
  INPUT:
    - SOURCE:
      LOCAL_HOSTNAME:
        - etl1-1
        - etl1-2
        - etl1-3
        - etl1-4
      PORT: 8081
      FILE:
        - /var/load/data/*
    - COLUMNS:
      - name: text
      - amount: float4
      - category: text
      - desc: text
      - date: date
    - FORMAT: text
    - DELIMITER: '|'
    - ERROR_LIMIT: 25
    - ERROR_TABLE: payables.err_expenses
  OUTPUT:
    - TABLE: payables.expenses
    - MODE: INSERT
  SQL:
    - BEFORE: "INSERT INTO audit VALUES('start', current_timestamp)"
    - AFTER: "INSERT INTO audit VALUES('end', current_timestamp)"

```

See Also

`gpfdist`, `CREATE EXTERNAL TABLE`

gplogfilter

Searches through Greenplum Database log files for specified entries.

Synopsis

```
gplogfilter [timestamp_options] [pattern_options]
[output_options] [input_options] [input_file]
```

```
gplogfilter --help
```

```
gplogfilter --version
```

Description

The `gplogfilter` utility can be used to search through a Greenplum Database log file for entries matching the specified criteria. If an input file is not supplied, then `gplogfilter` will use the `$MASTER_DATA_DIRECTORY` environment variable to locate the Greenplum master log file in the standard logging location. To read from standard input, use a dash (-) as the input file name. Input files may be compressed using `gzip`. In an input file, a log entry is identified by its timestamp in `YYYY-MM-DD [hh:mm[:ss]]` format.

You can also use `gplogfilter` to search through all segment log files at once by running it through the `gpssh` utility. For example, to display the last three lines of each segment log file:

```
gpssh -f seg_host_file
=> source /usr/local/greenplum-db/greenplum_path.sh
=> gplogfilter -n 3 /gpdata/*/pg_log/gpdb*.log
```

By default, the output of `gplogfilter` is sent to standard output. Use the `-o` option to send the output to a file or a directory. If you supply an output file name ending in `.gz`, the output file will be compressed by default using maximum compression. If the output destination is a directory, the output file is given the same name as the input file.

Options

Timestamp Options

-b *datetime* | --begin=*datetime*

Specifies a starting date and time to begin searching for log entries in the format of `YYYY-MM-DD [hh:mm[:ss]]`.

-e *datetime* | --end=*datetime*

Specifies an ending date and time to stop searching for log entries in the format of `YYYY-MM-DD [hh:mm[:ss]]`.

-d *time* | --duration=*time*

Specifies a time duration to search for log entries in the format of [hh] [:mm[:ss]]. If used without either the `-b` or `-e` option, will use the current time as a basis.

Pattern Matching Options

-c *i*[gnore] | *r*[espect] | --case=*i*[gnore] | *r*[espect]

Matching of alphabetic characters is case sensitive by default unless preceded by the `--case=ignore` option.

-C '*<string>*' | --columns='*<string>*'

Selects specific columns from the log file. Specify the desired columns as a comma-delimited string of column numbers beginning with 1, where the second column from left is 2, the third is 3, and so on. See “Log File Format” on page 224 for a list of the available columns and their associated number.

-f '*string*' | --find='*string*'

Finds the log entries containing the specified string.

-F '*string*' | --nofind='*string*'

Rejects the log entries containing the specified string.

-m *regex* | --match=*regex*

Finds log entries that match the specified Python regular expression. See <http://docs.python.org/library/re.html> for Python regular expression syntax.

-M *regex* | --nomatch=*regex*

Rejects log entries that match the specified Python regular expression. See <http://docs.python.org/library/re.html> for Python regular expression syntax.

-t | --trouble

Finds only the log entries that have `ERROR:`, `FATAL:`, or `PANIC:` in the first line.

Output Options

-n *integer* | --tail=*integer*

Limits the output to the last *integer* of qualifying log entries found.

-s *offset* [*limit*] | --slice=*offset* [*limit*]

From the list of qualifying log entries, returns the *limit* number of entries starting at the *offset* entry number, where an *offset* of zero (0) denotes the first entry in the result set and an *offset* of any number greater than zero counts back from the end of the result set.

-o *output_file* | --out=*output_file*

Writes the output to the specified file or directory location instead of `STDOUT`.

-z 0-9 | --zip=0-9

Compresses the output file to the specified compression level using `gzip`, where 0 is no compression and 9 is maximum compression. If you supply an output file name ending in `.gz`, the output file will be compressed by default using maximum compression.

-a | --append

If the output file already exists, appends to the file instead of overwriting it.

Input Options

input_file

The name of the input log file(s) to search through. If an input file is not supplied, `gplogfilter` will use the `$MASTER_DATA_DIRECTORY` environment variable to locate the Greenplum master log file. To read from standard input, use a dash (`-`) as the input file name.

-u | --unzip

Uncompress the input file using `gunzip`. If the input file name ends in `.gz`, it will be uncompressed by default.

--help

Displays the online help.

--version

Displays the version of this utility.

Examples

Display the last three error messages in the master log file:

```
gplogfilter -t -n 3
```

Display all log messages in the master log file timestamped in the last 10 minutes:

```
gplogfilter -d :10
```

Display log messages in the master log file containing the string `|con6 cmd11|`:

```
gplogfilter -f '|con6 cmd11|'
```

Using `gpssh`, run `gplogfilter` on the segment hosts and search for log messages in the segment log files containing the string `con6` and save output to a file.

```
gpssh -f seg_hosts_file -e 'source
/usr/local/greenplum-db/greenplum_path.sh ; gplogfilter -f
con6 /gpdata/*/pg_log/gpdb.log' > seglog.out
```

See Also

[gpssh](#), [gpscp](#)

gmapreduce

Runs Greenplum MapReduce jobs as defined in a YAML specification document.

Synopsis

```
gmapreduce -f yaml_file [dbname [username]] [-k name=value |
--key name=value] [-h hostname | --host hostname] [-p port | --port
port] [-U username | --username username] [-W] [-v]

gmapreduce -V | --version

gmapreduce -h | --help
```

Prerequisites

The following are required prior to running this program:

- You must have your MapReduce job defined in a YAML file. See “[Greenplum MapReduce Specification](#)” on page 791.
- You must be a Greenplum Database superuser to run MapReduce jobs written in untrusted Perl or Python.
- You must be a Greenplum Database superuser to run MapReduce jobs with EXEC and FILE inputs.
- You must be a Greenplum Database superuser to run MapReduce jobs with GPFDIST input unless the server configuration parameter `gp_external_grant_privileges` is set to on.

Description

MapReduce is a programming model developed by Google for processing and generating large data sets on an array of commodity servers. Greenplum MapReduce allows programmers who are familiar with the MapReduce paradigm to write map and reduce functions and submit them to the Greenplum Database parallel engine for processing.

In order for Greenplum to be able to process MapReduce functions, the functions need to be defined in a YAML document, which is then passed to the Greenplum MapReduce program, `gmapreduce`, for execution by the Greenplum Database parallel engine. The Greenplum system takes care of the details of distributing the input data, executing the program across a set of machines, handling machine failures, and managing the required inter-machine communication.

Options

-f *yaml_file*

Required. The YAML file that contains the Greenplum MapReduce job definitions. See “[Greenplum MapReduce Specification](#)” on page 791.

-? | --help

Show help, then exit.

-V | --version

Show version information, then exit.

-v | --verbose

Show verbose output.

-k | --key *name=value*

Sets a YAML variable. A value is required. Defaults to “key” if no variable name is specified.

Connection Options

-h *host* | --host *host*

Specifies the host name of the machine on which the Greenplum master database server is running. If not specified, reads from the environment variable `PGHOST` or defaults to `localhost`.

-p *port* | --port *port*

Specifies the TCP port on which the Greenplum master database server is listening for connections. If not specified, reads from the environment variable `PGPORT` or defaults to `5432`.

-U *username* | --username *username*

The database role name to connect as. If not specified, reads from the environment variable `PGUSER` or defaults to the current system user name.

-W | --password

Force a password prompt.

Examples

Run a MapReduce job as defined in *my_yaml.txt* and connect to the database *mydatabase*:

```
gpmaphreduce -f my_yaml.txt mydatabase
```

See Also

[Greenplum MapReduce Specification](#)

gpmigrator

Upgrades an existing Greenplum Database 3.2.x.x system to 3.3.7.

Synopsis

```
gpmigrator old_GPHOME_path new_GPHOME_path
[-d master_data_directory] [-l logfile_directory]
[-q] [--debug] [-R]

gpmigrator --prepare [--master_port port --port_base port
--mirror_port_base port] old_GPHOME_path new_GPHOME_path
[-d master_data_directory] [-l logfile_directory] [-q]
[--debug]

gpmigrator --version | -v

gpmigrator --help | -h
```

Prerequisites

The following tasks should be performed prior to executing an upgrade:

- Make sure you are logged in to the master host as the Greenplum Database superuser (gpadmin).
- Install the Greenplum Database 3.3.7 binaries on all Greenplum hosts.
- Copy any custom modules from your existing installation to your 3.3.7 installation on all Greenplum hosts. For example, shared library files for user-defined functions in `$GPHOME/lib` or PostgreSQL add-on modules (such as `plr.so` or `pgcrypto.so`) in `$GPHOME/lib/postgresql`.
- Copy or preserve any additional folders or files (such as backup folders) that you have added in the Greenplum data directories or `$GPHOME` directory. Only files or folders strictly related to Greenplum Database operation are preserved by the migration utility.
- Check for and recover any failed segments in your current Greenplum Database system (`gpstate`, `gprecoverseg`).
- Update your environment to source the 3.3.7 installation.
- Verify the integrity of your current system catalogs using the `--prepare` step of `gpmigrator`. This will check your system catalogs and indexes and initialize a temporary 3.3.7 system. If problems are found, contact Greenplum customer support.
- Update your environment to temporarily source the 3.2 installation (required to back up and shut down the 3.2 system).
- Backup your current databases (`gpcrondump` or ZFS snapshots). If you find any issues when testing your upgraded system, you can restore this backup.
- Remove the standby master from your system configuration (`gpinitstandby -r`).
- Do a clean shutdown of your current system (`gpstop`).

- Update your environment to source the 3.3.7 installation.
- Inform all database users of the upgrade and lockout time frame. Once the upgrade is in process, users will not be allowed on the system until the upgrade is complete.



Note: Failing to meet any of these prerequisites can result in an aborted upgrade, placing your system in an unusable or even unrecoverable state. Make sure you meet all the prerequisites before running `gpmigrator`.

Description

The `gpmigrator` utility upgrades an existing Greenplum Database 3.2.x.x system to 3.3.7. This utility updates the system catalog and internal version number, but not the actual software binaries.

During the migration process, all client connections to the master will be locked out. The migration utility locks all client access to the master, however it does not block direct utility mode access to the segments. To ensure a safe upgrade, make sure users are aware that any connections to a Greenplum segment are not safe during the upgrade time frame.

Prior to executing the upgrade, you can optionally run `gpmigrator` in prepare mode to perform some pre-upgrade validation without actually executing the upgrade. The prepare phase does three things; it runs a utility called `gpcheckcat` to verify your system catalogs, it checks for invalid indexes that need rebuilding, and it initializes a temporary 3.3.7 system.

Options

old_GPHOME_path

Required. The absolute path to the current version of Greenplum Database software you want to migrate away from.

new_GPHOME_path

Required. The absolute path to the new version of Greenplum Database software you want to migrate to.

`--prepare`

Optional. Running `gpmigrator` with the `--prepare` option will perform some pre-upgrade validation, but not execute the upgrade. The prepare phase does three things; it runs a utility called `gpcheckcat` to verify your system catalogs, it checks for invalid indexes that need rebuilding, and it initializes a temporary 3.3.7 system.

--master_port port
--port_base port
--mirror_port_base port

Optional. Used with the `--prepare` option to supply a temporary master port, primary segment port base, and mirror segment port base for the temporary 3.3.7 system to be initialized. These must be different than what your current system is using.

-R (revert)

Restores the backup directory in the event of a failed upgrade.

If the migration procedure fails and you cannot start the system using the binary files of the prior version, it is probable that `gpmigrator` was not able to revert automatically. In these cases, you can run `gpmigrator -R` to attempt to restore the backup directory.

Cases in which the migration is not automatically reverted include:

- The `gpmigrator` process is killed with a command such as `kill -9`
- The system crashes during the migration procedure

If a ZFS snapshot is available, backing up from those resources is preferred.

-d master_data_directory

Optional. The current master host data directory. If not specified, the value set for `$MASTER_DATA_DIRECTORY` will be used.

-l logfile_directory

The directory to write the log file. Defaults to `~/gpAdminLogs`.

-q (quiet mode)

Run in quiet mode. Command output is not displayed on the screen, but is still written to the log file.

--help | -h

Displays the online help.

--debug

Sets logging level to debug.

--version | -v

Displays the version of this utility.

Examples

Upgrade to version 3.3.7 from version 3.2.1.2 (first make sure you are using the 3.3.7 software version):

```
source /usr/local/greenplum-db-3.3.7.x/greenplum_path.sh
/usr/local/greenplum-db-3.3.7.x/bin/gpmigrator \
```

```
/usr/local/greenplum-db-3.2.1.2 \  
/usr/local/greenplum-db-3.3.7.x
```

Revert a failed upgrade that was not automatically reverted:

```
/usr/local/greenplum-db-3.3.7.x/bin/gpmigrator \  
/usr/local/greenplum-db-3.2.1.2 \  
/usr/local/greenplum-db-3.3.7.x -R
```

See Also

[gpstop](#), [gpstate](#), [gprecoverseg](#), [gpcrondump](#)

gprebuildsystem

Rebuilds a Greenplum Database array using a backup file of the Greenplum Database schema and system catalog tables.

Synopsis

```
gprebuildsystem -f gp_catalog_sql_file -d temp_db_data_directory
[-p temp_db_port] [-c] [-C number_checkpoints] [-x] [-a] [-q]
[-l logfile_directory] [-D]
```

```
gprebuildsystem -?
```

```
gprebuildsystem -v
```

Description

The `gprebuildsystem` utility will rebuild a Greenplum Database system using a SQL script file. This SQL script file must contain commands for creating the Greenplum Database schema and system catalog data from another Greenplum Database installation. A `gp_dump` or `gpcrondump` operation creates a file on the master host called `gp_catalog_1_<dbid>_<timestamp>`. This is the SQL script file that `gprebuildsystem` needs.

The use of the `gprebuildsystem` utility assumes that you have an existing or previous Greenplum Database system that you want to recreate. By default this utility uses the same hosts, ports, and data directories as the reference Greenplum Database system. You can use the `-c` option to reconfigure the system to a new set of hosts. This will prompt you to enter new segment host configuration information.

The use of this utility also assumes that you have the same number of segment instances as the reference system you are rebuilding. If you are dramatically changing your system configuration, you should use `gpinitssystem` to create a new Greenplum Database system from scratch.

Options

-f *gp_catalog_sql_file*

Required. A SQL file containing the Greenplum Database schema and system catalog information. This is the `gp_catalog_0_<dbid>_<timestamp>` file created on the Greenplum master host by a `gp_dump` or `gpcrondump` operation.

-d *temp_db_data_directory*

Required. The rebuild utility creates a temporary database in this location. The temporary database is used to parse the contents of the SQL script file, and is then deleted after the rebuild operation is complete. Choose a different data storage location than the one used by your production Greenplum master instance.

-c (prompt for changes to array configuration)

Optional. If rebuilding the array on a different set of hosts than the system that was originally backed up, use this option. The utility will then prompt you for the new segment host configuration information.

-C *number_checkpoint_segments*

Number of checkpoint segments. Default is 8.

-p *temp_db_port*

Optional. If not specified, the default port of 55555 is used. The rebuild utility starts the temporary database using this port. The temporary database is used to parse the contents of the SQL script file, and is then deleted after the rebuild operation is complete. Choose a different port than the one used by your production Greenplum master instance.

-s (spread mirror configuration)

If mirroring is enabled in the catalog dump file configuration, spreads the mirror segments across the available hosts. The default is to group the set of mirror segments together on an alternate host from their primary segment set. Mirror spreading will place each mirror on a different host within the Greenplum Database array. Spreading is only allowed if there is a sufficient number of hosts in the array (number of hosts is greater than or equal to the number of segment instances).

-a (do not prompt)

Do not prompt the user for confirmation.

-q (no screen output)

Run in quiet mode. Command output is not displayed on the screen, but is still written to the log file.

-x (do not use cdatabase file)

Do not create the database if *cdatabase* file exists in same directory as the *gp_catalog* SQL file. The default is to create the databases specified in this file if found.

-l *logfile_directory*

The directory to write the log file. Defaults to `~/gpAdminLogs`.

-D (debug)

Sets logging level to debug.

-? (help)

Displays the online help.

-v (show utility version)

Displays the version, status, last updated date, and check sum of this utility.

Examples

Rebuild an entire Greenplum Database system (master and segment instances) using the *exact* same array configuration as in the SQL backup file:

```
gprebuildsystem -f gp_catalog_0_1_2005103112453 -d  
/home/gpadmin/tmp
```

Rebuild an entire Greenplum Database system using a similar array configuration as in the SQL backup file, but prompt for changes to the segment host information:

```
gprebuildsystem -f gp_catalog_0_1_2005103112453 -d  
/home/gpadmin/tmp -c
```

See Also

[gpinitssystem](#), [gp_dump](#), [gpcrondump](#), [gp_restore](#), [gpdbrestore](#)

gprecoverseg

Recovers a primary or mirror segment instance that has been marked as invalid (if mirroring is enabled).

Synopsis

```
gprecoverseg [-d master_data_directory] [-p new_recover_host | -s
pri_datadir,mir_datadir | -i recover_config_file]
[-B parallel_processes] [-F] [-z seg_data_dir:seg_hostname | -S
seg_dbid] [-a] [-q] [-l logfile_directory]
```

```
gprecoverseg -o output_sample_recover_config
```

```
gprecoverseg -?
```

```
gprecoverseg --version
```

Description

The `gprecoverseg` utility recovers a failed segment instance if you have mirroring enabled. A segment instance can fail for several reasons, such as a host failure, network failure, or file system corruption. When a segment instance goes down it is marked as *invalid* in the Greenplum Database system catalog. In order to bring the segment instance back into operation again, you must first correct the problem that made it fail in the first place, and then recover the segment instance in the Greenplum Database system.

By default, a failed segment is recovered in place, meaning that the system brings the segment back online on the same host and data directory location on which it was originally configured. In some cases, this may not be possible (for example, if a host was physically damaged and cannot be recovered). In this situation, `gprecoverseg` allows you to recover failed segments to a completely new host (using `-p`), on an alternative data directory location on your remaining live segment hosts (using `-s`), or by supplying a recovery configuration file (using `-c`) in the format of:

```
failed_host:port:datadir [recovery_host:port:datadir]
```

The new recovery segment host must be pre-installed with the Greenplum Database software and configured exactly the same as the existing segment hosts. A spare data directory location must exist on all currently configured segment hosts and have enough disk space to accommodate the failed segments.

The Greenplum Database can run in two modes with regards to mirroring and fault tolerance — *continue* (read-write) mode and *readonly* mode. This mode is set with the `gp_fault_action` parameter in `postgresql.conf`.

If the system is in *continue* (read-write) mode, this utility will shutdown the system, copy the data directory of the active segment to the failed segment, update the segment `dbid` and `port` on the failed segment's local catalog after the copy, mark the failed segment as valid again in the global system catalog, and restart the system.

If the system is in *readonly* mode, the active segment and failed segment would not get out-of-sync in the event of a failure. The utility will still shutdown the system to recover in read-only mode, but the downtime is relatively short.

If you do not have mirroring enabled, you must take manual steps to recover a failed segment instance and then force restart the system as follows:

```
gpstop -r -z
```

Options

-a (do not prompt)

Do not prompt the user for confirmation.

-B *parallel_processes*

The number of segments to recover in parallel. If not specified, the utility will start up to four parallel processes depending on how many segment instances it needs to recover.

-d *master_data_directory*

Optional. The master host data directory. If not specified, the value set for `$MASTER_DATA_DIRECTORY` will be used.

-F (force shutdown)

Optional. Force shutdown and startup of segment instances marked as invalid.

-i *recover_config_file*

Specifies the name of a configuration file that has information about which failed segments to recover, and where to recover them. Each line in the file is in the following format:

```
failed_host:port:datadir [recovery_host:port:datadir]
```

You can use the `-o` option to output a sample recovery configuration file to use as a starting point.

-l *logfile_directory*

The directory to write the log file. Defaults to `~/gpAdminLogs`.

-o *output_sample_recover_config*

Specifies a file name and location to output a sample recovery configuration file. The output file lists the currently invalid segments and their default recovery location in the format that is required by the `-c` option. This file can be edited to supply alternate recovery locations if needed.

-p *new_recover_host*

Specifies a spare hostname outside of the currently configured Greenplum Database array on which to recover invalid segments. This spare host must have the Greenplum Database software installed and configured, and have the same hardware and OS configuration as the current segment hosts (same OS version, locales, `gpadmin` user account, data directory locations created, ssh keys exchanged, etc.).

-q (no screen output)

Run in quiet mode. Command output is not displayed on the screen, but is still written to the log file.

-s *pri_datadir,mir_datadir*

Specifies a spare data directory location on the currently configured segment hosts where you can recover failed segment instances. This file system location must exist on all segment hosts in the array and have sufficient disk space to accommodate recovered segments.

-S *seg_dbid* (force recovery of a valid primary)

Forces a recovery of a primary segment instance from its mirror even though the segment has not been marked invalid in the `gp_configuration` catalog. For example, when running in `readonly` mode, segments are not marked invalid in the catalog but you can still detect a down segment using `gpstate`. Use this option with care!

-z *seg_data_dir:seg_hostname* (force recovery of a valid primary)

Forces a recovery of a primary segment instance from its mirror even though the segment has not been marked invalid in the `gp_configuration` catalog. For example, when running in `readonly` mode, segments are not marked invalid in the catalog but you can still detect a down segment using `gpstate`. Use this option with care!

--version (version)

Displays the version of this utility.

-? (help)

Displays the online help.

Examples

Recover any failed segment instances in place:

```
gprecoverseg
```

Recover any failed segment instances to a newly configured spare segment host:

```
gprecoverseg -p newhostname
```

Output the default recovery configuration file:

```
gprecoverseg -o /home/gpadmin/recover_config
```

See Also

[gpstart](#), [gpstop](#)

gpscp

Copies files between multiple hosts at once.

Synopsis

```
gpscp { -f host_file | -h hostname [-h hostname ...] }
[-J character] [-v] [[user@]hostname:] file_to_copy [...]
[[user@]hostname:] copy_to_path
```

```
gpscp -?
```

```
gpscp --version
```

Description

The `gpscp` utility allows you to copy one or more files from the specified hosts to other specified hosts in one command using SCP (secure copy). For example, you can copy a file from the Greenplum Database master host to all of the segment hosts at the same time.

To specify the hosts involved in the SCP session, use the `-f` option to specify a file containing a list of host names, or use the `-h` option to name single host names on the command-line. At least one host name (`-h`) or a host file (`-f`) is required. The `-J` option allows you to specify a single character to substitute for the *hostname* in the copy from and to destination strings. If `-J` is not specified, the default substitution character is an equal sign (=). For example, the following command will copy `.bashrc` from the local host to `/home/gpadmin` on all hosts named in *host_file*:

```
gpscp -f host_file .bashrc =:/home/gpadmin
```

If a user name is not specified in the host list or with `user@` in the file path, `gpscp` will copy files as the currently logged in user. To determine the currently logged in user, do a `whoami` command. By default, `gpscp` goes to `$HOME` of the session user on the remote hosts after login. To ensure the file is copied to the correct location on the remote hosts, it is recommended that you use absolute paths.

Before using `gpscp`, you must have a trusted host setup between the hosts involved in the SCP session. You can use the utility `gpssh-exkeys` to update the known host files and exchange public keys between hosts if you have not done so already.

Options

-f *host_file*

Specifies the name of a file that contains a list of hosts that will participate in this SCP session. The host name is required, and you can optionally specify an alternate user name and/or SCP port number per host. The syntax of the host file is one host per line as follows:

```
[username@]hostname[:ssh_port]
```

-h *hostname*

Specifies a single host name that will participate in this SCP session. You can use the `-h` option multiple times to specify multiple host names.

-J *character*

The `-J` option allows you to specify a single character to substitute for the *hostname* in the copy from and to destination strings. If `-J` is not specified, the default substitution character is an equal sign (=).

-v (*verbose mode*)

Optional. Reports additional messages in addition to the SCP command output.

file_to_copy

Required. The file name (or absolute path) of a file that you want to copy to other hosts (or file locations). This can be either a file on the local host or on another named host.

copy_to_path

Required. The path where you want the file(s) to be copied on the named hosts. If an absolute path is not used, the file will be copied relative to `$HOME` of the session user. You can also use the equal sign '=' (or another character that you specify with the `-J` option) in place of a *hostname*. This will then substitute in each host name as specified in the supplied host file (`-f`) or with the `-h` option.

-? (*help*)

Displays the online help.

--*version*

Displays the version of this utility.

Examples

Copy the file named *installer.tar* to / on all the hosts in the file *host_file*.

```
gpscp -f host_file installer.tar =:/
```

Copy the file named *myfuncs.so* to the specified location on the hosts named *seg1* and *seg2*:

```
gpscp -h seg1 -h seg2 myfuncs.so \  
=:/usr/local/greenplum-db-3.3.7.x/lib
```

See Also

[gpssh-exkeys](#), [gpssh](#)

gpsizecalc

Shows the disk space usage for a given database, table, or index.

Synopsis

```
gpsizecalc {-x database_name [-p | -m] | {-t schema.table_name
[-h] [-i] [-a database_name]}} [-s b|k|m|g|t] [-B
parallel_processes] [-d data_directory] [-f] [-l
logfile_directory] [-D | -q]
```

```
gpsizecalc -?
```

```
gpsizecalc -v
```

Description

The `gpsizecalc` utility can be used to determine the disk space usage for a Greenplum database, table, or index. `gpsizecalc` displays the following information:

- The total disk space usage for a specified database (mirrors and primary combined).
- The primary disk space usage for a specified table. Note that for partitioned parent tables, you must use the `-h` option to inquire about child tables. This option only shows one level of inheritance.
- The primary disk space usage for a specified table index.
- The disk usage per active segment (if `-f` option is supplied).

If checking disk space usage on a specific table or index, sizing information is shown for active (primary) segments only. If checking size of a database, total sizing is shown by default (primary and mirror).

Options

-x database_name

Either a database name or a table name is required. The database for which you want to see the total disk space usage (primary and mirror segments).

-p (primary segments only)

Obtain database sizing information for primary segments only. The default is to show total disk space usage for databases (including mirrors) and to show primary disk space usage only for tables and indexes.

-m (mirror segments only)

Obtain database sizing information for mirror segments only. The default is to show total disk space usage for databases (including mirrors) and to show primary disk space usage only for tables and indexes.

-t *schema.table_name*

Either a database name or a table name is required. The schema qualified table name for which you want to see the disk space usage. For a table, disk space usage is shown for active segments only by default.

-h (show one level of child table hierarchy)

Optional. If the table supplied with the `-t` option has child tables associated with it, this option will show information for the first level of inherited child tables as well. For a nested partitioning design, only the first level of the hierarchy is shown.

-i (show index size)

Optional. Shows the index size for the named table.

-a *database_name_for_table*

Optional. The name of the database for the specified table. If not supplied, the utility uses the default database (which is the most recently created database).

-s *b | k | m | g | t* (show size in bytes, KB, MB, GB, or TB)

Optional. Display disk usage size information in bytes (b), kilobytes (k), megabytes (m), gigabytes (g), or terabytes (t). The default is kilobytes (k).

-B *parallel_processes*

The number of segments to check in parallel. If not specified, the utility will start up to 60 parallel processes depending on how many segment instances for which it needs to check sizing information.

-d *master_data_directory*

Optional. The master host data directory. If not specified, the value set for `$MASTER_DATA_DIRECTORY` will be used.

-f (per segment results)

Optional. Displays the disk space usage per active segment. The default is to show total disk space usage for all active segments.

-l *logfile_directory*

The directory to write the log file. Defaults to `~/gpAdminLogs`.

-D (show detailed output)

Optional. If supplied will output all utility log messages to the screen.

-q (no screen output)

Run in quiet mode. Command output is not displayed on the screen, but is still written to the log file.

-? (help)

Displays the online help.

-v (show utility version)

Displays the version, status, last updated date, and check sum of this utility.

Examples

Show the total disk space usage for a table:

```
gpsizecalc -t sales.products
```

Show the total disk space usage for a database. Show size in megabytes rather than kilobytes:

```
gpsizecalc -x gpdb -s m
```

Show the disk space usage per segment for a partitioned table:

```
gpsizecalc -t sales.products -h -f
```

Show the disk space usage per segment for a database:

```
gpsizecalc -x gpdb -f
```

Show the index sizes for a table that is in another database besides the default:

```
gpsizecalc -t sales.products -a gpdb2 -i
```

See Also

[gpstate](#), [gpskew](#)

gpskew

Shows data distribution information for a Greenplum Database table.

Synopsis

```
gpskew -t schema.table_name [-w "WHERE_constraint"] [-h] [-r]
[-f] [-m] [-a database_name] [-B parallel_processes]
[-c] [-d master_data_directory] [-l logfile_directory] [-D]
```

```
gpskew -?
```

```
gpskew -v
```

Description

The `gpskew` utility can be used to determine if table data is equally distributed across all of the active segments. `gpskew` reports the following information:

- The total number of records in the specified table. Note that for parent tables with inherited child tables, you must use the `-h` option to show child table skew as well.
- The number of records on each segment.
- The variance of records between segments. It takes the segment which has the maximum count, and the segment which has the minimum count, and reports the difference between these two segments.
- The segment response times (if `-r` is supplied).
- The distribution key column names (if `-c` is supplied).

Options

-t *schema.table_name*

Required. The schema qualified table name for which you want to see the data distribution.

-w "*WHERE_constraint*"

Allows a `WHERE` constraint to be applied when the utility does a `SELECT count(*)` to count the records on a segment. This is useful for checking child tables that are partitioned by date. For example:

```
gpskew -t schema.table -w "date between '2007-01-01' and
'2007-01-12'"
```

-h (show child table hierarchy)

Optional. If the table supplied with the `-t` option has child tables associated with it, this option will show information for the inherited child tables as well.

-r (show response times)

Optional. Show the response times for each segment.

-f (show full output)

Optional. Displays the full data skew results instead of the summary results.

-m (mirror segments only)

Show data skew for mirror segments. Default is to show primary segments only.

-c (show distribution columns)

Optional. Show the names of the distribution key columns only (does not compute data skew).

-a *database_name*

Optional. The database name. If not specified, `gpskew` tries to read the database name from the environment variable `PGDATABASE`. If that variable is not set, `template1` is used.

-B *parallel_processes*

The number of segments to check in parallel. If not specified, the utility will start up to 60 parallel processes depending on how many segment instances for which it needs to check data skew information.

-d *master_data_directory*

Optional. The master host data directory. If not specified, the value set for `$MASTER_DATA_DIRECTORY` will be used.

-l *logfile_directory*

The directory to write the log file. Defaults to `~/gpAdminLogs`.

-D (debug)

Sets logging level to debug.

-? (help)

Displays the online help.

-v (show utility version)

Displays the version, status, last updated date, and check sum of this utility.

Examples

Show the data distribution for a given table:

```
gpskew -t myschema.bigtable1
```

Show the data distribution for a partitioned table:

```
gpskew -t myschema.bigtable1 -h
```

Show the data distribution for a given table in a given database:

```
gpskew -t myschema.bigtable1 -a gpdb
```

Show the names of the distribution key columns only for the given table:

```
gpskew -t myschema.bigtable1 -c
```

See Also

[gpstate](#), [gpsizecalc](#)

gpssh

Provides ssh access to multiple hosts at once.

Synopsis

```
gpssh { -f host_file | -h hostname [-h hostname ...] } [-v] [-e]
[bash_command]
```

```
gpssh -?
```

```
gpssh --version
```

Description

The `gpssh` utility allows you to run bash shell commands on multiple hosts at once using SSH (secure shell). You can execute a single command by specifying it on the command-line, or omit the command to enter into an interactive command-line session.

To specify the hosts involved in the SSH session, use the `-f` option to specify a file containing a list of host names, or use the `-h` option to name single host names on the command-line. At least one host name (`-h`) or a host file (`-f`) is required. Note that the current host is *not* included in the session by default — to include the local host, you must explicitly declare it in the list of hosts involved in the session.

Before using `gpssh`, you must have a trusted host setup between the hosts involved in the SSH session. You can use the utility `gpssh-exkeys` to update the known host files and exchange public keys between hosts if you have not done so already.

If you do not specify a command on the command-line, `gpssh` will go into interactive mode. At the `gpssh` command prompt (`=>`), you can enter a command as you would in a regular bash terminal command-line, and the command will be executed on all hosts involved in the session. To end an interactive session, press `CTRL+D` on the keyboard or type `exit` or `quit`.

If a user name is not specified in the host file, `gpssh` will execute commands as the currently logged in user. To determine the currently logged in user, do a `whoami` command. By default, `gpssh` goes to `$HOME` of the session user on the remote hosts after login. To ensure commands are executed correctly on all remote hosts, you should always enter absolute paths.

Options

-f *host_file*

Specifies the name of a file that contains a list of hosts that will participate in this SSH session. The host name is required, and you can optionally specify an alternate user name and/or SSH port number per host. The syntax of the host file is one host per line as follows:

```
[username@]hostname[:ssh_port]
```

-h *hostname*

Specifies a single host name that will participate in this SSH session. You can use the `-h` option multiple times to specify multiple host names.

-v (verbose mode)

Optional. Reports additional messages in addition to the command output when running in non-interactive mode.

-e (echo)

Optional. Echoes the commands passed to each host and their resulting output while running in non-interactive mode.

bash_command

A bash shell command to execute on all hosts involved in this session (optionally enclosed in quotes). If not specified, `gpssh` will start an interactive session.

-? (help)

Displays the online help.

--version

Displays the version of this utility.

Examples

Start an interactive group SSH session with all hosts listed in the file *host_file*:

```
gpssh -f host_file
```

At the `gpssh` interactive command prompt, run a shell command on all the hosts involved in this session.

```
=> ls -a /data/p1
```

Exit an interactive session:

```
=> exit
```

```
=> quit
```

Start a non-interactive group SSH session with the hosts named *dw1* and *dw2* and pass a file containing several commands named *command_file* to `gpssh`:

```
gpssh -h dw1 -h dw2 -v -e < command_file
```

Execute single commands in non-interactive mode on hosts *dw2* and *localhost*:

```
gpssh -h dw2 -h localhost -v -e 'ls -a /dbfast1/gpdb-1'
```

```
gpssh -h dw2 -h localhost -v -e 'echo $GPHOME'
```

```
gpssh -h dw2 -h localhost -v -e 'ls -l | wc -l'
```

See Also

[gpssh-exkeys](#), [gpscp](#)

gpssh-exkeys

Exchanges SSH public keys between hosts.

Synopsis

```
gpssh-exkeys { -f host_file | -h hostname [-h hostname ...] | -e
host_file -x host_file}
```

```
gpssh-exkeys -?
```

```
gpssh-exkeys --version
```

Description

The `gpssh-exkeys` utility exchanges SSH keys between the specified hosts. This allows SSH connections between hosts without a password prompt. The utility is used to initially prepare a Greenplum Database system for password-free SSH access, and also to add hosts to an existing Greenplum Database system.

To specify the hosts involved in an initial SSH key exchange, use the `-f` option to specify a file containing a list of host names (recommended), or use the `-h` option to name single host names on the command-line. At least one host name (`-h`) or a host file is required. Note that the current host is included in the key exchange by default.

To specify the hosts involved in a system expansion SSH key exchange in an existing Greenplum Database system, use the `-e` and `-x` options. The `-e` option specifies a file containing a list of existing hosts in the system that already have SSH keys. The `-x` option specifies a file containing a list of new hosts that need to participate in the SSH key exchange.

Keys are exchanged as the currently logged in user. Greenplum recommends performing the key exchange process twice: once as `root` and once as the `gpadmin` user (the user designated to own your Greenplum Database installation). The Greenplum Database management utilities require that the same non-root user be created on all hosts in the Greenplum Database system, and the utilities must be able to connect as that user to all hosts without a password prompt.

The `gpssh-exkeys` utility performs key exchange using the following steps:

- Create an RSA identification key pair for the current user if one does not already exist. The public key of this pair is added to the current user's `authorized_keys` file.
- The host key of each host specified using the `-h`, `-f`, `-e`, and `-x` options is obtained using `ssh-keyscan`. These keys are added to the current user's `known_hosts` file.
- Access each host using SSH to (a) set up password-free SSH access for the current user from the current host and (b) obtain the `authorized_keys`, `known_hosts`, and `id_rsa.pub` files from each host specified using the `-e` option. This access may require entering a login password for each host.
- Add keys from the `id_rsa.pub` files obtained from each host specified using the `-e` option to the current user's `authorized_keys` file.

- For each host specified using the `-h`, `-f`, and `-x` options, push the current user's `authorized_keys`, `known_hosts`, and `id_rsa.pub` files to the host, replacing existing versions of those files, if any. For each host specified using the `-e` option, merge the current user's `authorized_keys` and `known_hosts` files.

Options

`-f host_file`

Specifies the name of a file that contains a list of hosts that will participate in the SSH key exchange. Each line of the file must contain a single host name.

`-h hostname`

Specifies a single host name that will participate in the SSH key exchange. You can use the `-h` option multiple times to specify multiple host names.

`-e existing_hosts_file`

Specifies the name of a file containing a list of hosts participating in an expansion SSH key exchange. Each host is presumed an existing host previously prepared using `gpssh-exkeys`. Each line of the file must contain a single host name. Hosts specified in this file can not be specified in the host file used with `-x`.

`-x new_hosts_file`

Specifies the name of a file containing a list of new hosts participating in an expansion SSH key exchange. Each host is considered a new host that was not previously prepared using `gpssh-exkeys`. Each line of the file must contain a single host name. Hosts specified in this file may not also be specified in the host file used with `-e`.

`-? (help)`

Displays the online help.

`--version`

Displays the version of this utility.

Examples

Exchange SSH keys between all hosts listed in the file *host_file*:

```
gpssh-exkeys -f host_file
```

Exchange SSH keys between the hosts *sdw1*, *sdw2*, and *sdw3*:

```
gpssh-exkeys -h sdw1 -h sdw2 -h sdw3
```

Exchange SSH keys between existing hosts *sdw1*, *sdw2* and *sdw3*, and new hosts *sdw4* and *sdw5* as part of a system expansion operation:

```
cat - >existing_hosts_file
sdw1
sdw2
```

```
sdw3  
^D  
cat - >new_hosts_file  
sdw4  
sdw5  
^D  
gpssh-exkeys -e existing_hosts_file -x new_hosts_file
```

See Also

[gpssh](#), [gpscp](#), [gpexpand](#)

gpstart

Starts a Greenplum Database system.

Synopsis

```
gpstart [-d master_data_directory] [-B parallel_processes] [-R]
[-m] [-y] [-a] [-l logfile_directory] [-v | -q]
```

```
gpstart --recover [-d master_data_directory] [-l
logfile_directory] [-v | -q]
```

```
gpstart -? | -h | --help
```

```
gpstart --version
```

Description

The `gpstart` utility is used to start the Greenplum Database server processes. When you start a Greenplum Database system, you are actually starting several `postgres` database server processes at once (the master and all of the segment instances). The `gpstart` utility handles the startup of the individual instances. Each instance is started in parallel.

The first time an administrator runs `gpstart`, the utility creates a hosts cache file named `.gpshostcache` in the user's home directory. Subsequently, the utility uses this list of hosts to start the system more efficiently. If new hosts are added to the system, you must manually remove this file from the `gpadmin` user's home directory. The utility will create a new hosts cache file at the next startup.

Before you can start a Greenplum Database system, you must have initialized the system using `gpinitssystem` first.

Options

-a (do not prompt)

Do not prompt the user for confirmation.

-B *parallel_processes*

The number of segments to start in parallel. If not specified, the utility will start up to 60 parallel processes depending on how many segment instances it needs to start.

-d *master_data_directory*

Optional. The master host data directory. If not specified, the value set for `$MASTER_DATA_DIRECTORY` will be used.

-l *logfile_directory*

The directory to write the log file. Defaults to `~/gpAdminLogs`.

-m (master only)

Optional. Starts the master instance only, which may be useful for maintenance tasks. This mode only allows connections to the master in utility mode. For example:

```
PGOPTIONS='-c gp_session_role=utility' psql
```

-q (no screen output)

Run in quiet mode. Command output is not displayed on the screen, but is still written to the log file.

-R (restricted mode)

Starts Greenplum Database in restricted mode (only database superusers are allowed to connect).

-v (verbose output)

Displays detailed status, progress and error messages output by the utility.

-y (do not start standby master)

Optional. Do not start the standby master host. The default is to start the standby master host and synchronization process.

-? | -h | --help (help)

Displays the online help.

--version (show utility version)

Displays the version of this utility.

Examples

Start a Greenplum Database system:

```
gpstart
```

Start a Greenplum Database system in restricted mode (only allow superuser connections):

```
gpstart -R
```

Start the Greenplum master instance only and connect in utility mode:

```
gpstart -m
PGOPTIONS='-c gp_session_role=utility' psql
```

Display the online help for the `gpstart` utility:

```
gpstart -?
```

See Also

[gpinitssystem](#), [gpstop](#)

gpstate

Shows the status of a running Greenplum Database system.

Synopsis

```
gpstate [-d master_data_directory] [-B parallel_processes] [-s |  
-b | -Q] [-m] [-c] [-p] [-i] [-f] [-q] [-l log_directory] [-D]
```

```
gpstate -t
```

```
gpstate -?
```

```
gpstate -v
```

Description

The `gpstate` utility displays information about a running database server instance. There is additional information you may want to know about a Greenplum Database system, since it is comprised of multiple PostgreSQL database instances spanning multiple machines. The `gpstate` utility provides additional status information for a Greenplum Database system, such as:

- Which segments are down (invalid status).
- Master and segment configuration information (hosts, data directories, etc.)
- The ports used by the system.
- A mapping of primary segments to their corresponding mirror segments.

Options

-b (brief status)

Optional. Display a brief summary of the state of the Greenplum Database system. This is the default option.

-B *parallel_processes*

The number of segments to check in parallel. If not specified, the utility will start up to 60 parallel processes depending on how many segment instances it needs to check.

-c (show primary to mirror mappings)

Optional. Display mapping of primary segments to their corresponding mirror segments.

-d *master_data_directory*

Optional. The master data directory. If not specified, the value set for `$MASTER_DATA_DIRECTORY` will be used.

-D (debug)

Sets logging level to debug.

- f (show standby master details)**
Display details of the standby master host if configured.
- i (show Greenplum Database version)**
Display the Greenplum Database software version information.
- l logfile_directory**
The directory to write the log file. Defaults to ~/gpAdminLogs.
- m (list mirrors)**
Optional. List the mirror segment instances in the system.
- p (show ports)**
List the port numbers used throughout the Greenplum Database system.
- q (no screen output)**
Run in quiet mode. Command output is not displayed on the screen, but is still written to the log file.
- Q (quick status)**
Optional. Checks segment status in the system catalog on the master host. Does not poll the segments for status.
- s (detailed status)**
Optional. Displays detailed status information for the Greenplum Database system.
- t (show default utility settings)**
Display the default settings used by the `gpstate` utility such as the master data directory and port.
- v (show utility version)**
Displays the version, status, last updated date, and check sum of this utility.
- ? (help)**
Displays the online help.

Examples

Show detailed status information of a Greenplum Database system:

```
gpstate -s
```

Do a quick check for invalid segments in the master host system catalog:

```
gpstate -Q
```

Show information about mirror segment instances:

```
gpstate -m -c
```

Show information about the standby master configuration:

```
gpstate -f
```

Display the Greenplum software version information:

```
gpstate -i
```

See Also

[gpstart](#), [gpskew](#), [gpsizecalc](#), [gplogfilter](#)

gpstop

Stops or restarts a Greenplum Database system.

Synopsis

```
gpstop [-d master_data_directory] [-B parallel_processes]
[-M smart | fast | immediate] [-r] [-y] [-a]
[-l logfile_directory] [-v | -q]

gpstop -m [-d master_data_directory] [-y] [-l logfile_directory]
[-C] [-v | -q]

gpstop -u [-d master_data_directory] [-l logfile_directory] [-v |
-q]

gpstop --version

gpstop -? | -h | --help
```

Description

The `gpstop` utility is used to stop the database servers that comprise a Greenplum Database system. When you stop a Greenplum Database system, you are actually stopping several `postgres` database server processes at once (the master and all of the segment instances). The `gpstop` utility handles the shutdown of the individual instances. Each instance is shutdown in parallel.

By default, you are not allowed to shut down Greenplum Database if there are any client connections to the database. Use the `-M immediate` option to terminate any connections before shutting down. If there are any transactions in progress, the default behavior is to wait for them to commit before shutting down. Use the `-M fast` or `-M immediate` options to either roll back or abort open transactions.

With the `-u` option, the utility uploads changes made to the master `pg_hba.conf` file or to *runtime* configuration parameters in the master `postgresql.conf` file without interruption of service. Note that any active sessions will not pickup the changes until they reconnect to the database.

Options

-a (do not prompt)

Do not prompt the user for confirmation.

-B parallel_processes

The number of segments to stop in parallel. If not specified, the utility will start up to 60 parallel processes depending on how many segment instances it needs to stop.

-d master_data_directory

Optional. The master host data directory. If not specified, the value set for `$MASTER_DATA_DIRECTORY` will be used.

-l logfile_directory

The directory to write the log file. Defaults to ~/gpAdminLogs.

-m (master only)

Optional. Shuts down a Greenplum master instance that was started in maintenance mode.

-M fast (fast shutdown - rollback)

Fast shut down. Any transactions in progress are interrupted and rolled back.

-M immediate (immediate shutdown - abort)

Immediate shut down. Any transactions in progress are aborted. Use this option with care if you are running in *continue* fault tolerance mode (read-write), as this could cause data corruption.

-M smart (smart shutdown - warn)

Smart shut down. If there are active connections, this command fails with a warning. This is the default shutdown mode.

-q (no screen output)

Run in quiet mode. Command output is not displayed on the screen, but is still written to the log file.

-r (restart)

Restart after shutdown is complete.

-u (reload master pg_hba.conf and postgresql.conf only)

This option tells the master instance to reload the `pg_hba.conf` file and the runtime parameters of the `postgresql.conf` file but does not shutdown the Greenplum Database array. Use this option to make new configuration settings active after editing `postgresql.conf` or `pg_hba.conf`. Note that this only applies to configuration parameters that are designated as *runtime* parameters.

-v (verbose output)

Displays detailed status, progress and error messages output by the utility.

--version (show utility version)

Displays the version of this utility.

-y (do not stop standby master)

Do not stop the standby master process. The default is to stop the standby master.

-? | -h | --help (help)

Displays the online help.

Examples

Stop a Greenplum Database system in smart mode:

```
gpstop
```

Stop a Greenplum Database system in immediate mode:

```
gpstop -M immediate
```

Stop all segment instances and then restart the system:

```
gpstop -r
```

Stop a master instance that was started in maintenance mode:

```
gpstop -m
```

Reload the master `postgresql.conf` file and `pg_hba.conf` after making configuration changes but do not shutdown the Greenplum Database array:

```
gpstop -u
```

See Also

[gpstart](#)

C. Client Utility Reference

This appendix provides references for the command-line client utilities provided with Greenplum Database. Greenplum Database utilizes the standard PostgreSQL client programs, and also has additional management utilities to facilitate the administration of a distributed Greenplum Database DBMS.

The following Greenplum Database client programs are located in `$GPHOME/bin`:

- `clusterdb`
- `createdb`
- `createlang`
- `createuser`
- `dropdb`
- `droplang`
- `dropuser`
- `ecpg`
- `pg_config`
- `pg_dump`
- `pg_dumpall`
- `pg_restore`
- `psql`
- `reindexdb`
- `vacuumdb`

Client Utility Summary

clusterdb

Reclusters tables that were previously clustered with `CLUSTER`.

```
clusterdb [connection_option ...] [-t table] [[-d] dbname] [-e] [-q]
```

```
clusterdb [connection_option ...] [-a] [-e] [-q]
```

```
-a | --all
```

```
[-d] dbname | [--dbname] dbname
```

```
-e | --echo
```

```
-q | --quiet
```

```
-t table | --table table
```

```
-h host | --host host
```

```
-p port | --port port
```

```
-U username | --username username
```

```
-W | --password
```

createdb

Creates a new database.

```
createdb [connection_option ...] [-D tablespace] [-E encoding] [-O owner] [-T template] [-e] [-q] [dbname ['description']]
```

```
dbname
```

```
description
```

```
-D tablespace | --tablespace tablespace
```

```
-E encoding | --encoding encoding
```

```
-O owner | --owner owner
```

```
-T template | --template template
```

```
-e | --echo
```

```
-q | --quiet
```

```
-h host | --host host
```

```
-p port | --port port
```

```
-U username | --username username
```

```
-W | --password
```

createlang

Defines a new procedural language for a database.

```
createlang [connection_option ...] [-e] langname [[-d] dbname]
```

```
createlang [connection-option ...] -l dbname
```

langname

```
[-d] dbname | [--dbname] dbname
```

```
-e | --echo
```

```
-l dbname | --list dbname
```

```
-h host | --host host
```

```
-p port | --port port
```

```
-U username | --username username
```

```
-W | --password
```

dropdb

Removes a database.

```
dropdb [connection_option ...] [-e] [-i] [-q] dbname
```

dbname

```
-e | --echo
```

```
-i | --interactive
```

```
-q | --quiet
```

```
-h host | --host host
```

```
-p port | --port port
```

```
-U username | --username username
```

```
-W | --password
```

droplang

Removes a procedural language.

```
droplang [connection-option ...] [-e] langname [[-d] dbname]
```

```
droplang [connection-option ...] [-e] -l dbname
```

langname

```
[-d] dbname | [--dbname] dbname
```

```
-e | --echo
```

```
-l | --list
```

```
-h host | --host host
```

```
-p port | --port port
```

```
-U username | --username username
```

```
-W | --password
```

dropuser

Removes a database role.

```
dropuser [connection_option ...] [-e] [-i] [-q] role_name  
role_name  
-e | --echo  
-i | --interactive  
-q | --quiet  
-h host | --host host  
-p port | --port port  
-U username | --username username  
-W | --password
```

ecpg

Embedded SQL C preprocessor.

```
ecpg [option ...] file ...  
file  
-c  
-C mode  
-D symbol  
-i  
-I directory  
-o filename  
-r option  
-t  
-v  
--help  
--version
```

pg_config

Retrieves information about the installed version of Greenplum Database.

```
pg_config [option ...]
```

```
--bindir
```

```
--docdir
```

```
--includedir
```

```
--pkgincludedir
```

```
--includedir-server
```

```
--libdir
```

```
--pkglibdir
```

```
--localedir
```

```
--mandir
```

```
--sharedir
```

```
--sysconfdir
```

```
--pgxs
```

```
--configure
```

```
--cc
```

```
--cppflags
```

```
--cflags
```

```
--cflags_sl
```

```
--ldflags
```

```
--ldflags_sl
```

```
--libs
```

```
--version
```

pg_dump

Extracts a database into a single script file or other archive file.

```
pg_dump [connection_option ...] [dump_option ...] dbname
```

dbname

```
-a | --data-only
-b | --blobs
-c | --clean
-C | --create
-d | --inserts
-D | --column-inserts | --attribute-inserts
-E encoding | --encoding=encoding
-f file | --file=file
-F p|c|t | --format=plain|custom|tar
-i | --ignore-version
-n schema | --schema=schema
-N schema | --exclude-schema=schema
-o | --oids
-O | --no-owner
-s | --schema-only
-S username | --superuser=username
-t table | --table=table
-T table | --exclude-table=table
-v | --verbose
-x | --no-privileges | --no-acl
--disable-dollar-quoting
--disable-triggers
--use-set-session-authorization
--gp-syntax | --no-gp-syntax
-Z 0..9 | --compress=0..9
-h host | --host host
-p port | --port port
-U username | --username username
-W | --password
```

createuser

Creates a new database role.

```
createuser [connection_option ...] [role_attribute ...] [-e] [-q] role_name  
role_name  
-s | --superuser  
-S | --no-superuser  
-d | --createdb  
-D | --no-createdb  
-r | --createrole  
-R | --no-createrole  
-l | --login  
-L | --no-login  
-i | --inherit  
-I | --no-inherit  
-c number | --connection-limit number  
-P | --pwprompt  
-E | --encrypted  
-N | --unencrypted  
-e | --echo  
-q | --quiet  
-h host | --host host  
-p port | --port port  
-U username | --username username  
-W | --password
```

pg_dumpall

Extracts all databases in a Greenplum Database system to a single script file or other archive file.

```
pg_dumpall [connection_option ...] [dump_option ...]
```

```
-a | --data-only  
-c | --clean  
-d | --inserts  
-D | --column-inserts | --attribute-inserts  
-g | --globals-only  
-i | --ignore-version  
-o | --oids  
-O | --no-owner  
-s | --schema-only  
-S username | --superuser=username  
-v | --verbose  
-x | --no-privileges | --no-acl  
--disable-dollar-quoting  
--disable-triggers  
--use-set-session-authorization  
--gp-syntax  
-h host | --host host  
-p port | --port port  
-U username | --username username  
-W | --password
```

pg_restore

Restores a database from an archive file created by `pg_dump`.

```
pg_restore [connection_option ...] [restore_option ...] filename
filename
-a | --data-only
-c | --clean
-C | --create
-d dbname | --dbname=dbname
-e | --exit-on-error
-f outfilename | --file=outfilename
-F t|c | --format=tar|custom
-i | --ignore-version
-I index | --index=index
-l | --list
-L list-file | --use-list=list-file
-n schema | --schema=schema
-O | --no-owner
-P function-name(argtype [, ...]) | --function=function-name(argtype [, ...])
-s | --schema-only
-S username | --superuser=username
-t table | --table=table
-T trigger | --trigger=trigger
-v | --verbose
-x | --no-privileges | --no-acl
--disable-triggers
--no-data-for-failed-tables
-h host | --host host
-p port | --port port
-U username | --username username
-W | --password
-1 | --single-transaction
```

psql

Interactive command-line interface for Greenplum Database

```
psql [option...] [dbname [username]]
-a | --echo-all
-A | --no-align
-c 'command' | --command 'command'
-d dbname | --dbname dbname
-e | --echo-queries
-E | --echo-hidden
-f filename | --file filename
-F separator | --field-separator separator
-H | --html
-l | --list
-L filename | --log-file filename
-o filename | --output filename
-P assignment | --pset assignment
-q | --quiet
-R separator | --record-separator separator
-s | --single-step
-S | --single-line
-t | --tuples-only
-T table_options | --table-attr table_options
-v assignment | --set assignment | --variable assignment
-V | --version
-x | --expanded
-X | --no-psqlrc
-1 | --single-transaction
-? | --help
-h host | --host host
-p port | --port port
-U username | --username username
-W | --password
```

reindexdb

Rebuilds indexes in a database.

```
reindexdb [connection-option...] [--table | -t table] [--index | -i index] [db-
```

name]

```
reindexdb [connection-option...] [--all | -a]
reindexdb [connection-option...] [--system | -s] [dbname]
-a | --all
-s | --system
-t table | --table table
-i index | --index index
[-d] dbname | [--dbname] dbname
-e | --echo
-q | --quiet
-h host | --host host
-p port | --port port
-U username | --username username
-W | --password
```

vacuumdb

Garbage-collects and analyzes a database.

```
vacuumdb [connection-option...] [--full | -f] [--verbose | -v] [--analyze | -z]
[--table | -t table [( column [,...] )] ] [dbname]
vacuumdb [connection-options...] [--all | -a] [--full | -f] [--verbose | -v]
[--analyze | -z]
-a | --all
[-d] dbname | [--dbname] dbname
-e | --echo
-f | --full
-q | --quiet
-t table [(column)] | --table table [(column)]
-v | --verbose
-z | --analyze
-h host | --host host
-p port | --port port
-U username | --username username
-W | --password
```

clusterdb

Reclusters tables that were previously clustered with `CLUSTER`.

Synopsis

```
clusterdb [connection_option ...] [-t table] [[-d] dbname] [-e]
[-q]
```

```
clusterdb [connection_option ...] [-a] [-e] [-q]
```

Description

To cluster a table means to physically reorder a table on disk according to an index so that index scan operations can access data on disk in a somewhat sequential order, thereby improving index seek performance for queries that use that index.

The `clusterdb` utility will find any tables in a database that have previously been clustered with the `CLUSTER` SQL command, and clusters them again on the same index that was last used. Tables that have never been clustered are not affected.

`clusterdb` is a wrapper around the SQL command `CLUSTER`. Although clustering a table in this way is supported in Greenplum Database, it is not recommended because the `CLUSTER` operation itself is extremely slow.

If you do need to order a table in this way to improve your query performance, Greenplum recommends using a `CREATE TABLE AS` statement to reorder the table on disk rather than using `CLUSTER`. If you do ‘cluster’ a table in this way, then `clusterdb` would not be relevant.

Options

-a | --all

Cluster all databases.

[-d] *dbname* | [--dbname] *dbname*

Specifies the name of the database to be clustered. If this is not specified, the database name is read from the environment variable `PGDATABASE`. If that is not set, the user name specified for the connection is used.

-e | --echo

Echo the commands that `clusterdb` generates and sends to the server.

-q | --quiet

Do not display a response.

-t *table* | --table *table*

Cluster the named table only.

Connection Options

-h *host* | --host *host*

The host name of the machine on which the Greenplum master database server is running. If not specified, reads from the environment variable `PGHOST` or defaults to `localhost`.

-p *port* | --port *port*

The TCP port on which the Greenplum master database server is listening for connections. If not specified, reads from the environment variable `PGPORT` or defaults to `5432`.

-U *username* | --username *username*

The database role name to connect as. If not specified, reads from the environment variable `PGUSER` or defaults to the current system role name.

-W | --password

Force a password prompt.

Examples

To cluster the database *test*:

```
clusterdb test
```

To cluster a single table *foo* in a database named *xyzyz*:

```
clusterdb --table foo xyzyz
```

See Also

[CLUSTER](#)

createdb

Creates a new database.

Synopsis

```
createdb [connection_option ...] [-D tablespace] [-E encoding]  
[-O owner] [-T template] [-e] [-q] [dbname ['description']]
```

Description

`createdb` creates a new database in a Greenplum Database system.

Normally, the database user who executes this command becomes the owner of the new database. However a different owner can be specified via the `-O` option, if the executing user has appropriate privileges.

`createdb` is a wrapper around the SQL command `CREATE DATABASE`.

Options

dbname

The name of the database to be created. The name must be unique among all other databases in the Greenplum system. If not specified, reads from the environment variable `PGDATABASE`, then `PGUSER` or defaults to the current system user.

description

A comment to be associated with the newly created database. Descriptions containing white space must be enclosed in quotes.

-D *tablespace* | **--tablespace** *tablespace*

The default tablespace for the database.

-E *encoding* | **--encoding** *encoding*

Character set encoding to use in the new database. Specify a string constant (such as `'SQL_ASCII'`), an integer encoding number, or `DEFAULT` to use the default encoding. See “[Character Set Support](#)” for the list of supported character sets.

-O *owner* | **--owner** *owner*

The name of the database user who will own the new database. Defaults to the user executing this command.

-T *template* | **--template** *template*

The name of the template from which to create the new database. Defaults to `template1`.

-e | **--echo**

Echo the commands that `createdb` generates and sends to the server.

-q | --quiet

Do not display a response.

Connection Options

-h *host* | --host *host*

The host name of the machine on which the Greenplum master database server is running. If not specified, reads from the environment variable `PGHOST` or defaults to `localhost`.

-p *port* | --port *port*

The TCP port on which the Greenplum master database server is listening for connections. If not specified, reads from the environment variable `PGPORT` or defaults to 5432.

-U *username* | --username *username*

The database role name to connect as. If not specified, reads from the environment variable `PGUSER` or defaults to the current system role name.

-W | --password

Force a password prompt.

Examples

To create the database *test* using the default options:

```
createdb test
```

To create the database *demo* using the Greenplum master on host *gpmaster*, port *54321*, using the *LATINI* encoding scheme:

```
createdb -p 54321 -h gpmaster -E LATIN1 demo
```

See Also

[CREATE DATABASE](#)

createlang

Defines a new procedural language for a database.

Synopsis

```
createlang [connection_option ...] [-e] langname [[-d] dbname]
createlang [connection-option ...] -l dbname
```

Description

`createlang` is a utility for adding a new programming language to a database. `createlang` is a wrapper around the `CREATE LANGUAGE` SQL command.

Options

langname

Specifies the name of the procedural programming language to be defined.

[-d] *dbname* | [--dbname] *dbname*

Specifies to which database the language should be added. The default is to use the `PGDATABASE` environment variable setting, or the same name as the current system user.

-e | --echo

Echo the commands that `createlang` generates and sends to the server.

-l *dbname* | --list *dbname*

Show a list of already installed languages in the target database.

Connection Options

-h *host* | --host *host*

The host name of the machine on which the Greenplum master database server is running. If not specified, reads from the environment variable `PGHOST` or defaults to `localhost`.

-p *port* | --port *port*

The TCP port on which the Greenplum master database server is listening for connections. If not specified, reads from the environment variable `PGPORT` or defaults to 5432.

-U *username* | --username *username*

The database role name to connect as. If not specified, reads from the environment variable `PGUSER` or defaults to the current system role name.

-W | --password

Force a password prompt.

Examples

To install the language *pltcl* into the database *template1*:

```
createlang pltcl template1
```

See Also

[CREATE LANGUAGE](#), [droplang](#)

dropdb

Removes a database.

Synopsis

```
dropdb [connection_option ...] [-e] [-i] [-q] dbname
```

Description

`dropdb` destroys an existing database. The user who executes this command must be a superuser or the owner of the database being dropped.

`dropdb` is a wrapper around the SQL command `DROP DATABASE`

Options

dbname

The name of the database to be removed.

-e | --echo

Echo the commands that `dropdb` generates and sends to the server.

-i | --interactive

Issues a verification prompt before doing anything destructive.

-q | --quiet

Do not display a response.

Connection Options

-h *host* | --host *host*

The host name of the machine on which the Greenplum master database server is running. If not specified, reads from the environment variable `PGHOST` or defaults to `localhost`.

-p *port* | --port *port*

The TCP port on which the Greenplum master database server is listening for connections. If not specified, reads from the environment variable `PGPORT` or defaults to 5432.

-U *username* | --username *username*

The database role name to connect as. If not specified, reads from the environment variable `PGUSER` or defaults to the current system role name.

-W | --password

Force a password prompt.

Examples

To destroy the database named *demo* using default connection parameters:

```
dropdb demo
```

To destroy the database named *demo* using connection options, with verification, and a peek at the underlying command:

```
dropdb -p 54321 -h masterhost -i -e demo
```

```
Database "demo" will be permanently deleted.
```

```
Are you sure? (y/n) y
```

```
DROP DATABASE "demo"
```

```
DROP DATABASE
```

See Also

[DROP DATABASE](#)

droplang

Removes a procedural language.

Synopsis

```
droplang [connection-option ...] [-e] langname [[-d] dbname]
```

```
droplang [connection-option ...] [-e] -l dbname
```

Description

`droplang` is a utility for removing an existing programming language from a database. `droplang` can drop any procedural language, even those not supplied by the Greenplum Database distribution.

Although programming languages can be removed directly using several SQL commands, it is recommended to use `droplang` because it performs a number of checks and is much easier to use.

`droplang` is a wrapper for the SQL command [DROP LANGUAGE](#).

Options

langname

Specifies the name of the programming language to be removed.

[-d] *dbname* | [--dbname] *dbname*

Specifies from which database the language should be removed. The default is to use the `PGDATABASE` environment variable setting, or the same name as the current system user.

-e | --echo

Echo the commands that `droplang` generates and sends to the server.

-l | --list

Show a list of already installed languages in the target database.

Connection Options

-h *host* | --host *host*

The host name of the machine on which the Greenplum master database server is running. If not specified, reads from the environment variable `PGHOST` or defaults to `localhost`.

-p *port* | --port *port*

The TCP port on which the Greenplum master database server is listening for connections. If not specified, reads from the environment variable `PGPORT` or defaults to 5432.

-U *username* | --username *username*

The database role name to connect as. If not specified, reads from the environment variable `PGUSER` or defaults to the current system role name.

-W | --password

Force a password prompt.

Examples

To remove the language *pltcl*:

```
droplang pltcl mydatabase
```

See Also

[DROP LANGUAGE](#)

dropuser

Removes a database role.

Synopsis

```
dropuser [connection_option ...] [-e] [-i] [-q] role_name
```

Description

`dropuser` removes an existing role from Greenplum Database. Only superusers and users with the `CREATEROLE` privilege can remove roles. To remove a superuser role, you must yourself be a superuser.

`dropuser` is a wrapper around the SQL command `DROP ROLE`.

Options

role_name

The name of the role to be removed. You will be prompted for a name if not specified on the command line.

-e | --echo

Echo the commands that `dropuser` generates and sends to the server.

-i | --interactive

Prompt for confirmation before actually removing the role.

-q | --quiet

Do not display a response.

Connection Options

-h *host* | --host *host*

The host name of the machine on which the Greenplum master database server is running. If not specified, reads from the environment variable `PGHOST` or defaults to `localhost`.

-p *port* | --port *port*

The TCP port on which the Greenplum master database server is listening for connections. If not specified, reads from the environment variable `PGPORT` or defaults to 5432.

-U *username* | --username *username*

The database role name to connect as. If not specified, reads from the environment variable `PGUSER` or defaults to the current system role name.

-W | --password

Force a password prompt.

Examples

To remove the role *joe* using default connection options:

```
dropuser joe
DROP ROLE
```

To remove the role *joe* using connection options, with verification, and a peek at the underlying command:

```
dropuser -p 54321 -h masterhost -i -e joe
Role "joe" will be permanently removed.
Are you sure? (y/n) y
DROP ROLE "joe"
DROP ROLE
```

See Also

[DROP ROLE](#)

ecpg

Embedded SQL C preprocessor.

Synopsis

```
ecpg [option ...] file ...
```

Description

`ecpg` is the embedded SQL preprocessor for C programs. It converts C programs with embedded SQL statements to normal C code by replacing the SQL invocations with special function calls. The output files can then be processed with any C compiler tool chain.

`ecpg` will convert each input file given on the command line to the corresponding C output file. Input files preferably have the extension `.pgc`, in which case the extension will be replaced by `.c` to determine the output file name. If the extension of the input file is not `.pgc`, then the output file name is computed by appending `.c` to the full file name. The output file name can also be overridden using the `-o` option.

This reference page does not describe the embedded SQL language. See the [ECPG - Embedded SQL in C](#) chapter of the PostgreSQL documentation for more information.

Options

file

The file to convert.

-c

Automatically generate certain C code from SQL code. Currently, this works for `EXEC SQL TYPE`.

-C mode

Set a compatibility mode. `mode` may be `INFORMIX` or `INFORMIX_SE`.

-D symbol

Define a C preprocessor symbol.

-i

Parse system include files as well.

-I directory

Specify an additional include path, used to find files included via `EXEC SQL INCLUDE`. Defaults are `.` (current directory), `/usr/local/include`, the Greenplum Database include directory (`/usr/local/greenplum-db-3.3.7.x/include`), and `/usr/include`, in that order.

-o filename

Specifies that `ecpg` should write all its output to the given filename.

-r option

Selects a run-time behavior. Currently, option can only be `no_indicator`.

-t

Turn on autocommit of transactions. In this mode, each SQL command is automatically committed unless it is inside an explicit transaction block. In the default mode, commands are committed only when `EXEC SQL COMMIT` is issued.

-v

Print additional information including the version and the include path.

--help

Show a brief summary of the command usage, then exit.

--version

Output version information, then exit.

Examples

If you have an embedded SQL C source file named `prog1.pgc`, you can create an executable program using the following sequence of commands:

```
ecpg prog1.pgc
cc -I/usr/local/pgsql/include -c prog1.c
cc -o prog1 prog1.o -L/usr/local/greenplum-db-3.3.7.x/lib
-llepg
```

pg_config

Retrieves information about the installed version of Greenplum Database.

Synopsis

```
pg_config [option ...]
```

Description

The `pg_config` utility prints configuration parameters of the currently installed version of Greenplum Database. It is intended, for example, to be used by software packages that want to interface to Greenplum Database to facilitate finding the required header files and libraries. Note that information printed out by `pg_config` is for the Greenplum Database master only.

If more than one option is given, the information is printed in that order, one item per line. If no options are given, all available information is printed, with labels.

Options

--bindir

Print the location of user executables. Use this, for example, to find the `psql` program. This is normally also the location where the `pg_config` program resides.

--docdir

Print the location of documentation files.

--includedir

Print the location of C header files of the client interfaces.

--pkgincludedir

Print the location of other C header files.

--includedir-server

Print the location of C header files for server programming.

--libdir

Print the location of object code libraries.

--pkglibdir

Print the location of dynamically loadable modules, or where the server would search for them. (Other architecture-dependent data files may also be installed in this directory.)

--localedir

Print the location of locale support files.

--mandir

Print the location of manual pages.

--shardir

Print the location of architecture-independent support files.

--sysconfdir

Print the location of system-wide configuration files.

--pgxs

Print the location of extension makefiles.

--configure

Print the options that were given to the configure script when Greenplum Database was configured for building.

--cc

Print the value of the CC variable that was used for building Greenplum Database. This shows the C compiler used.

--cppflags

Print the value of the CPPFLAGS variable that was used for building Greenplum Database. This shows C compiler switches needed at preprocessing time.

--cflags

Print the value of the CFLAGS variable that was used for building Greenplum Database. This shows C compiler switches.

--cflags_sl

Print the value of the CFLAGS_SL variable that was used for building Greenplum Database. This shows extra C compiler switches used for building shared libraries.

--ldflags

Print the value of the LDFLAGS variable that was used for building Greenplum Database. This shows linker switches.

--ldflags_sl

Print the value of the LDFLAGS_SL variable that was used for building Greenplum Database. This shows linker switches used for building shared libraries.

--libs

Print the value of the LIBS variable that was used for building Greenplum Database. This normally contains -l switches for external libraries linked into Greenplum Database.

--version

Print the version of Greenplum Database.

Examples

To reproduce the build configuration of the current Greenplum Database installation, run the following command:

```
eval ./configure 'pg_config --configure'
```

The output of `pg_config --configure` contains shell quotation marks so arguments with spaces are represented correctly. Therefore, using `eval` is required for proper results.

pg_dump

Extracts a database into a single script file or other archive file.

Synopsis

```
pg_dump [connection_option ...] [dump_option ...] dbname
```

Description

`pg_dump` is a standard PostgreSQL utility for backing up a database, and is also supported in Greenplum Database. It creates a single (non-parallel) dump file. For routine backups of Greenplum Database it is better to use Greenplum's parallel dump utility, `gp_dump`, for the best performance.

Use `pg_dump` if you are migrating your data to another database vendor's system, or to another Greenplum Database system with a different segment configuration (for example, if the system you are migrating to has greater or fewer segment instances). To restore, you must use the corresponding `pg_restore` utility (if the dump file is in archive format), or you can use a client program such as `psql` (if the dump file is in plain text format).

Since `pg_dump` is compatible with regular PostgreSQL, it can be used to migrate data into Greenplum Database. The `pg_dump` utility in Greenplum Database is very similar to the PostgreSQL `pg_dump` utility, with the following exceptions and limitations:

- If using `pg_dump` to backup a Greenplum database, keep in mind that the dump operation can take a long time (several hours) for very large databases. Also, you must make sure you have sufficient disk space to create the dump file.
- If you are migrating data from one Greenplum Database system to another, use the `--gp-syntax` command-line option to include the `DISTRIBUTED BY` clause in `CREATE TABLE` statements. This ensures that Greenplum Database table data is distributed with the correct distribution key columns upon restore.

`pg_dump` makes consistent backups even if the database is being used concurrently. `pg_dump` does not block other users accessing the database (readers or writers).

When used with one of the archive file formats and combined with `pg_restore`, `pg_dump` provides a flexible archival and transfer mechanism. `pg_dump` can be used to backup an entire database, then `pg_restore` can be used to examine the archive and/or select which parts of the database are to be restored. The most flexible output file format is the `custom` format (`-Fc`). It allows for selection and reordering of all archived items, and is compressed by default. The `tar` format (`-Ft`) is not compressed and it is not possible to reorder data when loading, but it is otherwise quite flexible. It can be manipulated with standard Unix tools such as `tar`.

Options

dbname

Specifies the name of the database to be dumped. If this is not specified, the environment variable `PGDATABASE` is used. If that is not set, the user name specified for the connection is used.

Dump Options

-a | --data-only

Dump only the data, not the schema (data definitions). This option is only meaningful for the plain-text format. For the archive formats, you may specify the option when you call `pg_restore`.

-b | --blobs

Include large objects in the dump. This is the default behavior except when `--schema`, `--table`, or `--schema-only` is specified, so the `-b` switch is only useful to add large objects to selective dumps.

-c | --clean

Adds commands to the text output file to clean (drop) database objects prior to (the commands for) creating them. This option is only meaningful for the plain-text format. For the archive formats, you may specify the option when you call `pg_restore`.

-C | --create

Begin the output with a command to create the database itself and reconnect to the created database. (With a script of this form, it doesn't matter which database you connect to before running the script.) This option is only meaningful for the plain-text format. For the archive formats, you may specify the option when you call `pg_restore`.

-d | --inserts

Dump data as `INSERT` commands (rather than `COPY`). This will make restoration very slow; it is mainly useful for making dumps that can be loaded into non-PostgreSQL-based databases. Also, since this option generates a separate command for each row, an error in reloading a row causes only that row to be lost rather than the entire table contents. Note that the restore may fail altogether if you have rearranged column order. The `-D` option is safe against column order changes, though even slower.

-D | --column-inserts | --attribute-inserts

Dump data as `INSERT` commands with explicit column names (`INSERT INTO table (column, ...) VALUES ...`). This will make restoration very slow; it is mainly useful for making dumps that can be loaded into non-PostgreSQL-based databases. Also, since this option generates a separate command for each row, an error in reloading a row causes only that row to be lost rather than the entire table contents.

-E encoding | --encoding=encoding

Create the dump in the specified character set encoding. By default, the dump is created in the database encoding. (Another way to get the same result is to set the `PGCLIENTENCODING` environment variable to the desired dump encoding.)

-f file | --file=file

Send output to the specified file. If this is omitted, the standard output is used.

-F p|c|t | --format=plain|custom|tar

Selects the format of the output. format can be one of the following:

p | plain — Output a plain-text SQL script file (the default).

c | custom — Output a custom archive suitable for input into `pg_restore`. This is the most flexible format in that it allows reordering of loading data as well as object definitions. This format is also compressed by default.

t | tar — Output a tar archive suitable for input into `pg_restore`. Using this archive format allows reordering and/or exclusion of database objects at the time the database is restored. It is also possible to limit which data is reloaded at restore time.

-i | --ignore-version

Ignore version mismatch between `pg_dump` and the database server. `pg_dump` can dump from servers running previous releases of Greenplum Database (or PostgreSQL), but very old versions may not be supported anymore. Use this option if you need to override the version check.

-n schema | --schema=schema

Dump only schemas matching the schema pattern; this selects both the schema itself, and all its contained objects. When this option is not specified, all non-system schemas in the target database will be dumped. Multiple schemas can be selected by writing multiple `-n` switches. Also, the schema parameter is interpreted as a pattern according to the same rules used by `psql`'s `\d` commands, so multiple schemas can also be selected by writing wildcard characters in the pattern. When using wildcards, be careful to quote the pattern if needed to prevent the shell from expanding the wildcards.

Note: When `-n` is specified, `pg_dump` makes no attempt to dump any other database objects that the selected schema(s) may depend upon. Therefore, there is no guarantee that the results of a specific-schema dump can be successfully restored by themselves into a clean database.

Note: Non-schema objects such as blobs are not dumped when `-n` is specified. You can add blobs back to the dump with the `--blobs` switch.

-N schema | --exclude-schema=schema

Do not dump any schemas matching the schema pattern. The pattern is interpreted according to the same rules as for `-n`. `-N` can be given more than once to exclude schemas matching any of several patterns. When both `-n` and `-N` are given, the

behavior is to dump just the schemas that match at least one `-n` switch but no `-N` switches. If `-N` appears without `-n`, then schemas matching `-N` are excluded from what is otherwise a normal dump.

-o | --oids

Dump object identifiers (OIDs) as part of the data for every table. Use of this option is not recommended for files that are intended to be restored into Greenplum Database.

-O | --no-owner

Do not output commands to set ownership of objects to match the original database. By default, `pg_dump` issues `ALTER OWNER` or `SET SESSION AUTHORIZATION` statements to set ownership of created database objects. These statements will fail when the script is run unless it is started by a superuser (or the same user that owns all of the objects in the script). To make a script that can be restored by any user, but will give that user ownership of all the objects, specify `-O`. This option is only meaningful for the plain-text format. For the archive formats, you may specify the option when you call `pg_restore`.

-s | --schema-only

Dump only the object definitions (schema), not data.

-S *username* | --superuser=*username*

Specify the superuser user name to use when disabling triggers. This is only relevant if `--disable-triggers` is used. It is better to leave this out, and instead start the resulting script as a superuser.

-t *table* | --table=*table*

Dump only tables (or views or sequences) matching the table pattern. Multiple tables can be selected by writing multiple `-t` switches. Also, the table parameter is interpreted as a pattern according to the same rules used by `psql`'s `\d` commands, so multiple tables can also be selected by writing wildcard characters in the pattern. When using wildcards, be careful to quote the pattern if needed to prevent the shell from expanding the wildcards. The `-n` and `-N` switches have no effect when `-t` is used, because tables selected by `-t` will be dumped regardless of those switches, and non-table objects will not be dumped.

Note: When `-t` is specified, `pg_dump` makes no attempt to dump any other database objects that the selected table(s) may depend upon. Therefore, there is no guarantee that the results of a specific-table dump can be successfully restored by themselves into a clean database.

Note: `-t` cannot be used to specify a child table partition. To dump a partitioned table, you must specify the parent table name.

-T *table* | --exclude-table=*table*

Do not dump any tables matching the table pattern. The pattern is interpreted according to the same rules as for `-t`. `-T` can be given more than once to exclude tables matching any of several patterns. When both `-t` and `-T` are given, the

behavior is to dump just the tables that match at least one `-t` switch but no `-T` switches. If `-T` appears without `-t`, then tables matching `-T` are excluded from what is otherwise a normal dump.

`-v` | `--verbose`

Specifies verbose mode. This will cause `pg_dump` to output detailed object comments and start/stop times to the dump file, and progress messages to standard error.

`-x` | `--no-privileges` | `--no-acl`

Prevent dumping of access privileges (`GRANT/REVOKE` commands).

`--disable-dollar-quoting`

This option disables the use of dollar quoting for function bodies, and forces them to be quoted using SQL standard string syntax.

`--disable-triggers`

This option is only relevant when creating a data-only dump. It instructs `pg_dump` to include commands to temporarily disable triggers on the target tables while the data is reloaded. Use this if you have triggers on the tables that you do not want to invoke during data reload. The commands emitted for `--disable-triggers` must be done as superuser. So, you should also specify a superuser name with `-S`, or preferably be careful to start the resulting script as a superuser. This option is only meaningful for the plain-text format. For the archive formats, you may specify the option when you call `pg_restore`.

`--use-set-session-authorization`

Output SQL-standard `SET SESSION AUTHORIZATION` commands instead of `ALTER OWNER` commands to determine object ownership. This makes the dump more standards compatible, but depending on the history of the objects in the dump, may not restore properly. A dump using `SET SESSION AUTHORIZATION` will require superuser privileges to restore correctly, whereas `ALTER OWNER` requires lesser privileges.

`--gp-syntax` | `--no-gp-syntax`

Use `--gp-syntax` to dump Greenplum Database syntax in the `CREATE TABLE` statements. This allows the distribution policy (`DISTRIBUTED BY` or `DISTRIBUTED RANDOMLY` clauses) of a Greenplum Database table to be dumped, which is useful for restoring into other Greenplum Database systems. The default is to include Greenplum Database syntax when connected to a Greenplum system, and to exclude it when connected to a regular PostgreSQL system.

`-Z 0..9` | `--compress=0..9`

Specify the compression level to use in archive formats that support compression. Currently only the *custom* archive format supports compression.

Connection Options

-h host | --host host

The host name of the machine on which the Greenplum master database server is running. If not specified, reads from the environment variable `PGHOST` or defaults to `localhost`.

-p port | --port port

The TCP port on which the Greenplum master database server is listening for connections. If not specified, reads from the environment variable `PGPORT` or defaults to `5432`.

-U username | --username username

The database role name to connect as. If not specified, reads from the environment variable `PGUSER` or defaults to the current system role name.

-W | --password

Force a password prompt.

Notes

When a data-only dump is chosen and the option `--disable-triggers` is used, `pg_dump` emits commands to disable triggers on user tables before inserting the data and commands to re-enable them after the data has been inserted. If the restore is stopped in the middle, the system catalogs may be left in the wrong state.

Members of `tar` archives are limited to a size less than 8 GB. (This is an inherent limitation of the `tar` file format.) Therefore this format cannot be used if the textual representation of any one table exceeds that size. The total size of a `tar` archive and any of the other output formats is not limited, except possibly by the operating system.

The dump file produced by `pg_dump` does not contain the statistics used by the optimizer to make query planning decisions. Therefore, it is wise to run `ANALYZE` after restoring from a dump file to ensure good performance.

Examples

Dump a database called `mydb` into a SQL-script file:

```
pg_dump mydb > db.sql
```

To reload such a script into a (freshly created) database named `newdb`:

```
psql -d newdb -f db.sql
```

Dump a Greenplum database in `tar` file format and include distribution policy information:

```
pg_dump -Ft --gp-syntax mydb > db.tar
```

To dump a database into a custom-format archive file:

```
pg_dump -Fc mydb > db.dump
```

To reload an archive file into a (freshly created) database named *newdb*:

```
pg_restore -d newdb db.dump
```

To dump a single table named *mytab*:

```
pg_dump -t mytab mydb > db.sql
```

To specify an upper-case or mixed-case name in `-t` and related switches, you need to double-quote the name; else it will be folded to lower case. But double quotes are special to the shell, so in turn they must be quoted. Thus, to dump a single table with a mixed-case name, you need something like:

```
pg_dump -t '"MixedCaseName"' mydb > mytab.sql
```

See Also

[gp_dump](#), [pg_dumpall](#), [pg_restore](#), [gp_restore](#), [psql](#)

createuser

Creates a new database role.

Synopsis

```
createuser [connection_option ...] [role_attribute ...] [-e] [-q]
role_name
```

Description

`createuser` creates a new Greenplum Database role. Only superusers and users with `CREATEROLE` privilege can create new roles, so `createuser` must be invoked by someone who can connect as a superuser or a role with `CREATEROLE` privilege.

If you wish to create a new superuser, you must connect as a superuser, not merely with `CREATEROLE` privilege. Being a superuser implies the ability to bypass all access permission checks within the database, so superuser privileges should not be granted lightly.

`createuser` is a wrapper around the SQL command `CREATE ROLE`.

Options

role_name

The name of the role to be created. This name must be different from all existing roles in this Greenplum Database installation.

-s | --superuser

The new role will be a superuser.

-S | --no-superuser

The new role will not be a superuser. This is the default.

-d | --createdb

The new role will be allowed to create databases.

-D | --no-createdb

The new role will not be allowed to create databases. This is the default.

-r | --createrole

The new role will be allowed to create new roles (`CREATEROLE` privilege).

-R | --no-createrole

The new role will not be allowed to create new roles. This is the default.

-l | --login

The new role will be allowed to log in to Greenplum Database. This is the default.

-L | --no-login

The new role will not be allowed to log in (a group-level role).

-i | --inherit

The new role will automatically inherit privileges of roles it is a member of. This is the default.

-I | --no-inherit

The new role will not automatically inherit privileges of roles it is a member of.

-c *number* | --connection-limit *number*

Set a maximum number of connections for the new role. The default is to set no limit.

-P | --pwprompt

If given, `createuser` will issue a prompt for the password of the new role. This is not necessary if you do not plan on using password authentication.

-E | --encrypted

Encrypts the role's password stored in the database. If not specified, the default password behavior is used.

-N | --unencrypted

Does not encrypt the role's password stored in the database. If not specified, the default password behavior is used.

-e | --echo

Echo the commands that `createuser` generates and sends to the server.

-q | --quiet

Do not display a response.

Connection Options**-h *host* | --host *host***

The host name of the machine on which the Greenplum master database server is running. If not specified, reads from the environment variable `PGHOST` or defaults to `localhost`.

-p *port* | --port *port*

The TCP port on which the Greenplum master database server is listening for connections. If not specified, reads from the environment variable `PGPORT` or defaults to 5432.

-U *username* | --username *username*

The database role name to connect as. If not specified, reads from the environment variable `PGUSER` or defaults to the current system role name.

-W | --password

Force a password prompt.

Examples

Create a role named *joe* using the default options:

```
createuser joe
Shall the new role be a superuser? (y/n) n
Shall the new role be allowed to create databases? (y/n) n
Shall the new role be allowed to create more new roles? (y/n)
n
CREATE ROLE
```

To create the same role *joe* using connection options and avoiding the prompts and taking a look at the underlying command:

```
createuser -h masterhost -p 54321 -S -D -R -e joe
CREATE ROLE joe NOSUPERUSER NOCREATEDB NOCREATEROLE INHERIT
LOGIN;
CREATE ROLE
```

To create the role *joe* as a superuser, and assign a password immediately:

```
createuser -P -s -e joe
Enter password for new role: admin123
Enter it again: admin123
CREATE ROLE joe PASSWORD 'admin123' SUPERUSER CREATEDB
CREATEROLE INHERIT LOGIN;
CREATE ROLE
```

In the above example, the new password is not actually echoed when typed, but we show what was typed for clarity. However the password will appear in the echoed command, as illustrated if the `-e` option is used.

See Also

[CREATE ROLE](#)

pg_dumpall

Extracts all databases in a Greenplum Database system to a single script file or other archive file.

Synopsis

```
pg_dumpall [connection_option ...] [dump_option ...]
```

Description

`pg_dumpall` is a standard PostgreSQL utility for backing up all databases in a Greenplum Database (or PostgreSQL) instance, and is also supported in Greenplum Database. It creates a single (non-parallel) dump file. For routine backups of Greenplum Database it is better to use Greenplum's parallel dump utility, `gp_dump`, for the best performance.

`pg_dumpall` creates a single script file that contains SQL commands that can be used as input to `psql` to restore the databases. It does this by calling `pg_dump` for each database. `pg_dumpall` also dumps global objects that are common to all databases. (`pg_dump` does not save these objects.) This currently includes information about database users and groups, and access permissions that apply to databases as a whole.

Since `pg_dumpall` reads tables from all databases you will most likely have to connect as a database superuser in order to produce a complete dump. Also you will need superuser privileges to execute the saved script in order to be allowed to add users and groups, and to create databases.

The SQL script will be written to the standard output. Shell operators should be used to redirect it into a file.

`pg_dumpall` needs to connect several times to the Greenplum Database master server (once per database). If you use password authentication it is likely to ask for a password each time. It is convenient to have a `~/ .pgpass` file in such cases.

Options

Dump Options

-a | --data-only

Dump only the data, not the schema (data definitions). This option is only meaningful for the plain-text format. For the archive formats, you may specify the option when you call `pg_restore`.

-c | --clean

Output commands to clean (drop) database objects prior to (the commands for) creating them. This option is only meaningful for the plain-text format. For the archive formats, you may specify the option when you call `pg_restore`.

-d | --inserts

Dump data as `INSERT` commands (rather than `COPY`). This will make restoration very slow; it is mainly useful for making dumps that can be loaded into non-PostgreSQL-based databases. Also, since this option generates a separate command for each row, an error in reloading a row causes only that row to be lost rather than the entire table contents. Note that the restore may fail altogether if you have rearranged column order. The `-D` option is safe against column order changes, though even slower.

-D | --column-inserts | --attribute-inserts

Dump data as `INSERT` commands with explicit column names (`INSERT INTO table (column, ...) VALUES ...`). This will make restoration very slow; it is mainly useful for making dumps that can be loaded into non-PostgreSQL-based databases. Also, since this option generates a separate command for each row, an error in reloading a row causes only that row to be lost rather than the entire table contents.

-g | --globals-only

Dump only global objects (roles and tablespaces), no databases.

-i | --ignore-version

Ignore version mismatch between `pg_dump` and the database server. `pg_dump` can dump from servers running previous releases of Greenplum Database (or PostgreSQL), but very old versions may not be supported anymore. Use this option if you need to override the version check.

-o | --oids

Dump object identifiers (OIDs) as part of the data for every table. Use of this option is not recommended for files that are intended to be restored into Greenplum Database.

-O | --no-owner

Do not output commands to set ownership of objects to match the original database. By default, `pg_dump` issues `ALTER OWNER` or `SET SESSION AUTHORIZATION` statements to set ownership of created database objects. These statements will fail when the script is run unless it is started by a superuser (or the same user that owns all of the objects in the script). To make a script that can be restored by any user, but will give that user ownership of all the objects, specify `-O`. This option is only meaningful for the plain-text format. For the archive formats, you may specify the option when you call `pg_restore`.

-s | --schema-only

Dump only the object definitions (schema), not data.

-S *username* | --superuser=*username*

Specify the superuser user name to use when disabling triggers. This is only relevant if `--disable-triggers` is used. It is better to leave this out, and instead start the resulting script as a superuser.

-v | --verbose

Specifies verbose mode. This will cause `pg_dump` to output detailed object comments and start/stop times to the dump file, and progress messages to standard error.

-x | --no-privileges | --no-acl

Prevent dumping of access privileges (`GRANT/REVOKE` commands).

--disable-dollar-quoting

This option disables the use of dollar quoting for function bodies, and forces them to be quoted using SQL standard string syntax.

--disable-triggers

This option is only relevant when creating a data-only dump. It instructs `pg_dumpall` to include commands to temporarily disable triggers on the target tables while the data is reloaded. Use this if you have triggers on the tables that you do not want to invoke during data reload. The commands emitted for `--disable-triggers` must be done as superuser. So, you should also specify a superuser name with `-s`, or preferably be careful to start the resulting script as a superuser.

--use-set-session-authorization

Output SQL-standard `SET SESSION AUTHORIZATION` commands instead of `ALTER OWNER` commands to determine object ownership. This makes the dump more standards compatible, but depending on the history of the objects in the dump, may not restore properly. A dump using `SET SESSION AUTHORIZATION` will require superuser privileges to restore correctly, whereas `ALTER OWNER` requires lesser privileges.

--gp-syntax

Output Greenplum Database syntax in the `CREATE TABLE` statements. This allows the distribution policy (`DISTRIBUTED BY` or `DISTRIBUTED RANDOMLY` clauses) of a Greenplum Database table to be dumped, which is useful for restoring into other Greenplum Database systems.

Connection Options**-h *host* | --host *host***

The host name of the machine on which the Greenplum master database server is running. If not specified, reads from the environment variable `PGHOST` or defaults to `localhost`.

-p *port* | --port *port*

The TCP port on which the Greenplum master database server is listening for connections. If not specified, reads from the environment variable `PGPORT` or defaults to 5432.

-U *username* | --username *username*

The database role name to connect as. If not specified, reads from the environment variable `PGUSER` or defaults to the current system role name.

-W | --password

Force a password prompt.

Notes

Since `pg_dumpall` calls `pg_dump` internally, some diagnostic messages will refer to `pg_dump`.

Once restored, it is wise to run `ANALYZE` on each database so the query planner has useful statistics. You can also run `vacuumdb -a -z` to analyze all databases.

`pg_dumpall` requires all needed tablespace directories to exist before the restore or database creation will fail for databases in non-default locations.

Examples

To dump all databases:

```
pg_dumpall > db.out
```

To reload this file:

```
psql template1 -f db.out
```

To dump only global objects:

```
pg_dumpall -g
```

See Also

[gp_dump](#), [pg_dump](#)

pg_restore

Restores a database from an archive file created by `pg_dump`.

Synopsis

```
pg_restore [connection_option ...] [restore_option ...] filename
```

Description

`pg_restore` is a utility for restoring a database from an archive created by `pg_dump` in one of the non-plain-text formats. It will issue the commands necessary to reconstruct the database to the state it was in at the time it was saved. The archive files also allow `pg_restore` to be selective about what is restored, or even to reorder the items prior to being restored.

`pg_restore` can operate in two modes. If a database name is specified, the archive is restored directly into the database. Otherwise, a script containing the SQL commands necessary to rebuild the database is created and written to a file or standard output. The script output is equivalent to the plain text output format of `pg_dump`. Some of the options controlling the output are therefore analogous to `pg_dump` options.

`pg_restore` cannot restore information that is not present in the archive file. For instance, if the archive was made using the “dump data as `INSERT` commands” option, `pg_restore` will not be able to load the data using `COPY` statements.

Options

filename

Specifies the location of the archive file to be restored. If not specified, the standard input is used.

-a | --data-only

Restore only the data, not the schema (data definitions).

-c | --clean

Clean (drop) database objects before recreating them.

-C | --create

Create the database before restoring into it. (When this option is used, the database named with `-d` is used only to issue the initial `CREATE DATABASE` command. All data is restored into the database name that appears in the archive.)

-d dbname | --dbname=dbname

Connect to this database and restore directly into this database. The default is to use the `PGDATABASE` environment variable setting, or the same name as the current system user.

-e | --exit-on-error

Exit if an error is encountered while sending SQL commands to the database. The default is to continue and to display a count of errors at the end of the restoration.

-f *outfile* | --file=*outfile*

Specify output file for generated script, or for the listing when used with `-l`. Default is the standard output.

-F *t|c* | --format=*tar|custom*

The format of the archive produced by `pg_dump`. It is not necessary to specify the format, since `pg_restore` will determine the format automatically. Format can be either `tar` or `custom`.

-i | --ignore-version

Ignore database version checks.

-I *index* | --index=*index*

Restore definition of named index only.

-l | --list

List the contents of the archive. The output of this operation can be used with the `-L` option to restrict and reorder the items that are restored.

-L *list-file* | --use-list=*list-file*

Restore elements in the *list-file* only, and in the order they appear in the file. Lines can be moved and may also be commented out by placing a `;` at the start of the line.

-n *schema* | --schema=*schema*

Restore only objects that are in the named schema. This can be combined with the `-t` option to restore just a specific table.

-O | --no-owner

Do not output commands to set ownership of objects to match the original database. By default, `pg_restore` issues `ALTER OWNER` or `SET SESSION AUTHORIZATION` statements to set ownership of created schema elements. These statements will fail unless the initial connection to the database is made by a superuser (or the same user that owns all of the objects in the script). With `-O`, any user name can be used for the initial connection, and this user will own all the created objects.

**-P *function-name*(*argtype* [, ...]) |
--function=*function-name*(*argtype* [, ...])**

Restore the named function only. Be careful to spell the function name and arguments exactly as they appear in the dump file's table of contents.

-s | --schema-only

Restore only the schema (data definitions), not the data (table contents). Sequence current values will not be restored, either. (Do not confuse this with the `--schema` option, which uses the word `schema` in a different meaning.)

-S *username* | --superuser=*username*

Specify the superuser user name to use when disabling triggers. This is only relevant if `--disable-triggers` is used.

-t *table* | --table=*table*

Restore definition and/or data of named table only.

-T *trigger* | --trigger=*trigger*

Restore named trigger only.

-v | --verbose

Specifies verbose mode.

-x | --no-privileges | --no-acl

Prevent restoration of access privileges (`GRANT/REVOKE` commands).

--disable-triggers

This option is only relevant when performing a data-only restore. It instructs `pg_restore` to execute commands to temporarily disable triggers on the target tables while the data is reloaded. Use this if you have triggers on the tables that you do not want to invoke during data reload. The commands emitted for `--disable-triggers` must be done as superuser. So, you should also specify a superuser name with `-S`, or preferably run `pg_restore` as a superuser.

--no-data-for-failed-tables

By default, table data is restored even if the creation command for the table failed (e.g., because it already exists). With this option, data for such a table is skipped. This behavior is useful when the target database may already contain the desired table contents. Specifying this option prevents duplicate or obsolete data from being loaded. This option is effective only when restoring directly into a database, not when producing SQL script output.

Connection Options**-h *host* | --host *host***

The host name of the machine on which the Greenplum master database server is running. If not specified, reads from the environment variable `PGHOST` or defaults to `localhost`.

-p *port* | --port *port*

The TCP port on which the Greenplum master database server is listening for connections. If not specified, reads from the environment variable `PGPORT` or defaults to `5432`.

-U *username* | --username *username*

The database role name to connect as. If not specified, reads from the environment variable `PGUSER` or defaults to the current system role name.

-W | --password

Force a password prompt.

-1 | --single-transaction

Execute the restore as a single transaction. This ensures that either all the commands complete successfully, or no changes are applied.

Notes

If your installation has any local additions to the `template1` database, be careful to load the output of `pg_restore` into a truly empty database; otherwise you are likely to get errors due to duplicate definitions of the added objects. To make an empty database without any local additions, copy from `template0` not `template1`, for example:

```
CREATE DATABASE foo WITH TEMPLATE template0;
```

When restoring data to a pre-existing table and the option `--disable-triggers` is used, `pg_restore` emits commands to disable triggers on user tables before inserting the data then emits commands to re-enable them after the data has been inserted. If the restore is stopped in the middle, the system catalogs may be left in the wrong state.

`pg_restore` will not restore large objects for a single table. If an archive contains large objects, then all large objects will be restored.

See also the `pg_dump` documentation for details on limitations of `pg_dump`.

Once restored, it is wise to run `ANALYZE` on each restored table so the query planner has useful statistics.

Examples

Assume we have dumped a database called `mydb` into a custom-format dump file:

```
pg_dump -Fc mydb > db.dump
```

To drop the database and recreate it from the dump:

```
dropdb mydb
pg_restore -C -d template1 db.dump
```

To reload the dump into a new database called `newdb`. Notice there is no `-C`, we instead connect directly to the database to be restored into. Also note that we clone the new database from `template0` not `template1`, to ensure it is initially empty:

```
createdb -T template0 newdb
pg_restore -d newdb db.dump
```

To reorder database items, it is first necessary to dump the table of contents of the archive:

```
pg_restore -l db.dump > db.list
```

The listing file consists of a header and one line for each item, for example,

```
; Archive created at Fri Jul 28 22:28:36 2006
;   dbname: mydb
;   TOC Entries: 74
;   Compression: 0
;   Dump Version: 1.4-0
;   Format: CUSTOM
;
; Selected TOC Entries:
;
2; 145344 TABLE species postgres
3; 145344 ACL species
4; 145359 TABLE nt_header postgres
5; 145359 ACL nt_header
6; 145402 TABLE species_records postgres
7; 145402 ACL species_records
8; 145416 TABLE ss_old postgres
9; 145416 ACL ss_old
10; 145433 TABLE map_resolutions postgres
11; 145433 ACL map_resolutions
12; 145443 TABLE hs_old postgres
13; 145443 ACL hs_old
```

Semicolons start a comment, and the numbers at the start of lines refer to the internal archive ID assigned to each item. Lines in the file can be commented out, deleted, and reordered. For example,

```
10; 145433 TABLE map_resolutions postgres
;2; 145344 TABLE species postgres
;4; 145359 TABLE nt_header postgres
6; 145402 TABLE species_records postgres
;8; 145416 TABLE ss_old postgres
```

Could be used as input to `pg_restore` and would only restore items 10 and 6, in that order:

```
pg_restore -L db.list db.dump
```

See Also

[pg_dump](#), [gp_restore](#), [gp_dump](#)

psql

Interactive command-line interface for Greenplum Database

Synopsis

```
psql [option...] [dbname [username]]
```

Description

`psql` is a terminal-based front-end to Greenplum Database. It enables you to type in queries interactively, issue them to Greenplum Database, and see the query results. Alternatively, input can be from a file. In addition, it provides a number of meta-commands and various shell-like features to facilitate writing scripts and automating a wide variety of tasks.

Options

-a | --echo-all

Print all input lines to standard output as they are read. This is more useful for script processing rather than interactive mode.

-A | --no-align

Switches to unaligned output mode. (The default output mode is aligned.)

-c 'command' | --command 'command'

Specifies that `psql` is to execute the specified command string, and then exit. This is useful in shell scripts. *command* must be either a command string that is completely parseable by the server, or a single backslash command. Thus you cannot mix SQL and `psql` meta-commands with this option. To achieve that, you could pipe the string into `psql`, like this: `echo '\x \ SELECT * FROM foo;' | psql.` (`\` is the separator meta-command.)

If the command string contains multiple SQL commands, they are processed in a single transaction, unless there are explicit `BEGIN/COMMIT` commands included in the string to divide it into multiple transactions. This is different from the behavior when the same string is fed to `psql`'s standard input.

-d dbname | --dbname dbname

Specifies the name of the database to connect to. This is equivalent to specifying `dbname` as the first non-option argument on the command line.

-e | --echo-queries

Copy all SQL commands sent to the server to standard output as well.

-E | --echo-hidden

Echo the actual queries generated by `\d` and other backslash commands. You can use this to study `psql`'s internal operations.

-f filename | --file filename

Use a file as the source of commands instead of reading commands interactively. After the file is processed, `psql` terminates. If *filename* is `-` (hyphen), then standard input is read. Using this option is subtly different from writing `psql < filename`. In general, both will do what you expect, but using `-f` enables some nice features such as error messages with line numbers.

-F separator | --field-separator separator

Use the specified separator as the field separator for unaligned output.

-H | --html

Turn on HTML tabular output.

-l | --list

List all available databases, then exit. Other non-connection options are ignored.

-L filename | --log-file filename

Write all query output into the specified log file, in addition to the normal output destination.

-o filename | --output filename

Put all query output into the specified file.

-P assignment | --pset assignment

Allows you to specify printing options in the style of `\pset` on the command line. Note that here you have to separate name and value with an equal sign instead of a space. Thus to set the output format to LaTeX, you could write `-P format=latex`.

-q | --quiet

Specifies that `psql` should do its work quietly. By default, it prints welcome messages and various informational output. If this option is used, none of this happens. This is useful with the `-c` option.

-R separator | --record-separator separator

Use *separator* as the record separator for unaligned output.

-s | --single-step

Run in single-step mode. That means the user is prompted before each command is sent to the server, with the option to cancel execution as well. Use this to debug scripts.

-S | --single-line

Runs in single-line mode where a new line terminates an SQL command, as a semicolon does.

-t | --tuples-only

Turn off printing of column names and result row count footers, etc.

-T *table_options* | --table-attr *table_options*

Allows you to specify options to be placed within the HTML table tag. See `\pset` for details.

-v *assignment* | --set *assignment* | --variable *assignment*

Perform a variable assignment, like the `\set` internal command. Note that you must separate name and value, if any, by an equal sign on the command line. To unset a variable, leave off the equal sign. To just set a variable without a value, use the equal sign but leave off the value. These assignments are done during a very early stage of start-up, so variables reserved for internal purposes might get overwritten later.

-V | --version

Print the `psql` version and exit.

-x | --expanded

Turn on the expanded table formatting mode.

-X | --no-psqlrc

Do not read the start-up file (neither the system-wide `psqlrc` file nor the user's `~/.psqlrc` file).

-1 | --single-transaction

When `psql` executes a script with the `-f` option, adding this option wraps `BEGIN/COMMIT` around the script to execute it as a single transaction. This ensures that either all the commands complete successfully, or no changes are applied.

If the script itself uses `BEGIN`, `COMMIT`, or `ROLLBACK`, this option will not have the desired effects. Also, if the script contains any command that cannot be executed inside a transaction block, specifying this option will cause that command (and hence the whole transaction) to fail.

-? | --help

Show help about `psql` command line arguments, and exit.

Connection Options**-h *host* | --host *host***

The host name of the machine on which the Greenplum master database server is running. If not specified, reads from the environment variable `PGHOST` or defaults to `localhost`.

-p *port* | --port *port*

The TCP port on which the Greenplum master database server is listening for connections. If not specified, reads from the environment variable `PGPORT` or defaults to 5432.

-U *username* | --username *username*

The database role name to connect as. If not specified, reads from the environment variable `PGUSER` or defaults to the current system role name.

-W | --password

Force a password prompt. `psql` should automatically prompt for a password whenever the server requests password authentication. However, currently password request detection is not totally reliable, hence this option to force a prompt. If no password prompt is issued and the server requires password authentication, the connection attempt will fail.

Exit Status

`psql` returns 0 to the shell if it finished normally, 1 if a fatal error of its own (out of memory, file not found) occurs, 2 if the connection to the server went bad and the session was not interactive, and 3 if an error occurred in a script and the variable `ON_ERROR_STOP` was set.

Usage

Connecting To A Database

`psql` is a client application for Greenplum Database. In order to connect to a database you need to know the name of your target database, the host name and port number of the Greenplum master server and what database user name you want to connect as. `psql` can be told about those parameters via command line options, namely `-d`, `-h`, `-p`, and `-U` respectively. If an argument is found that does not belong to any option it will be interpreted as the database name (or the user name, if the database name is already given). Not all these options are required; there are useful defaults. If you omit the host name, `psql` will connect via a Unix-domain socket to a master server on the local host, or via TCP/IP to `localhost` on machines that do not have Unix-domain sockets. The default master port number is 5432. If you use a different port for the master, you must specify the port. The default database user name is your Unix user name, as is the default database name. Note that you cannot just connect to any database under any user name. Your database administrator should have informed you about your access rights.

When the defaults are not right, you can save yourself some typing by setting the environment variables `PGDATABASE`, `PGHOST`, `PGPORT` and/or `PGUSER` to appropriate values.

It is also convenient to have a `~/.pgpass` file to avoid regularly having to type in passwords. This file should reside in your home directory and contain lines of the following format:

```
hostname:port:database:username:password
```

The permissions on `.pgpass` must disallow any access to world or group (for example: `chmod 0600 ~/.pgpass`). If the permissions are less strict than this, the file will be ignored. (The file permissions are not currently checked on Microsoft Windows clients, however.)

If the connection could not be made for any reason (insufficient privileges, server is not running, etc.), `psql` will return an error and terminate.

Entering SQL Commands

In normal operation, `psql` provides a prompt with the name of the database to which `psql` is currently connected, followed by the string `=>` for a regular user or `=#` for a superuser. For example:

```
testdb=>
testdb=#
```

At the prompt, the user may type in SQL commands. Ordinarily, input lines are sent to the server when a command-terminating semicolon is reached. An end of line does not terminate a command. Thus commands can be spread over several lines for clarity. If the command was sent and executed without error, the results of the command are displayed on the screen.

Meta-Commands

Anything you enter in `psql` that begins with an unquoted backslash is a `psql` meta-command that is processed by `psql` itself. These commands help make `psql` more useful for administration or scripting. Meta-commands are more commonly called slash or backslash commands.

The format of a `psql` command is the backslash, followed immediately by a command verb, then any arguments. The arguments are separated from the command verb and each other by any number of whitespace characters.

To include whitespace into an argument you may quote it with a single quote. To include a single quote into such an argument, use two single quotes. Anything contained in single quotes is furthermore subject to C-like substitutions for `\n` (new line), `\t` (tab), `\digits` (octal), and `\xdigits` (hexadecimal).

If an unquoted argument begins with a colon (:), it is taken as a `psql` variable and the value of the variable is used as the argument instead.

Arguments that are enclosed in backquotes (`) are taken as a command line that is passed to the shell. The output of the command (with any trailing newline removed) is taken as the argument value. The above escape sequences also apply in backquotes.

Some commands take an SQL identifier (such as a table name) as argument. These arguments follow the syntax rules of SQL: Unquoted letters are forced to lowercase, while double quotes (") protect letters from case conversion and allow incorporation of whitespace into the identifier. Within double quotes, paired double quotes reduce to a single double quote in the resulting name. For example, `FOO"BAR"BAZ` is interpreted as `fooBARbaz`, and `"A weird" name"` becomes `A weird" name`.

Parsing for arguments stops when another unquoted backslash occurs. This is taken as the beginning of a new meta-command. The special sequence `\\` (two backslashes) marks the end of arguments and continues parsing SQL commands, if any. That way SQL and `psql` commands can be freely mixed on a line. But in any case, the arguments of a meta-command cannot continue beyond the end of the line.

The following meta-commands are defined:

\a

If the current table output format is unaligned, it is switched to aligned. If it is not unaligned, it is set to unaligned. This command is kept for backwards compatibility. See `\pset` for a more general solution.

\cd [directory]

Changes the current working directory. Without argument, changes to the current user's home directory. To print your current working directory, use `\!pwd`.

\C [title]

Sets the title of any tables being printed as the result of a query or unset any such title. This command is equivalent to `\pset title`.

\c | \connect [dbname [username] [host] [port]]

Establishes a new connection. If the new connection is successfully made, the previous connection is closed. If any of `dbname`, `username`, `host` or `port` are omitted, the value of that parameter from the previous connection is used. If the connection attempt failed, the previous connection will only be kept if `psql` is in interactive mode. When executing a non-interactive script, processing will immediately stop with an error. This distinction was chosen as a user convenience against typos, and a safety mechanism that scripts are not accidentally acting on the wrong database.

```
\copy {table [(column_list)] | (query)}
{from | to} {filename | stdin | stdout | pstdin | pstdout}
[with] [binary] [oids] [delimiter [as] 'character']
[null [as] 'string'] [csv [header]]
[quote [as] 'character'] [escape [as] 'character']
[force quote column_list] [force not null column_list]]
```

Performs a frontend (client) copy. This is an operation that runs an SQL `COPY` command, but instead of the server reading or writing the specified file, `psql` reads or writes the file and routes the data between the server and the local file system. This means that file accessibility and privileges are those of the local user, not the server, and no SQL superuser privileges are required.

The syntax of the command is similar to that of the SQL `COPY` command. Note that, because of this, special parsing rules apply to the `\copy` command. In particular, the variable substitution rules and backslash escapes do not apply.

`\copy ... from stdin | to stdout` reads/writes based on the command input and output respectively. All rows are read from the same source that issued the command, continuing until `\.` is read or the stream reaches EOF. Output is sent to the same place as command output. To read/write from `psql`'s standard input or output, use `pstdin` or `pstdout`. This option is useful for populating tables in-line within a SQL script file.

This operation is not as efficient as the SQL `COPY` command because all data must pass through the client/server connection.

\copyright

Shows the copyright and distribution terms of PostgreSQL on which Greenplum Database is based.

\d [relation_pattern] | \d+ [relation_pattern]

For each relation (table, external table, view, index, or sequence) matching the relation pattern, show all columns, their types, the tablespace (if not the default) and any special attributes such as `NOT NULL` or defaults, if any. Associated indexes, constraints, rules, and triggers are also shown, as is the view definition if the relation is a view.

The command form `\d+` is identical, except that more information is displayed: any comments associated with the columns of the table are shown, as is the presence of OIDs in the table.

If `\d` is used without a pattern argument, it is equivalent to `\dtvs` which will show a list of all tables, views, and sequences.

\da [aggregate_pattern]

Lists all available aggregate functions, together with the data types they operate on. If a pattern is specified, only aggregates whose names match the pattern are shown.

\db [tablespace_pattern] | \db+ [tablespace_pattern]

Lists all available tablespaces and the array of primary and mirror segment instance tablespace locations. If pattern is specified, only tablespaces whose names match the pattern are shown. If `+` is appended to the command name, each object is listed with its associated permissions.

\dc [conversion_pattern]

Lists all available conversions between character-set encodings. If pattern is specified, only conversions whose names match the pattern are listed.

\dC

Lists all available type casts.

\dd [object_pattern]

Lists all available objects. If pattern is specified, only matching objects are shown.

\dD [domain_pattern]

Lists all available domains. If pattern is specified, only matching domains are shown.

\df [function_pattern] | \df+ [function_pattern]

Lists available functions, together with their argument and return types. If pattern is specified, only functions whose names match the pattern are shown. If the form `\df+` is used, additional information about each function, including language and description, is shown. To reduce clutter, `\df` does not show data type I/O functions. This is implemented by ignoring functions that accept or return type `cstring`.

\dg [role_pattern]

Lists all database roles. If pattern is specified, only those roles whose names match the pattern are listed.

\distPvxS [*index* | *sequence* | *table* | *parent table* | *view* | *external_table* | *system_object*]

This is not the actual command name: the letters *i*, *s*, *t*, *P*, *v*, *x*, *S* stand for index, sequence, table, parent table, view, external table, and system table, respectively. You can specify any or all of these letters, in any order, to obtain a listing of all the matching objects. The letter *s* restricts the listing to system objects; without *s*, only non-system objects are shown. If *+* is appended to the command name, each object is listed with its associated description, if any. If a pattern is specified, only objects whose names match the pattern are listed.

\dl

This is an alias for `\lo_list`, which shows a list of large objects.

\dn [*schema_pattern*] | **\dn+** [*schema_pattern*]

Lists all available schemas (namespaces). If *pattern* is specified, only schemas whose names match the pattern are listed. Non-local temporary schemas are suppressed. If *+* is appended to the command name, each object is listed with its associated permissions and description, if any.

\do [*operator_pattern*]

Lists available operators with their operand and return types. If *pattern* is specified, only operators whose names match the pattern are listed.

\dp [*relation_pattern_to_show_privileges*]

Produces a list of all available tables, views and sequences with their associated access privileges. If *pattern* is specified, only tables, views and sequences whose names match the pattern are listed. The `GRANT` and `REVOKE` commands are used to set access privileges.

\dT [*datatype_pattern*] | **\dT+** [*datatype_pattern*]

Lists all data types or only those that match *pattern*. The command form `\dT+` shows extra information.

\du [*role_pattern*]

Lists all database roles, or only those that match *pattern*.

\e | **\edit** [*filename*]

If a file name is specified, the file is edited; after the editor exits, its content is copied back to the query buffer. If no argument is given, the current query buffer is copied to a temporary file which is then edited in the same fashion. The new query buffer is then re-parsed according to the normal rules of `psql`, where the whole buffer is treated as a single line. (Thus you cannot make scripts this way. Use `\i` for that.) This means also that if the query ends with (or rather contains) a semicolon, it is immediately executed. In other cases it will merely wait in the query buffer.

`psql` searches the environment variables `PSQL_EDITOR`, `EDITOR`, and `VISUAL` (in that order) for an editor to use. If all of them are unset, `vi` is used on Unix systems, `notepad.exe` on Windows systems.

`\echo text [...]`

Prints the arguments to the standard output, separated by one space and followed by a newline. This can be useful to intersperse information in the output of scripts.

If you use the `\o` command to redirect your query output you may wish to use `\qecho` instead of this command.

`\encoding [encoding]`

Sets the client character set encoding. Without an argument, this command shows the current encoding.

`\f [field_separator_string]`

Sets the field separator for unaligned query output. The default is the vertical bar (`|`). See also `\pset` for a generic way of setting output options.

`\g [{filename | command}]`

Sends the current query input buffer to the server and optionally stores the query's output in a file or pipes the output into a separate Unix shell executing command. A bare `\g` is virtually equivalent to a semicolon. A `\g` with argument is a one-shot alternative to the `\o` command.

`\h | \help [sql_command]`

Gives syntax help on the specified SQL command. If a command is not specified, then `psql` will list all the commands for which syntax help is available. Use an asterisk (`*`) to show syntax help on all SQL commands. To simplify typing, commands that consists of several words do not have to be quoted.

`\H`

Turns on HTML query output format. If the HTML format is already on, it is switched back to the default aligned text format. This command is for compatibility and convenience, but see `\pset` about setting other output options.

`\i input_filename`

Reads input from a file and executes it as though it had been typed on the keyboard. If you want to see the lines on the screen as they are read you must set the variable `ECHO` to `all`.

`\l | \list | \l+ | \list+`

List the names, owners, and character set encodings of all the databases in the server. If `+` is appended to the command name, database descriptions are also displayed.

`\lo_export oid filename`

Reads the large object with OID `oid` from the database and writes it to `filename`. Note that this is subtly different from the server function `lo_export`, which acts with the permissions of the user that the database server runs as and on the server's file system. Use `\lo_list` to find out the large object's OID.

`\lo_import large_object_filename [comment]`

Stores the file into a large object. Optionally, it associates the given comment with the object. Example:

```
mydb=> \lo_import '/home/gpadmin/pictures/photo.xcf' 'a
picture of me'
lo_import 152801
```

The response indicates that the large object received object ID 152801 which one ought to remember if one wants to access the object ever again. For that reason it is recommended to always associate a human-readable comment with every object. Those can then be seen with the `\lo_list` command. Note that this command is subtly different from the server-side `lo_import` because it acts as the local user on the local file system, rather than the server's user and file system.

`\lo_list`

Shows a list of all large objects currently stored in the database, along with any comments provided for them.

`\lo_unlink largeobject_oid`

Deletes the large object of the specified OID from the database. Use `\lo_list` to find out the large object's OID.

`\o [{query_result_filename | |command}]`

Saves future query results to a file or pipes them into a Unix shell command. If no arguments are specified, the query output will be reset to the standard output. Query results include all tables, command responses, and notices obtained from the database server, as well as output of various backslash commands that query the database (such as `\d`), but not error messages. To intersperse text output in between query results, use `\qecho`.

`\p`

Print the current query buffer to the standard output.

`\password [username]`

Changes the password of the specified user (by default, the current user). This command prompts for the new password, encrypts it, and sends it to the server as an `ALTER ROLE` command. This makes sure that the new password does not appear in cleartext in the command history, the server log, or elsewhere.

`\pset print_option [value]`

This command sets options affecting the output of query result tables.

`print_option` describes which option is to be set. Adjustable printing options are:

- **format** — Sets the output format to one of `unaligned`, `aligned`, `html`, `latex`, or `troff-ms`. First letter abbreviations are allowed. Unaligned writes all columns of a row on a line, separated by the currently active field separator. This is intended to create output that might be intended to be read in by other programs. Aligned mode is the standard, human-readable, nicely formatted text output that is default. The HTML and LaTeX modes put out

tables that are intended to be included in documents using the respective mark-up language. They are not complete documents! (This might not be so dramatic in HTML, but in LaTeX you must have a complete document wrapper.)

- **border** — The second argument must be a number. In general, the higher the number the more borders and lines the tables will have, but this depends on the particular format. In HTML mode, this will translate directly into the `border=...` attribute, in the others only values 0 (no border), 1 (internal dividing lines), and 2 (table frame) make sense.
- **expanded | x** — Toggles between regular and expanded format. When expanded format is enabled, query results are displayed in two columns, with the column name on the left and the data on the right. This mode is useful if the data would not fit on the screen in the normal horizontal mode. Expanded mode is supported by all four output formats.
- **null** — The second argument is a string that should be printed whenever a column is null. The default is not to print anything, which can easily be mistaken for, say, an empty string. Thus, one might choose to write `\pset null '(null)'`.
- **fieldsep** — Specifies the field separator to be used in unaligned output mode. That way one can create, for example, tab- or comma-separated output, which other programs might prefer. To set a tab as field separator, type `\pset fieldsep '\t'`. The default field separator is '|' (a vertical bar).
- **footer** — Toggles the display of the default footer (`x` rows).
- **numericlocale** — Toggles the display of a locale-aware character to separate groups of digits to the left of the decimal marker. It also enables a locale-aware decimal marker.
- **recordsep** — Specifies the record (line) separator to use in unaligned output mode. The default is a newline character.
- **tuples_only | t** — Toggles between tuples only and full display. Full display may show extra information such as column headers, titles, and various footers. In tuples only mode, only actual table data is shown.
- **title [text]** — Sets the table title for any subsequently printed tables. This can be used to give your output descriptive tags. If no argument is given, the title is unset.
- **tableattr | T [text]** — Allows you to specify any attributes to be placed inside the HTML table tag. This could for example be `cellpadding` or `bgcolor`. Note that you probably don't want to specify `border` here, as that is already taken care of by `\pset border`.
- **pager** — Controls the use of a pager for query and `psql` help output. When `on`, if the environment variable `PAGER` is set, the output is piped to the specified program. Otherwise a platform-dependent default (such as `more`) is used. When `off`, the pager is not used. When `on`, the pager is used only when appropriate. Pager can also be set to `always`, which causes the pager to be always used.

\q

Quits the `psql` program.

\qecho *text* [...]

This command is identical to `\echo` except that the output will be written to the query output channel, as set by `\o`.

\r

Resets (clears) the query buffer.

\s [*history_filename*]

Print or save the command line history to *filename*. If *filename* is omitted, the history is written to the standard output.

\set [*name* [*value* [...]]]

Sets the internal variable *name* to *value* or, if more than one value is given, to the concatenation of all of them. If no second argument is given, the variable is just set with no value. To unset a variable, use the `\unset` command.

Valid variable names can contain characters, digits, and underscores. See “Variables” on page 743. Variable names are case-sensitive.

Although you are welcome to set any variable to anything you want, `psql` treats several variables as special. They are documented in the section about variables.

This command is totally separate from the SQL command `SET`.

\t

Toggles the display of output column name headings and row count footer.

\T *table_options*

Allows you to specify attributes to be placed within the table tag in HTML tabular output mode.

\timing

Toggles a display of how long each SQL statement takes, in milliseconds.

\w {*filename* | |*command*}

Outputs the current query buffer to a file or pipes it to a Unix command.

\x

Toggles expanded table formatting mode.

\z [*relation_to_show_privileges*]

Produces a list of all available tables, views and sequences with their associated access privileges. If a pattern is specified, only tables, views and sequences whose names match the pattern are listed. This is an alias for `\dp`.

\! [*command*]

Escapes to a separate Unix shell or executes the Unix command. The arguments are not further interpreted, the shell will see them as is.

\?Shows help information about the `psql` backslash commands.

Patterns

The various `\d` commands accept a pattern parameter to specify the object name(s) to be displayed. In the simplest case, a pattern is just the exact name of the object. The characters within a pattern are normally folded to lower case, just as in SQL names; for example, `\dt FOO` will display the table named `foo`. As in SQL names, placing double quotes around a pattern stops folding to lower case. Should you need to include an actual double quote character in a pattern, write it as a pair of double quotes within a double-quote sequence; again this is in accord with the rules for SQL quoted identifiers. For example, `\dt "FOO" "BAR"` will display the table named `FOO"BAR` (not `foo"bar`). Unlike the normal rules for SQL names, you can put double quotes around just part of a pattern, for instance `\dt FOO"FOO"BAR` will display the table named `fooFOObar`.

Within a pattern, `*` matches any sequence of characters (including no characters) and `?` matches any single character. (This notation is comparable to Unix shell file name patterns.) For example, `\dt int*` displays all tables whose names begin with `int`. But within double quotes, `*` and `?` lose these special meanings and are just matched literally.

A pattern that contains a dot (`.`) is interpreted as a schema name pattern followed by an object name pattern. For example, `\dt foo*.bar*` displays all tables whose table name starts with `bar` that are in schemas whose schema name starts with `foo`. When no dot appears, then the pattern matches only objects that are visible in the current schema search path. Again, a dot within double quotes loses its special meaning and is matched literally.

Advanced users can use regular-expression notations. All regular expression special characters work as specified in the [PostgreSQL documentation on regular expressions](#), except for `.` which is taken as a separator as mentioned above, `*` which is translated to the regular-expression notation `.`, and `?` which is translated to `.`. You can emulate these pattern characters at need by writing `?` for `.`, `(R+|)` for `R*`, or `(R|)` for `R?`. Remember that the pattern must match the whole name, unlike the usual interpretation of regular expressions; write `*` at the beginning and/or end if you don't wish the pattern to be anchored. Note that within double quotes, all regular expression special characters lose their special meanings and are matched literally. Also, the regular expression special characters are matched literally in operator name patterns (such as the argument of `\do`).

Whenever the pattern parameter is omitted completely, the `\d` commands display all objects that are visible in the current schema search path — this is equivalent to using the pattern `*`. To see all objects in the database, use the pattern `*.*`.

Advanced Features

Variables

`psql` provides variable substitution features similar to common Unix command shells. Variables are simply name/value pairs, where the value can be any string of any length. To set variables, use the `psql` meta-command `\set`:

```
testdb=> \set foo bar
```

sets the variable `foo` to the value `bar`. To retrieve the content of the variable, precede the name with a colon and use it as the argument of any slash command:

```
testdb=> \echo :foo
bar
```

Note: The arguments of `\set` are subject to the same substitution rules as with other commands. Thus you can construct interesting references such as `\set :foo 'something'` and get ‘soft links’ or ‘variable variables’ of Perl or PHP fame, respectively. Unfortunately, there is no way to do anything useful with these constructs. On the other hand, `\set bar :foo` is a perfectly valid way to copy a variable.

If you call `\set` without a second argument, the variable is set, with an empty string as *value*. To unset (or delete) a variable, use the command `\unset`.

`psql`'s internal variable names can consist of letters, numbers, and underscores in any order and any number of them. A number of these variables are treated specially by `psql`. They indicate certain option settings that can be changed at run time by altering the value of the variable or represent some state of the application. Although you can use these variables for any other purpose, this is not recommended, as the program behavior might behave unexpectedly. By convention, all specially treated variables consist of all upper-case letters (and possibly numbers and underscores). To ensure maximum compatibility in the future, avoid using such variable names for your own purposes. A list of all specially treated variables are as follows:

AUTOCOMMIT

When on (the default), each SQL command is automatically committed upon successful completion. To postpone commit in this mode, you must enter a `BEGIN` or `START TRANSACTION` SQL command. When off or unset, SQL commands are not committed until you explicitly issue `COMMIT` or `END`. The autocommit-on mode works by issuing an implicit `BEGIN` for you, just before any command that is not already in a transaction block and is not itself a `BEGIN` or other transaction-control command, nor a command that cannot be executed inside a transaction block (such as `VACUUM`).

In autocommit-off mode, you must explicitly abandon any failed transaction by entering `ABORT` or `ROLLBACK`. Also keep in mind that if you exit the session without committing, your work will be lost.

The autocommit-on mode is PostgreSQL's traditional behavior, but autocommit-off is closer to the SQL spec. If you prefer autocommit-off, you may wish to set it in your `~/.psqlrc` file.

DBNAME

The name of the database you are currently connected to. This is set every time you connect to a database (including program start-up), but can be unset.

ECHO

If set to all, all lines entered from the keyboard or from a script are written to the standard output before they are parsed or executed. To select this behavior on program start-up, use the switch `-a`. If set to queries, `psql` merely prints all queries as they are sent to the server. The switch for this is `-e`.

ECHO_HIDDEN

When this variable is set and a backslash command queries the database, the query is first shown. This way you can study the Greenplum Database internals and provide similar functionality in your own programs. (To select this behavior on program start-up, use the switch `-E`.) If you set the variable to the value `noexec`, the queries are just shown but are not actually sent to the server and executed.

ENCODING

The current client character set encoding.

FETCH_COUNT

If this variable is set to an integer value > 0 , the results of `SELECT` queries are fetched and displayed in groups of that many rows, rather than the default behavior of collecting the entire result set before display. Therefore only a limited amount of memory is used, regardless of the size of the result set. Settings of 100 to 1000 are commonly used when enabling this feature. Keep in mind that when using this feature, a query may fail after having already displayed some rows.

Although you can use any output format with this feature, the default aligned format tends to look bad because each group of `FETCH_COUNT` rows will be formatted separately, leading to varying column widths across the row groups. The other output formats work better.

HISTCONTROL

If this variable is set to `ignoreSpace`, lines which begin with a space are not entered into the history list. If set to a value of `ignoreDups`, lines matching the previous history line are not entered. A value of `ignoreBoth` combines the two options. If unset, or if set to any other value than those above, all lines read in interactive mode are saved on the history list.

HISTFILE

The file name that will be used to store the history list. The default value is `~/.psql_history`. For example, putting

```
\set HISTFILE ~/.psql_history- :DBNAME
```

in `~/.psqlrc` will cause `psql` to maintain a separate history for each database.

HISTSIZE

The number of commands to store in the command history. The default value is 500.

HOST

The database server host you are currently connected to. This is set every time you connect to a database (including program start-up), but can be unset.

IGNOREEOF

If unset, sending an EOF character (usually CTRL+D) to an interactive session of `psql` will terminate the application. If set to a numeric value, that many EOF characters are ignored before the application terminates. If the variable is set but has no numeric value, the default is 10.

LASTOID

The value of the last affected OID, as returned from an `INSERT` or `lo_insert` command. This variable is only guaranteed to be valid until after the result of the next SQL command has been displayed.

ON_ERROR_ROLLBACK

When on, if a statement in a transaction block generates an error, the error is ignored and the transaction continues. When interactive, such errors are only ignored in interactive sessions, and not when reading script files. When off (the default), a statement in a transaction block that generates an error aborts the entire transaction. The `on_error_rollback-on` mode works by issuing an implicit `SAVEPOINT` for you, just before each command that is in a transaction block, and rolls back to the savepoint on error.

ON_ERROR_STOP

By default, if non-interactive scripts encounter an error, such as a malformed SQL command or internal meta-command, processing continues. This has been the traditional behavior of `psql` but it is sometimes not desirable. If this variable is set, script processing will immediately terminate. If the script was called from another script it will terminate in the same fashion. If the outermost script was not called from an interactive `psql` session but rather using the `-f` option, `psql` will return error code 3, to distinguish this case from fatal error conditions (error code 1).

PORT

The database server port to which you are currently connected. This is set every time you connect to a database (including program start-up), but can be unset.

PROMPT1**PROMPT2****PROMPT3**

These specify what the prompts `psql` issues should look like. See “[Prompting](#)” on page 747.

QUIET

This variable is equivalent to the command line option `-q`. It is not very useful in interactive mode.

SINGLELINE

This variable is equivalent to the command line option `-s`.

SINGLESTEP

This variable is equivalent to the command line option `-s`.

USER

The database user you are currently connected as. This is set every time you connect to a database (including program start-up), but can be unset.

VERBOSITY

This variable can be set to the values `default`, `verbose`, or `terse` to control the verbosity of error reports.

SQL Interpolation

An additional useful feature of `psql` variables is that you can substitute (interpolate) them into regular SQL statements. The syntax for this is again to prepend the variable name with a colon (`:`).

```
testdb=> \set foo 'my_table'
testdb=> SELECT * FROM :foo;
```

would then query the table `my_table`. The value of the variable is copied literally, so it can even contain unbalanced quotes or backslash commands. You must make sure that it makes sense where you put it. Variable interpolation will not be performed into quoted SQL entities.

A popular application of this facility is to refer to the last inserted OID in subsequent statements to build a foreign key scenario. Another possible use of this mechanism is to copy the contents of a file into a table column. First load the file into a variable and then proceed as above.

```
testdb=> \set content '' `cat my_file.txt` ''
testdb=> INSERT INTO my_table VALUES (:content);
```

One problem with this approach is that `my_file.txt` might contain single quotes. These need to be escaped so that they don't cause a syntax error when the second line is processed. This could be done with the program `sed`:

```
testdb=> \set content '' `sed -e "s/'/'/'g" < my_file.txt`
''
```

If you are using non-standard-conforming strings then you'll also need to double backslashes. This is a bit tricky:

```
testdb=> \set content '' `sed -e "s/'/'/'g" -e
's/\\/\//g' < my_file.txt` ''
```

Note the use of different shell quoting conventions so that neither the single quote marks nor the backslashes are special to the shell. Backslashes are still special to `sed`, however, so we need to double them.

Since colons may legally appear in SQL commands, the following rule applies: the character sequence `":name"` is not changed unless `"name"` is the name of a variable that is currently set. In any case you can escape a colon with a backslash to protect it from substitution. (The colon syntax for variables is standard SQL for embedded query languages, such as ECPG. The colon syntax for array slices and type casts are Greenplum Database extensions, hence the conflict.)

Prompting

The prompts `psql` issues can be customized to your preference. The three variables `PROMPT1`, `PROMPT2`, and `PROMPT3` contain strings and special escape sequences that describe the appearance of the prompt. Prompt 1 is the normal prompt that is issued when `psql` requests a new command. Prompt 2 is issued when more input is expected during command input because the command was not terminated with a semicolon or a quote was not closed. Prompt 3 is issued when you run an SQL `COPY` command and you are expected to type in the row values on the terminal.

The value of the selected prompt variable is printed literally, except where a percent sign (`%`) is encountered. Depending on the next character, certain other text is substituted instead. Defined substitutions are:

`%M`

The full host name (with domain name) of the database server, or `[local]` if the connection is over a Unix domain socket, or `[local:/dir/name]`, if the Unix domain socket is not at the compiled in default location.

`%m`

The host name of the database server, truncated at the first dot, or `[local]` if the connection is over a Unix domain socket.

`%>`

The port number at which the database server is listening.

`%n`

The database session user name. (The expansion of this value might change during a database session as the result of the command `SET SESSION AUTHORIZATION`.)

`%/`

The name of the current database.

`%~`

Like `%/`, but the output is `~` (tilde) if the database is your default database.

`%#`

If the session user is a database superuser, then a `#`, otherwise a `>`. (The expansion of this value might change during a database session as the result of the command `SET SESSION AUTHORIZATION`.)

%R

In prompt 1 normally =, but ^ if in single-line mode, and ! if the session is disconnected from the database (which can happen if `\connect` fails). In prompt 2 the sequence is replaced by -, *, a single quote, a double quote, or a dollar sign, depending on whether `psql` expects more input because the command wasn't terminated yet, because you are inside a `/* ... */` comment, or because you are inside a quoted or dollar-escaped string. In prompt 3 the sequence doesn't produce anything.

%x

Transaction status: an empty string when not in a transaction block, or * when in a transaction block, or ! when in a failed transaction block, or ? when the transaction state is indeterminate (for example, because there is no connection).

%digits

The character with the indicated octal code is substituted.

%:name:

The value of the `psql` variable name. See “Variables” on page 743 for details.

%`command`

The output of command, similar to ordinary back-tick substitution.

%[... %]

Prompts may contain terminal control characters which, for example, change the color, background, or style of the prompt text, or change the title of the terminal window. In order for line editing to work properly, these non-printing control characters must be designated as invisible by surrounding them with `%[` and `%)`. Multiple pairs of these may occur within the prompt. For example,

```
testdb=> \set PROMPT1 '%[%033[1;33;40m%]n@%/%R%[%033[0m%]##
'
```

results in a boldfaced (1;) yellow-on-black (33;40) prompt on VT100-compatible, color-capable terminals. To insert a percent sign into your prompt, write `%%`. The default prompts are `'%/%R%# '` for prompts 1 and 2, and `'>> '` for prompt 3.

Command-Line Editing

`psql` supports the NetBSD *libedit* library for convenient line editing and retrieval. The command history is automatically saved when `psql` exits and is reloaded when `psql` starts up. Tab-completion is also supported, although the completion logic makes no claim to be an SQL parser. If for some reason you do not like the tab completion, you can turn it off by putting this in a file named `.inputrc` in your home directory:

```
$if psql
set disable-completion on
$endif
```

Environment

PAGER

If the query results do not fit on the screen, they are piped through this command. Typical values are `more` or `less`. The default is platform-dependent. The use of the pager can be disabled by using the `\pset` command.

PGDATABASE

PGHOST

PGPORT

PGUSER

Default connection parameters.

PSQL_EDITOR

EDITOR

VISUAL

Editor used by the `\e` command. The variables are examined in the order listed; the first that is set is used.

SHELL

Command executed by the `\!` command.

TMPDIR

Directory for storing temporary files. The default is `/tmp`.

Files

Before starting up, `psql` attempts to read and execute commands from the user's `~/.psqlrc` file.

The command-line history is stored in the file `~/.psql_history`.

Notes

`psql` only works smoothly with servers of the same version. That does not mean other combinations will fail outright, but subtle and not-so-subtle problems might come up. Backslash commands are particularly likely to fail if the server is of a different version.

Notes for Windows users

`psql` is built as a console application. Since the Windows console windows use a different encoding than the rest of the system, you must take special care when using 8-bit characters within `psql`. If `psql` detects a problematic console code page, it will warn you at startup. To change the console code page, two things are necessary:

Set the code page by entering `cmd.exe /c chcp 1252`. (1252 is a character encoding of the Latin alphabet, used by Microsoft Windows for English and some other Western languages.) If you are using Cygwin, you can put this command in `/etc/profile`.

Set the console font to Lucida Console, because the raster font does not work with the ANSI code page.

Examples

Start `psql` in interactive mode:

```
psql -p 54321 -U sally mydatabase
```

In `psql` interactive mode, spread a command over several lines of input. Notice the changing prompt:

```
testdb=> CREATE TABLE my_table (
testdb(> first integer not null default 0,
testdb(> second text)
testdb-> ;
CREATE TABLE
```

Look at the table definition:

```
testdb=> \d my_table
                Table "my_table"
Attribute | Type   | Modifier
-----+-----+-----
first     | integer | not null default 0
second    | text    |
```

Run `psql` in non-interactive mode by passing in a file containing SQL commands:

```
psql -f /home/gpadmin/test/myscript.sql
```

reindexdb

Rebuilds indexes in a database.

Synopsis

```
reindexdb [connection-option...] [--table | -t table] [--index | -i index] [dbname]
```

```
reindexdb [connection-option...] [--all | -a]
```

```
reindexdb [connection-option...] [--system | -s] [dbname]
```

Description

`reindexdb` is a utility for rebuilding indexes in Greenplum Database, and is a wrapper around the SQL command [REINDEX](#).

Options

-a | **--all**

Reindex all databases.

-s | **--system**

Reindex system catalogs.

-t *table* | **--table** *table*

Reindex table only.

-i *index* | **--index** *index*

Recreate index only.

[-d] *dbname* | **[--dbname]** *dbname*

Specifies the name of the database to be reindexed. If this is not specified and `--all` is not used, the database name is read from the environment variable `PGDATABASE`. If that is not set, the user name specified for the connection is used.

-e | **--echo**

Echo the commands that `reindexdb` generates and sends to the server.

-q | **--quiet**

Do not display a response.

Connection Options

-h *host* | **--host** *host*

Specifies the host name of the machine on which the Greenplum master database server is running. If not specified, reads from the environment variable `PGHOST` or defaults to `localhost`.

-p *port* | --port *port*

Specifies the TCP port on which the Greenplum master database server is listening for connections. If not specified, reads from the environment variable `PGPORT` or defaults to 5432.

-U *username* | --username *username*

The database role name to connect as. If not specified, reads from the environment variable `PGUSER` or defaults to the current system user name.

-W | --password

Force a password prompt.

Notes

`reindexdb` might need to connect several times to the master server, asking for a password each time. It is convenient to have a `~/ .pgpass` file in such cases.

Examples

To reindex the database *mydb*:

```
reindexdb mydb
```

To reindex the table *foo* and the index *bar* in a database named *abcd*:

```
reindexdb --table foo --index bar abcd
```

See Also

REINDEX

vacuumdb

Garbage-collects and analyzes a database.

Synopsis

```
vacuumdb [connection-option...] [--full | -f] [--verbose | -v]
[--analyze | -z] [--table | -t table [(column [,...])] ]
[dbname]

vacuumdb [connection-options...] [--all | -a] [--full | -f]
[--verbose | -v] [--analyze | -z]
```

Description

`vacuumdb` is a utility for cleaning a PostgreSQL database. `vacuumdb` will also generate internal statistics used by the PostgreSQL query optimizer.

`vacuumdb` is a wrapper around the SQL command `VACUUM`. There is no effective difference between vacuuming databases via this utility and via other methods for accessing the server.

Options

-a | --all

Vacuums all databases.

[-d] dbname | [--dbname] dbname

The name of the database to vacuum. If this is not specified and `--all` is not used, the database name is read from the environment variable `PGDATABASE`. If that is not set, the user name specified for the connection is used.

-e | --echo

Echo the commands that `reindexdb` generates and sends to the server.

-f | --full

Selects a full vacuum, which may reclaim more space, but takes much longer and exclusively locks the table.

Warning: A `VACUUM FULL` is not recommended in Greenplum Database.

-q | --quiet

Do not display a response.

-t table [(column)] | --table table [(column)]

Clean or analyze this table only. Column names may be specified only in conjunction with the `--analyze` option. If you specify columns, you probably have to escape the parentheses from the shell.

-v | --verbose

Print detailed information during processing.

-z | --analyze

Collect statistics for use by the query planner.

Connection Options

-h *host* | --host *host*

Specifies the host name of the machine on which the Greenplum master database server is running. If not specified, reads from the environment variable `PGHOST` or defaults to `localhost`.

-p *port* | --port *port*

Specifies the TCP port on which the Greenplum master database server is listening for connections. If not specified, reads from the environment variable `PGPORT` or defaults to `5432`.

-U *username* | --username *username*

The database role name to connect as. If not specified, reads from the environment variable `PGUSER` or defaults to the current system user name.

-W | --password

Force a password prompt.

Notes

`vacuumdb` might need to connect several times to the master server, asking for a password each time. It is convenient to have a `~/ .pgpass` file in such cases.

Examples

To clean the database *test*:

```
vacuumdb test
```

To clean and analyze a database named *bigdb*:

```
vacuumdb --analyze bigdb
```

To clean a single table *foo* in a database named *mydb*, and analyze a single column *bar* of the table. Note the quotes around the table and column names to escape the parentheses from the shell:

```
vacuumdb --analyze --verbose --table 'foo(bar)' mydb
```

See Also

[VACUUM](#), [ANALYZE](#)

D. Server Configuration Parameters

There are many configuration parameters that affect the behavior of the Greenplum Database system. Most of these configuration parameters have the same names, settings, and behaviors as in a regular PostgreSQL database system.

Parameter Types and Values

All parameter names are case-insensitive. Every parameter takes a value of one of four types: `boolean`, `integer`, `floating point`, or `string`. Boolean values may be written as `ON`, `OFF`, `TRUE`, `FALSE`, `YES`, `NO`, `1`, `0` (all case-insensitive).

Some settings specify a memory size or time value. Each of these has an implicit unit, which is either kilobytes, blocks (typically eight kilobytes), milliseconds, seconds, or minutes. Valid memory size units are `kB` (kilobytes), `MB` (megabytes), and `GB` (gigabytes). Valid time units are `ms` (milliseconds), `s` (seconds), `min` (minutes), `h` (hours), and `d` (days). Note that the multiplier for memory units is 1024, not 1000.

Setting Parameters

Many of the configuration parameters have limitations on who can change them and where or when they can be set. For example, to change certain parameters, you must be a Greenplum Database superuser. Other parameters can only be set in the `postgresql.conf` file or require a restart of the system for the changes to take effect. A parameter that is classified as *runtime* can be set at the system level (in the `postgresql.conf` file), at the database-level (using `ALTER DATABASE`), at the role-level (using `ALTER ROLE`), or at the session-level (using `SET`).

In Greenplum Database, the master and each segment instance has its own `postgresql.conf` file (located in their respective data directories). Some parameters are considered *local* parameters, meaning that each segment instance looks to its own `postgresql.conf` file to get the value of that parameter. You must set local parameters on every instance in the system (master and segments). Others parameters are considered *global* or *master* parameters. Global and master parameters need only be set at the master instance.

Table D.1 Settable Classifications

Set Classification	Description
superuser	Can only be set by a database superuser. Regular database users cannot set this parameter.
runtime	Can be set at the session, role or database level. Runtime parameters are always <i>master</i> or <i>global</i> parameters in Greenplum Database.
server start	Requires a system restart for changes to take effect - cannot be set at the session, role, or database level.
postgresql.conf	Can only be set in the <code>postgresql.conf</code> file - cannot be set at the session, role, or database level.
read only	The current value of the parameter can be shown but not altered.
master	Only needs to be set on the Greenplum master - the segments ignore this parameter. Master parameters are only relevant to master processes such as query planning and client authentication.
global	Only needs to be set on the Greenplum master - the segments inherit their value from the master.
local	Must be set in the <code>postgresql.conf</code> file of the master AND each segment instance. Requires a system restart for changes to take effect.

Table D.2 Server Configuration Parameters

Parameter	Value Range	Default	Description	Settable by	GPDB
add_missing_from	boolean	off	Automatically adds missing table references to FROM clauses. Present for compatibility with releases of PostgreSQL prior to 8.1, where this behavior was allowed by default.	runtime	master
array_nulls	boolean	on	This controls whether the array input parser recognizes unquoted NULL as specifying a null array element. By default, this is on, allowing array values containing null values to be entered. Greenplum Database versions before 3.0 did not support null values in arrays, and therefore would treat NULL as specifying a normal array element with the string value 'NULL'.	runtime	master
authentication_timeout	number of seconds	1min	Maximum time to complete client authentication. This prevents hung clients from occupying a connection indefinitely.	superuser server start postgresql.conf	local
autovacuum	boolean	off	Autovacuum is currently disabled in Greenplum Database. Controls whether the server should start the autovacuum subprocess. <i>stats_start_collector</i> and <i>stats_row_level</i> must also be on for this to start.	superuser server start postgresql.conf	local
autovacuum_analyze_scale_factor	floating point	0.1	A fraction of the table size to add to <i>autovacuum_analyze_threshold</i> when deciding whether to trigger an ANALYZE.	superuser server start postgresql.conf	local
autovacuum_analyze_threshold	integer > 0	250	Specifies the minimum number of inserted, updated or deleted tuples needed to trigger an ANALYZE in any one table.	superuser server start postgresql.conf	local
autovacuum_freeze_max_age	integer > 0	20000000 0	Specifies the maximum age (in transactions) before a VACUUM operation is forced to prevent transaction ID wraparound within a table. Note that the system will launch autovacuum processes to prevent wraparound even when autovacuum is otherwise disabled.	superuser server start postgresql.conf	local

Table D.2 Server Configuration Parameters

Parameter	Value Range	Default	Description	Settable by	GPDB
autovacuum_naptime	number of seconds	1min	Specifies the delay between activity rounds for the autovacuum subprocess. In each round the subprocess examines one database and issues VACUUM and ANALYZE commands as needed for tables in that database.	superuser server start postgresql.conf	local
autovacuum_vacuum_cost_delay	integer	-1	Specifies the cost delay value that will be used in automatic VACUUM operations. If -1, the regular <i>vacuum_cost_delay</i> value will be used.	superuser server start postgresql.conf	local
autovacuum_vacuum_cost_limit	integer	-1	Specifies the cost limit value that will be used in automatic VACUUM operations. If -1, the regular <i>vacuum_cost_limit</i> value will be used.	superuser server start postgresql.conf	local
autovacuum_vacuum_scale_factor	floating point	0.2	Specifies a fraction of the table size to add to <i>autovacuum_vacuum_threshold</i> when deciding whether to trigger a VACUUM.	superuser server start postgresql.conf	local
autovacuum_vacuum_threshold	integer > 0	500	Specifies the minimum number of updated or deleted tuples needed to trigger a VACUUM in any one table.	superuser server start postgresql.conf	local
backslash_quote	on (allow \ always) off (reject always) safe_encoding (allow only if client encoding does not allow ASCII \ within a multibyte character)	safe_encoding	This controls whether a quote mark can be represented by \ in a string literal. The preferred, SQL-standard way to represent a quote mark is by doubling it (") but PostgreSQL has historically also accepted \. However, use of \ creates security risks because in some client character set encodings, there are multibyte characters in which the last byte is numerically equivalent to ASCII \.	runtime	master
bgwriter_all_maxpages	number of buffers	5	In each round, no more than this many buffers will be written as a result of the scan of the entire buffer pool. (If this limit is reached, the scan stops, and resumes at the next buffer during the next round.)	superuser server start postgresql.conf	local

Table D.2 Server Configuration Parameters

Parameter	Value Range	Default	Description	Settable by	GPDB
bgwriter_all_percent	floating point (percentage of the total number of shared buffers)	0.333	To reduce the amount of work that will be needed at checkpoint time, the background writer also does a circular scan through the entire buffer pool, writing buffers that are found to be dirty. In each round, it examines this percentage of the buffers. With the default <i>bgwriter_delay</i> setting, this will allow the entire shared buffer pool to be scanned about once per minute.	superuser server start postgresql.conf	local
bgwriter_delay	milliseconds > 0 (in multiples of 10)	200ms	Specifies the delay between activity rounds for the background writer. In each round the writer issues writes for some number of dirty buffers. It then sleeps for a number of milliseconds, and repeats.	superuser server start postgresql.conf	local
bgwriter_lru_maxpages	integer > 0	5	In each round, no more than this many buffers will be written as a result of scanning soon-to-be-recycled buffers.	superuser server start postgresql.conf	local
bgwriter_lru_percent	floating point (percentage of the total number of shared buffers)	1.0	To reduce the probability that server processes will need to issue their own writes, the background writer tries to write buffers that are likely to be recycled soon. In each round, it examines this percentage of the buffers that are nearest to being recycled, and writes any that are dirty.	superuser server start postgresql.conf	local
block_size	number of bytes	32768	Reports the size of a disk block.	read only	-
bonjour_name	string	unset	Specifies the Bonjour broadcast name. By default, the computer name is used, specified as an empty string. This option is ignored if the server was not compiled with Bonjour support.	superuser server start postgresql.conf	master
check_function_bodies	boolean	on	When set to off, disables validation of the function body string during CREATE FUNCTION. Disabling validation is occasionally useful to avoid problems such as forward references when restoring function definitions from a dump.	runtime	master
checkpoint_segments	number of WAL segments	8	Maximum distance between automatic WAL checkpoints, in log file segments (each segment is normally 16 megabytes).	superuser server start postgresql.conf	local
checkpoint_timeout	number of seconds	5min	Maximum time between automatic WAL checkpoints in seconds.	superuser server start postgresql.conf	local

Table D.2 Server Configuration Parameters

Parameter	Value Range	Default	Description	Settable by	GPDB
checkpoint_warning	number of seconds	30s	Write a message to the server log if checkpoints caused by the filling of checkpoint segment files happen closer together than this many seconds (which suggests that <i>checkpoint_segments</i> ought to be raised). 0 disables the warning.	superuser server start postgresql.conf	local
client_encoding	character set	UTF8	Sets the client-side encoding (character set). The default is to use the same as the database encoding. See Supported Character Sets in the PostgreSQL documentation.	runtime	global
client_min_messages	DEBUG5 DEBUG4 DEBUG3 DEBUG2 DEBUG1 LOG NOTICE WARNING ERROR FATAL PANIC	NOTICE	Controls which message levels are sent to the client. Each level includes all the levels that follow it. The later the level, the fewer messages are sent.	runtime	global
commit_delay	number of microseconds	0	Time delay between writing a commit record to the WAL buffer and flushing the buffer out to disk. A nonzero delay can allow multiple transactions to be committed with only one <i>fsync()</i> system call, if system load is high enough that additional transactions become ready to commit within the given interval.	runtime	global
commit_siblings	number of transactions	5	Minimum number of concurrent open transactions to require before performing the commit delay. A larger value makes it more probable that at least one other transaction will become ready to commit during the delay interval.	runtime	global
config_file	string	<data_directory>/postgresql.conf	Specifies the main server configuration file (postgresql.conf).	superuser server start postgresql.conf	local

Table D.2 Server Configuration Parameters

Parameter	Value Range	Default	Description	Settable by	GPDB
constraint_exclusion	boolean	on	Enables or disables the query planner's use of table constraints to optimize queries. When this parameter is on, the planner compares query conditions with table CHECK constraints, and omits scanning tables for which the conditions contradict the constraints. This can improve performance when inheritance is used to build partitioned tables. Turn this off if you are not using partitioned tables.	runtime	master
cpu_index_tuple_cost	floating point	0.005	Sets the planner's estimate of the cost of processing each index row during an index scan. This is measured as a fraction of the cost of a sequential page fetch.	runtime	master
cpu_operator_cost	floating point	0.0025	Sets the planner's estimate of the cost of processing each operator in a WHERE clause. This is measured as a fraction of the cost of a sequential page fetch.	runtime	master
cpu_tuple_cost	floating point	0.01	Sets the planner's estimate of the cost of processing each row during a query. This is measured as a fraction of the cost of a sequential page fetch.	runtime	master
cursor_tuple_fraction	integer	1	Tells the query planner how many rows are expected to be fetched in a cursor query, thereby allowing the planner to use this information to optimize the query plan. The default of 1 means all rows will be fetched.	runtime	master
custom_variable_classes	comma-separated list of class names	unset	Specifies one or several class names to be used for custom variables. A custom variable is a variable not normally known to the server but used by some add-on module. Such variables must have names consisting of a class name, a dot, and a variable name.	superuser server start postgresql.conf	local
data_directory	string		Specifies the directory to use for data storage. This option can only be set at server start.	superuser server start postgresql.conf	local
DateStyle	<format>, <date style> where <format> is ISO, Postgres, SQL, or German and <date style> is DMY, MDY, or YMD.	ISO, MDY	Sets the display format for date and time values, as well as the rules for interpreting ambiguous date input values. This variable contains two independent components: the output format specification and the input/output specification for year/month/day ordering.	runtime	global

Table D.2 Server Configuration Parameters

Parameter	Value Range	Default	Description	Settable by	GPDB
db_user_namespace	boolean	off	This enables per-database user names. If on, you should create users as <i>username@dbname</i> . To create ordinary global users, simply append @ when specifying the user name in the client.	superuser server start postgresql.conf	local
deadlock_timeout	number of milliseconds	1s	The number of milliseconds to wait on a lock before checking to see if there is a deadlock condition. On a heavily loaded server you might want to raise this value. Ideally the setting should exceed your typical transaction time, so as to improve the odds that a lock will be released before the waiter decides to check for deadlock.	superuser server start postgresql.conf	local
debug_assertions	boolean	off	Turns on various assertion checks.	superuser server start postgresql.conf	local
debug_pretty_print	boolean	off	Indents debug output to produce a more readable but much longer output format. <i>client_min_messages</i> or <i>log_min_messages</i> must be DEBUG1 or lower.	runtime	master
debug_print_parse	boolean	off	For each executed query, prints the resulting parse tree. <i>client_min_messages</i> or <i>log_min_messages</i> must be DEBUG1 or lower.	runtime	master
debug_print_plan	boolean	off	For each executed query, prints the Greenplum parallel query execution plan. <i>client_min_messages</i> or <i>log_min_messages</i> must be DEBUG1 or lower.	runtime	master
debug_print_prelim_plan	boolean	off	For each executed query, prints the preliminary query plan. <i>client_min_messages</i> or <i>log_min_messages</i> must be DEBUG1 or lower.	runtime	master
debug_print_rewritten	boolean	off	For each executed query, prints the query rewriter output. <i>client_min_messages</i> or <i>log_min_messages</i> must be DEBUG1 or lower.	runtime	master
debug_print_slice_table	boolean	off	For each executed query, prints the Greenplum query slice plan. <i>client_min_messages</i> or <i>log_min_messages</i> must be DEBUG1 or lower.	runtime	master

Table D.2 Server Configuration Parameters

Parameter	Value Range	Default	Description	Settable by	GPDB
default_statistics_target	integer > 0	25	Sets the default statistics target for table columns that have not had a column-specific target set via ALTER TABLE SET STATISTICS. Larger values increase the time needed to do ANALYZE, but may improve the quality of the planner's estimates.	runtime	master
default_tablespace	name of a tablespace	unset	The default tablespace in which to create objects (tables and indexes) when a CREATE command does not explicitly specify a tablespace.	superuser server start postgresql.conf	local
default_transaction_isolation	read committed read uncommitted repeatable read serializable	read committed	Controls the default isolation level of each new transaction.	runtime	global
default_transaction_read_only	boolean	off	Controls the default read-only status of each new transaction. A read-only SQL transaction cannot alter non-temporary tables.	runtime	master
default_with_oids	boolean	off	This controls whether CREATE TABLE includes an OID column in newly-created tables by default. The use of OIDs in user tables is considered deprecated, so this option is only present for backwards compatibility with PostgreSQL versions prior to 8.1.	runtime	master
dynamic_library_path	a list of absolute directory paths separated by colons	\$libdir	If a dynamically loadable module needs to be opened and the file name specified in the CREATE FUNCTION or LOAD command does not have a directory component (i.e. the name does not contain a slash), the system will search this path for the required file. The compiled-in PostgreSQL package library directory is substituted for \$libdir. This is where the modules provided by the standard PostgreSQL distribution are installed.	superuser server start postgresql.conf	local
effective_cache_size	floating point	512MB	Sets the planner's assumption about the effective size of the disk cache that is available to a single query. This is factored into estimates of the cost of using an index; a higher value makes it more likely index scans will be used, a lower value makes it more likely sequential scans will be used. This parameter has no effect on the size of shared memory allocated by a Greenplum server instance, nor does it reserve kernel disk cache; it is used only for estimation purposes.	runtime	master

Table D.2 Server Configuration Parameters

Parameter	Value Range	Default	Description	Settable by	GPDB
enable_bitmapscan	boolean	on	Enables or disables the query planner's use of bitmap-scan plan types. Note that this is different than a Bitmap Index Scan. A Bitmap Scan means that indexes will be dynamically converted to bitmaps in memory when appropriate, giving faster index performance on complex queries against very large tables. It is used when there are multiple predicates on different indexed columns. Each bitmap per column can be compared to create a final list of selected tuples.	runtime	master
enable_groupagg	boolean	on	Enables or disables the query planner's use of group aggregation plan types.	runtime	master
enable_hashagg	boolean	on	Enables or disables the query planner's use of hash aggregation plan types.	runtime	master
enable_hashjoin	boolean	on	Enables or disables the query planner's use of hash-join plan types.	runtime	master
enable_indexscan	boolean	on	Enables or disables the query planner's use of index-scan plan types.	runtime	master
enable_mergejoin	boolean	off	Enables or disables the query planner's use of merge-join plan types. Merge join is based on the idea of sorting the left- and right-hand tables into order and then scanning them in parallel. So, both data types must be capable of being fully ordered, and the join operator must be one that can only succeed for pairs of values that fall at the 'same place' in the sort order. In practice this means that the join operator must behave like equality.	runtime	master
enable_nestloop	boolean	off	Enables or disables the query planner's use of nested-loop join plans. It's not possible to suppress nested-loop joins entirely, but turning this variable off discourages the planner from using one if there are other methods available.	runtime	master
enable_seqscan	boolean	on	Enables or disables the query planner's use of sequential scan plan types. It's not possible to suppress sequential scans entirely, but turning this variable off discourages the planner from using one if there are other methods available.	runtime	master

Table D.2 Server Configuration Parameters

Parameter	Value Range	Default	Description	Settable by	GPDB
enable_sort	boolean	on	Enables or disables the query planner's use of explicit sort steps. It's not possible to suppress explicit sorts entirely, but turning this variable off discourages the planner from using one if there are other methods available.	runtime	master
enable_tidscan	boolean	on	Enables or disables the query planner's use of tuple identifier (TID) scan plan types.	runtime	master
escape_string_warning	boolean	on	When on, a warning is issued if a backslash (\) appears in an ordinary string literal ('...' syntax). Escape string syntax (E'...') should be used for escapes, because in future versions, ordinary strings will have the SQL standard-conforming behavior of treating backslashes literally.	runtime	master
explain_pretty_print	boolean	on	Determines whether EXPLAIN VERBOSE uses the indented or non-indented format for displaying detailed query-tree dumps.	runtime	master
external_pid_file	string	unset	Specifies the name of an additional process-id (PID) file that the postmaster should create for use by server administration programs.	superuser server start postgresql.conf	local
extra_float_digits	integer	0	Adjusts the number of digits displayed for floating-point values, including float4, float8, and geometric data types. The parameter value is added to the standard number of digits. The value can be set as high as 2, to include partially-significant digits; this is especially useful for dumping float data that needs to be restored exactly. Or it can be set negative to suppress unwanted digits.	runtime	master
from_collapse_limit	1- <i>n</i>	8	The planner will merge sub-queries into upper queries if the resulting FROM list would have no more than this many items. Smaller values reduce planning time but may yield inferior query plans.	runtime	master
fsynch	boolean	on	If enabled, the server will try to make sure that updates are physically written to disk, by issuing <i>fsync()</i> system calls. This ensures that the database can recover to a consistent state after an operating system or hardware crash.	superuser server start postgresql.conf	local

Table D.2 Server Configuration Parameters

Parameter	Value Range	Default	Description	Settable by	GPDB
full_page_writes	boolean	on	When on, the server writes the entire content of each disk page to WAL during the first modification of that page after a checkpoint. This parameter is currently ignored (treated as always on) because turning it off can cause failure to recover from crashes even when no hardware or OS-level error occurred.	not settable	local
gin_fuzzy_search_limit		0 (no limit)	The primary goal of GIN indexes is to support full-text search. There are often situations when a full-text search returns a very large set of results. Moreover, this often happens when the query contains very frequent words, so that the large result set is not even useful. To facilitate controlled execution of such queries GIN has a configurable soft upper limit. If a non-zero limit is set, then the returned set is a subset of the whole result set, chosen at random.	runtime	master
gp_adjust_selectivity_for_outerjoins	boolean	on	Enables the selectivity of NULL tests over outer joins.	runtime	master
gp_analyze_relative_error	floating point < 1.0	0.25	Sets the estimated acceptable error in the cardinality of the table — a value of 0.5 is supposed to be equivalent to an acceptable error of 50% (this is the default value used in PostgreSQL). If the statistics collected during ANALYZE are not producing good estimates of cardinality for a particular table attribute, decreasing the relative error fraction (accepting less error) tells the system to sample more rows.	runtime	master

Table D.2 Server Configuration Parameters

Parameter	Value Range	Default	Description	Settable by	GPDB
gp_autostats_mode	none on_change on_no_stats	on_no_stats	Specifies the mode for triggering automatic statistics collection with <code>ANALYZE</code> . The <code>on_no_stats</code> option triggers statistics collection for <code>CREATE TABLE AS SELECT</code> , <code>INSERT</code> , or <code>COPY</code> operations on any table that has no existing statistics. The <code>on_change</code> option triggers statistics collection only when the number of rows affected meets or exceeds the threshold defined by <code>gp_autostats_on_change_threshold</code> . Operations that can trigger automatic statistics collection with <code>on_change</code> are: <code>CREATE TABLE AS SELECT</code> <code>UPDATE</code> <code>DELETE</code> <code>INSERT</code> <code>COPY</code> Default is <code>on_no_stats</code> .	runtime	master
gp_autostats_on_change_threshold	integer	0	Specifies the threshold for automatic statistics collection when <code>gp_autostats_mode</code> is set to <code>on_change</code> . When a triggering table operation affects a number of rows exceeding this threshold, <code>ANALYZE</code> is added and statistics are collected for the table.	runtime	master
gp_cached_segworkers_threshold	integer > 0	5	When a user starts a session with Greenplum Database and issues a query, the system creates groups or 'gangs' of worker processes on each segment to do the work. After the work is done, the segment worker processes are destroyed except for a cached number which is set by this parameter. A lower setting conserves system resources on the segment hosts, but a higher setting may improve performance for power-users that want to issue many complex queries in a row.	runtime	master
gp_command_count	integer > 0		Shows how many commands the master has received from the client. Note that a single <code>SQLcommand</code> might actually involve more than one command internally, so the counter may increment by more than one for a single query. This counter also is shared by all of the segment processes working on the command.	read only	global

Table D.2 Server Configuration Parameters

Parameter	Value Range	Default	Description	Settable by	GPDB
gp_connections_per_thread	1-255	512	A tuning parameter for the Greenplum master that sets the number of client connections per thread that a query dispatcher process can create to a segment.	runtime	master
gp_debug_linger	number of seconds	0	Number of seconds for a Greenplum process to linger after a fatal internal error.	runtime	global
gp_enable_adaptive_nestloop	boolean	on	Enables the query planner to use a new type of join node called “Adaptive Nestloop” at query execution time. This causes the planner to favor a hash-join over a nested-loop join if the number of rows on the outer side of the join exceeds a precalculated threshold. This parameter improves performance of index operations, which previously favored slower nested-loop joins.	runtime	master
gp_enable_agg_distinct	boolean	on	Enables or disables two-phase aggregation to compute a single distinct-qualified aggregate. This applies only to subqueries that include a single distinct-qualified aggregate function.	runtime	master
gp_enable_agg_distinct_pruning	boolean	on	Enables or disables three-phase aggregation and join to compute distinct-qualified aggregates. This applies only to subqueries that include one or more distinct-qualified aggregate functions.	runtime	master
gp_enable_fallback_plan	boolean	on	Allows use of disabled plan types when a query would not be feasible without them.	runtime	master
gp_enable_fast_sri	boolean	on	When set to <code>on</code> , the query planner plans single row inserts so that they are sent directly to the correct segment instance (no motion operation required). This significantly improves performance of single-row-insert statements.	runtime	master
gp_enable_gpperfmon	boolean	off	Enables monitoring by turning on segment agent processes that collect performance data and send it through the interconnect to the master. Only set this parameter to <code>on</code> when the performance monitor is deployed on the master. Otherwise, unconsumed data files may accumulate on the master host.	superuser postgres.conf	global

Table D.2 Server Configuration Parameters

Parameter	Value Range	Default	Description	Settable by	GPDB
gp_enable_groupext_distinct_gather	boolean	on	Enables or disables gathering data to a single node to compute distinct-qualified aggregates on grouping extension queries. When this parameter and <code>gp_enable_groupext_distinct_pruning</code> are both enabled, the planner uses the cheaper plan.	runtime	master
gp_enable_groupext_distinct_pruning	boolean	on	Enables or disables three-phase aggregation and join to compute distinct-qualified aggregates on grouping extension queries. Usually, enabling this parameter generates a cheaper query plan that the planner will use in preference to existing plan.	runtime	master
gp_enable_multiphase_agg	boolean	on	Enables or disables the query planner's use of two or three-stage parallel aggregation plans. This approach applies to any subquery with aggregation. If <code>gp_enable_multiphase_agg</code> is off, then <code>gp_enable_agg_distinct</code> and <code>gp_enable_agg_distinct_pruning</code> are disabled.	runtime	master
gp_enable_predicate_propagation	boolean	on	When enabled, the query planner applies query predicates to both table expressions in cases where the tables are joined on their distribution key column(s). Filtering both tables prior to doing the join (when possible) is more efficient.	runtime	master
gp_enable_preunique	boolean	on	Enables two-phase duplicate removal for SELECT DISTINCT queries (not SELECT COUNT(DISTINCT)). When enabled, it adds an extra SORT DISTINCT set of plan nodes before motioning. In cases where the distinct operation greatly reduces the number of rows, this extra SORT DISTINCT is much cheaper than the cost of sending the rows across the Interconnect.	runtime	master
gp_enable_sequential_window_plans	boolean	on	If on, enables non-parallel (sequential) query plans for queries containing window function calls. If off, evaluates compatible window functions in parallel and rejoins the results. This is an experimental parameter.	runtime	master
gp_enable_sort_distinct	boolean	on	Enable duplicates to be removed while sorting.	runtime	master

Table D.2 Server Configuration Parameters

Parameter	Value Range	Default	Description	Settable by	GPDB
gp_enable_sort_limit	boolean	on	Enable LIMIT operation to be performed while sorting. Sorts more efficiently when the plan requires the first <i>limit_number</i> of rows at most.	runtime	master
gp_external_enable_exec	boolean	on	Enables or disables the use of external tables that execute OS commands or scripts on the segment hosts (<code>CREATE EXTERNAL TABLE EXECUTE syntax</code>).	superuser server start postgresql.conf	master
gp_external_grant_privileges	boolean	off	Enables or disables non-superusers to issue a <code>CREATE EXTERNAL [WEB] TABLE</code> command in cases where the <code>LOCATION</code> clause specifies <code>http</code> or <code>gpfdist</code> .	superuser server start postgresql.conf	master
gp_external_max_segs	integer	64	Sets the number of segments that will scan external table data during an external table operation, the purpose being not to overload the system with scanning data and take away resources from other concurrent operations. This only applies to external tables that use the <code>gpfdist://</code> protocol to access external table data.	superuser server start postgresql.conf	master
gp_fault_action	none readonly continue	readonly	Sets system behavior when a primary or mirror segment goes down. <i>readonly</i> means DDL/DML statements are not allowed if a segment is down. <i>continue</i> means all operations will continue as long as there is one active segment instance alive per content segment.	superuser server start postgresql.conf	global
gp_fts_probe_interval	10 seconds or greater	1min	Specifies the cycle time for the <code>fts_prober</code> process. Another complete probe of all segments starts each time this period expires. The <code>fts_prober</code> process will take approximately this amount of time to detect segment failures. Typically, this parameter is set to the same value as <code>gp_segment_connect_timeout</code> .	superuser server start postgresql.conf	global
gp_fts_probe_threadcount	1 - 128	5	Specifies the number of <code>fts_prober</code> threads to create. This parameter should be set to a value equal to or greater than the number of segments per host.	superuser server start postgresql.conf	global
gp_gpperfmon_send_interval	number of seconds	1	Sets the frequency that backend server processes send updates to the Greenplum Performance Monitor process (<code>gpperfmon</code>).	superuser server start postgresql.conf	global

Table D.2 Server Configuration Parameters

Parameter	Value Range	Default	Description	Settable by	GPDB
gp_hashagg_compress_spill_files	nothing zlib threaded_nothing threaded_zlib	nothing	When a hash aggregation operation spills to disk during query processing, specifies the compression algorithm to use on the spill files. If using zlib, it must be in your \$PATH on all segments.	runtime	master
gp_hashjoin_tuples_per_bucket	integer	5	Sets the target density of the hash table used by HashJoin operations. A smaller value will tend to produce larger hash tables, which can increase join performance.	runtime	master
gp_interconnect_hash_multiplier	2-25	2	Sets the size of the hash table used by the UDP interconnect to track connections. This number is multiplied by the number of segments to determine the number of buckets in the hash table. Increasing the value may increase interconnect performance for complex multi-slice queries (while consuming slightly more memory on the segment hosts).	runtime	global
gp_interconnect_queue_depth	1-256	4	Sets the amount of data per-peer to be queued by the UDP interconnect on receivers (when data is received but no space is available to receive it the data will be dropped, and the transmitter will need to resend it). Increasing the depth from its default value will cause the system to use more memory; but may increase performance. It is reasonable for this to be set between 1 and 10. Queries with data skew potentially perform better when this is increased. Increasing this may radically increase the amount of memory used by the system.	runtime	global
gp_interconnect_setup_timeout	0- <i>n</i> seconds	20s	Number of seconds to wait for the Interconnect to complete setup before it times out.	runtime	global
gp_interconnect_type	TCP UDP	UDP	Sets the networking protocol used for Interconnect traffic. With the TCP protocol, Greenplum Database has an upper limit of 1000 segment instances - less than that if the query workload involves complex, multi-slice queries. UDP allows for greater interconnect scalability. Note that the Greenplum software does the additional packet verification and checking not performed by UDP, so reliability and performance is equivalent to TCP.	runtime	global

Table D.2 Server Configuration Parameters

Parameter	Value Range	Default	Description	Settable by	GPDB
gp_log_format	csv text	csv	Specifies the format of the server log files. If using the administrative jetpack tools, the log files must be in csv format.	superuser server start postgresql.conf	local
gp_log_gang	OFF TERSE VERBOSE DEBUG	TERSE	Controls the logging for query executor worker process events. TERSE and VERBOSE messages are written with LOG severity level. The DEBUG messages are written with severity levels DEBUG1-5. TERSE logs creation and destruction of query executor processes. VERBOSE additionally logs the assignment of query executor process gangs to query slices before dispatch.	runtime	global
gp_log_interconnect	OFF TERSE VERBOSE DEBUG	TERSE	Controls the logging of Greenplum Interconnect events. TERSE and VERBOSE messages are written with LOG severity level. The DEBUG messages are written with severity levels DEBUG1-3. OFF suppresses normal status messages (errors and unusual occurrences are logged regardless).	runtime	global
gp_max_csv_line_length	number of bytes	65536	The maximum length of a line in a CSV formatted file that will be imported into the system. May need to be increased if using the jetpack administrative views. Maximum allowed is 1048575 (1MB).	postgresql server start	local
gp_max_local_distributed_cache	integer	1024	Sets the number of local to distributed transactions to cache. Higher settings may improve performance.	postgresql server start	local
gp_max_packet_size	512-65536	8192	Sets the size (in kilobytes) of messages sent by the UDP interconnect, and sets the tuple-serialization chunk size for both the UDP and TCP interconnect.	superuser server start postgresql.conf	global
gp_motion_cost_per_row	floating point	0	Sets the query planner cost estimate for a Motion operator to transfer a row from one segment to another, measured as a fraction of the cost of a sequential page fetch. If 0, then the value used is two times the value of <i>cpu_tuple_cost</i> .	runtime	master
gp_reject_percent_threshold	1- <i>n</i>	300	For single row error handling on COPY and external table SELECTs, sets the number of rows processed before SEGMENT REJECT LIMIT <i>n</i> PERCENT starts calculating.	runtime	master

Table D.2 Server Configuration Parameters

Parameter	Value Range	Default	Description	Settable by	GPDB
gp_reraise_signal	boolean	on	If enabled, will attempt to dump core if a fatal server error occurs.	runtime	global
gp_role	dispatch execute utility		The role of this server process — set to <i>dispatch</i> for the master and <i>execute</i> for a segment.	read only	local
gp_safeswriteseize	integer	0	Specifies a minimum size for safe write operations to append-only tables in a non-mature file system. When a number of bytes greater than zero is specified, the append-only writer adds padding data up to that number in order to prevent data corruption due to file system errors. Each non-mature file system has a known safe write size that must be specified here when using Greenplum Database with that type of file system. This is commonly set to a multiple of the extent size of the file system; for example, Linux ext3 is 4096 bytes, so a value of 32768 is commonly used.	superuser server start postgresql.conf	local
gp_segment_connect_timeout	0- <i>n</i> seconds	1min	Time that the Greenplum Interconnect will try to connect to a segment instance over the network before timing out.	runtime	global
gp_segments_for_planner	0- <i>n</i>	0	Sets the number of primary segment instances for the planner to assume in its cost and size estimates. If 0, then the value used is the actual number of primary segments. This variable affects the planner's estimates of the number of rows handled by each sending and receiving process in Motion operators.	runtime	master
gp_session_id	1- <i>n</i>		A system assigned ID number for a client session. Starts counting from 1 when the master instance is first started.	read only	master
gp_set_proc_affinity	boolean	off	If enabled, when a Greenplum server process (postmaster) is started it will bind to a CPU.	superuser server start postgresql.conf	global
gp_set_read_only	boolean	off	Set to on to disable writes to the database. Any in progress transactions must finish before read-only mode takes affect.	runtime	global
gp_statistics_pullup_from_child_partition	boolean	on	Enables the query planner to utilize statistics from child tables when planning queries on the parent table.	runtime	master

Table D.2 Server Configuration Parameters

Parameter	Value Range	Default	Description	Settable by	GPDB
gp_statistics_use_fkeys	boolean	on	Allows the optimizer to use foreign key information stored in the system catalog to optimize joins between foreign keys and primary keys. Even though referential integrity is not enforced in Greenplum Database, foreign keys can still be declared in DDL statements.	runtime	master
gp_use_dispatch_agent	boolean	off	This parameter is for a feature that is currently under development and should be left to off in 3.3.7.	runtime	global
gp_vmem_protect_gang_cache_limit	number of megabytes	500	If a query executor process consumes more than this configured amount, then the process will not be cached for use in subsequent queries after the process completes. Systems with lots of connections or idle processes may want to reduce this number to free more memory on the segments. Note that this is a local parameter and must be set for every segment.	superuser server start postgresql.conf	local
gp_vmem_protect_limit	integer	8192	Sets the amount of memory (in number of MBs) that all postgres processes of a segment instance can consume. To prevent over allocation of memory, set to: $(\text{physical_memory}/\text{segments}) + (\text{swap_space}/\text{segments}/2)$ For example, on a segment host with 16GB physical memory, 16GB swap space, and 4 segment instances the calculation would be: $(16/4) + (16/4/2) = 6\text{GB}$ $6 * 1024 = 6144\text{MB}$ If a query causes this limit to be exceeded, memory will not be allocated and the query will fail. Note that this is a local parameter and must be set for every segment.	superuser server start postgresql.conf	local
gpperfmon_port	integer	8888	Sets the port on which all performance monitor agents communicate with the master.	superuser server start postgresql.conf	global
hba_file	string	<data_directory>/pg_hba.conf	Specifies the configuration file for host-based authentication (pg_hba.conf).	superuser server start postgresql.conf	local
ident_file	string	<data_directory>/pg_ident.conf	Specifies the configuration file for ident authentication (pg_ident.conf).	superuser server start postgresql.conf	local

Table D.2 Server Configuration Parameters

Parameter	Value Range	Default	Description	Settable by	GPDB
integer_datetimes	boolean	on	Reports whether PostgreSQL was built with support for 64-bit-integer dates and times.	read-only	-
IntervalStyle	postgres postgres_verbose sql_standard iso_8601	postgres	Sets the display format for interval values. The value <i>sql_standard</i> produces output matching SQL standard interval literals. The value <i>postgres</i> produces output matching PostgreSQL releases prior to 8.4 when the <i>DateStyle</i> parameter was set to ISO. The value <i>postgres_verbose</i> produces output matching Greenplum releases prior to 3.3 when the <i>DateStyle</i> parameter was set to non-ISO output. The value <i>iso_8601</i> will produce output matching the time interval <i>format with designators</i> defined in section 4.4.3.2 of ISO 8601. See the PostgreSQL 8.4 documentation for more information.	runtime	global
join_collapse_limit	1- <i>n</i>	8	The planner will rewrite explicit inner JOIN constructs into lists of FROM items whenever a list of no more than this many items in total would result. By default, this variable is set the same as <i>from_collapse_limit</i> , which is appropriate for most uses. Setting it to 1 prevents any reordering of inner JOINS. Setting this variable to a value between 1 and <i>from_collapse_limit</i> might be useful to trade off planning time against the quality of the chosen plan (higher values produce better plans).	runtime	master
krb_caseins_users	boolean	off	Sets whether Kerberos user names should be treated case-insensitively. The default is case sensitive (off).	superuser server start postgresql.conf	master
krb_server_hostname	host name	unset	Sets the host name part of the service principal. This, combined with <i>krb_srvname</i> , is used to generate the complete service principal, that is <i>krb_srvname/krb_server_hostname@REALM</i> .	superuser server start postgresql.conf	master
krb_server_keyfile	path and file name	unset	Sets the location of the Kerberos server key file.	superuser server start postgresql.conf	master
krb_srvname	service name	postgres	Sets the Kerberos service name.	superuser server start postgresql.conf	master

Table D.2 Server Configuration Parameters

Parameter	Value Range	Default	Description	Settable by	GPDB
lc_collate	<system dependent>	en_US.UTF-8	Reports the locale in which sorting of textual data is done. The value is determined when the Greenplum Database array is initialized.	read only	local
lc_ctype	<system dependent>	en_US.utf8	Reports the locale that determines character classifications. The value is determined when the Greenplum Database array is initialized.	read only	local
lc_messages	<system dependent>	en_US.utf8	Sets the language in which messages are displayed. The locales available depends on what was installed with your operating system - use <i>locale -a</i> to list available locales. The default value is inherited from the execution environment of the server. On some systems, this locale category does not exist. Setting this variable will still work, but there will be no effect. Also, there is a chance that no translated messages for the desired language exist. In that case you will continue to see the English messages.	superuser server start postgresql.conf	local
lc_monetary	<system dependent>	en_US.utf8	Sets the locale to use for formatting monetary amounts, for example with the <i>to_char</i> family of functions. The locales available depends on what was installed with your operating system - use <i>locale -a</i> to list available locales. The default value is inherited from the execution environment of the server.	superuser server start postgresql.conf	local
lc_numeric	<system dependent>	en_US.utf8	Sets the locale to use for formatting numbers, for example with the <i>to_char</i> family of functions. The locales available depends on what was installed with your operating system - use <i>locale -a</i> to list available locales. The default value is inherited from the execution environment of the server.	superuser server start postgresql.conf	local
lc_time	<system dependent>	en_US.utf8	This parameter currently does nothing, but may in the future.	superuser server start postgresql.conf	local
listen_addresses	localhost, host names, IP addresses, *	*	Specifies the TCP/IP address(es) on which the server is to listen for connections from client applications - a comma-separated list of host names and/or numeric IP addresses. The special entry * corresponds to all available IP interfaces. If the list is empty, only Unix-domain sockets can connect.	superuser server start postgresql.conf	master

Table D.2 Server Configuration Parameters

Parameter	Value Range	Default	Description	Settable by	GPDB
local_preload_libraries			Comma separated list of shared library files to preload at the start of a client session.	superuser server start postgresql.conf	local
log_autostats	boolean	on	Logs information about automatic <code>ANALYZE</code> operations related to <code>gp_autostats_mode</code> and <code>gp_autostats_on_change_threshold</code> .	superuser runtime	global
log_connections	boolean	off	This outputs a line to the server log detailing each successful connection. Some client programs, like <code>psql</code> , attempt to connect twice while determining if a password is required, so duplicate “connection received” messages do not always indicate a problem.	superuser server start postgresql.conf	local
log_disconnections	boolean	off	This outputs a line in the server log at termination of a client session, and includes the duration of the session.	superuser server start postgresql.conf	local
log_dispatch_stats	boolean	off	When set to “on,” this parameter adds a log message with verbose information about the dispatch of the statement.	superuser server start postgresql.conf	master
log_duration	boolean	off	Causes the duration of every completed statement which satisfies <code>log_statement</code> to be logged.	superuser runtime	global
log_error_verbosity	TERSE DEFAULT VERBOSE	DEFAULT	Controls the amount of detail written in the server log for each message that is logged.	superuser runtime	global
log_executor_stats	boolean	off	For each query, write performance statistics of the query executor to the server log. This is a crude profiling instrument. Cannot be enabled together with <code>log_statement_stats</code> .	superuser server start postgresql.conf	local
log_filename	string	gpdb-%Y-%m-%d_%H%M%S.csv	Sets the file names of the created log files. %-escapes can be used to specify time-varying file names. Log files are located in <code>pg_log</code> of the master or segment data directory. The logs are rotated daily and named: <code>gpdb-YYYY-MM-DD_HHMMSS.log</code>	superuser server start postgresql.conf	local
log_hostname	boolean	off	By default, connection log messages only show the IP address of the connecting host. Turning on this option causes logging of the host name as well. Note that depending on your host name resolution setup this might impose a non-negligible performance penalty.	superuser server start postgresql.conf	local

Table D.2 Server Configuration Parameters

Parameter	Value Range	Default	Description	Settable by	GPDB
log_min_duration_statement	number of milliseconds, 0, -1	-1	Logs the statement and its duration on a single log line if its duration is greater than or equal to the specified number of milliseconds. Setting this to 0 will print all statements and their durations. -1 disables the feature. For example, if you set it to 250 then all SQL statements that run 250ms or longer will be logged. Enabling this option can be useful in tracking down unoptimized queries in your applications.	superuser runtime	global
log_min_error_statement	DEBUG5 DEBUG4 DEBUG3 DEBUG2, DEBUG1 INFO NOTICE WARNING ERROR FATAL PANIC	ERROR	Controls whether or not the SQL statement that causes an error condition will also be recorded in the server log. All SQL statements that cause an error of the specified level or higher are logged. The default is PANIC (effectively turning this feature off for normal use). Enabling this option can be helpful in tracking down the source of any errors that appear in the server log.	superuser runtime	global
log_min_messages	DEBUG5 DEBUG4 DEBUG3 DEBUG2 DEBUG1 INFO NOTICE WARNING ERROR LOG FATAL PANIC	NOTICE	Controls which message levels are written to the server log. Each level includes all the levels that follow it. The later the level, the fewer messages are sent to the log.	superuser runtime	global
log_parser_stats	boolean	off	For each query, write performance statistics of the query parser to the server log. This is a crude profiling instrument. Cannot be enabled together with <i>log_statement_stats</i> .	superuser runtime	global
log_planner_stats	boolean	off	For each query, write performance statistics of the query planner to the server log. This is a crude profiling instrument. Cannot be enabled together with <i>log_statement_stats</i> .	superuser runtime	global
log_rotation_age	number of minutes	1d	Determines the maximum lifetime of an individual log file. After this time has elapsed, a new log file will be created. Set to zero to disable time-based creation of new log files.	superuser server start postgresql.conf	local

Table D.2 Server Configuration Parameters

Parameter	Value Range	Default	Description	Settable by	GPDB
log_rotation_size	number of kilobytes	0	Determines the maximum size of an individual log file. After this many kilobytes have been emitted into a log file, a new log file will be created. Set to zero to disable size-based creation of new log files.	superuser server start postgresql.conf	local
log_statement	NONE DDL MOD ALL	ALL	Controls which SQL statements are logged. DDL logs all data definition commands like CREATE, ALTER, and DROP commands. MOD logs all DDL statements, plus INSERT, UPDATE, DELETE, TRUNCATE, and COPY FROM. PREPARE and EXPLAIN ANALYZE statements are also logged if their contained command is of an appropriate type.	superuser runtime	local
log_statement_stats	boolean	off	For each query, write total performance statistics of the query parser, planner, and executor to the server log. This is a crude profiling instrument.	superuser runtime	global
log_timezone	string	unknown	Sets the time zone used for timestamps written in the log. Unlike TimeZone , this value is system-wide, so that all sessions will report timestamps consistently. The default is <code>unknown</code> , which means to use whatever the system environment specifies as the time zone.	superuser server start postgresql.conf	local
log_truncate_on_rotation	boolean	off	Truncates (overwrites), rather than appends to, any existing log file of the same name. Truncation will occur only when a new file is being opened due to time-based rotation. For example, using this setting in combination with a log_filename such as <code>gpseg#-%H.log</code> would result in generating twenty-four hourly log files and then cyclically overwriting them. When off, pre-existing files will be appended to in all cases.	superuser server start postgresql.conf	local

Table D.2 Server Configuration Parameters

Parameter	Value Range	Default	Description	Settable by	GPDB
<code>maintenance_work_mem</code>	number of kilobytes	64MB	Specifies the maximum amount of memory (in kilobytes) to be used in maintenance operations, such as VACUUM or CREATE INDEX. Since only one of these operations can be executed at a time by a database session, and an installation normally doesn't have very many of them happening concurrently, it's safe to set this value significantly larger than <code>work_mem</code> . Larger settings may improve performance for vacuuming and for restoring database dumps.	runtime	global
<code>max_appendonly_tables</code>	2048		Sets the maximum number of append-only relations that can be written to or loaded concurrently. Append-only table partitions and subpartitions are considered as unique tables against this limit. Increasing the limit will allocate more shared memory at server start.	superuser server start postgresql.conf	global
<code>max_connections</code>	1- <i>n</i>	25 on master 125 on segments	The maximum number of concurrent connections to the database server. In a Greenplum Database system, user client connections go through the Greenplum master instance only. Segment instances should allow 5-10 times the amount as the master. When you increase this parameter, <code>max_prepared_transactions</code> must be increased as well. For more information, see Limiting Concurrent Connections . Increasing this parameter may cause Greenplum Database to request more shared memory.	superuser server start postgresql.conf	local
<code>max_files_per_process</code>	integer	1000	Sets the maximum number of simultaneously open files allowed to each server subprocess. If the kernel is enforcing a safe per-process limit, you don't need to worry about this setting. Some platforms such as BSD, the kernel will allow individual processes to open many more files than the system can really support.	superuser server start postgresql.conf	local
<code>max_fsm_pages</code>	integer > 16 * <code>max_fsm_relations</code>	200000	Sets the maximum number of disk pages for which free space will be tracked in the shared free-space map. Six bytes of shared memory are consumed for each page slot.	superuser server start postgresql.conf	local

Table D.2 Server Configuration Parameters

Parameter	Value Range	Default	Description	Settable by	GPDB
max_fsm_relations	integer	1000	Sets the maximum number of relations for which free space will be tracked in the shared memory free-space map. Should be set to a value larger than the total number of: tables + indexes + system tables. It costs about 60 bytes of memory for each relation per segment instance. It is better to allow some room for overhead and set too high rather than too low.	superuser server start postgresql.conf	local
max_function_args	integer	100	Reports the maximum number of function arguments.	read only	-
max_identifier_length	integer	63	Reports the maximum identifier length.	read only	-
max_index_keys	integer	32	Reports the maximum number of index keys.	read only	-
max_locks_per_transaction	integer	64	The shared lock table is created with room to describe locks on <i>max_locks_per_transaction</i> * (<i>max_connections</i> + <i>max_prepared_transactions</i>) objects, so no more than this many distinct objects can be locked at any one time. This is not a hard limit on the number of locks taken by any one transaction, but rather a maximum average value. You might need to raise this value if you have clients that touch many different tables in a single transaction.	superuser server start postgresql.conf	local
max_prepared_transactions	integer	50 on master 50 on segments	Sets the maximum number of transactions that can be in the prepared state simultaneously. Greenplum uses prepared transactions internally to ensure data integrity across the segments. This value must be at least as large as the value of max_connections on the master. Segment instances should be set to the same value as the master. For more information, see Limiting Concurrent Connections .	superuser server start postgresql.conf	local
max_resource_portals_per_transaction	integer	64	Sets the maximum number of simultaneously open user-declared cursors allowed per transaction. Note that an open cursor will hold an active query slot in a resource queue. Used for workload management.	superuser server start postgresql.conf	master

Table D.2 Server Configuration Parameters

Parameter	Value Range	Default	Description	Settable by	GPDB
max_resource_queues	integer	8	Sets the maximum number of resource queues that can be created in a Greenplum Database system. Note that resource queues are system-wide (as are roles) so they apply to all databases in the system.	superuser server start postgresql.conf	master
max_stack_depth	number of kilobytes	2MB	Specifies the maximum safe depth of the server's execution stack. The ideal setting for this parameter is the actual stack size limit enforced by the kernel (as set by <i>ulimit -s</i> or local equivalent), less a safety margin of a megabyte or so. Setting the parameter higher than the actual kernel limit will mean that a runaway recursive function can crash an individual backend process.	superuser server start postgresql.conf	local
password_encryption	boolean	on	When a password is specified in CREATE USER or ALTER USER without writing either ENCRYPTED or UNENCRYPTED, this option determines whether the password is to be encrypted.	runtime	master
port	any valid port number	5432	The database listener port for a Greenplum instance. The master and each segment has its own port. Port numbers for the Greenplum system must also be changed in the gp_configuration catalog. You must shut down your Greenplum Database system before changing port numbers.	superuser server start postgresql.conf	local
random_page_cost	floating point	100	Sets the planner's estimate of the cost of a nonsequentially fetched disk page. This is measured as a multiple of the cost of a sequential page fetch. A higher value makes it more likely a sequential scan will be used, a lower value makes it more likely an index scan will be used.	runtime	master
regex_flavor	advanced extended basic	advanced	The 'extended' setting may be useful for exact backwards compatibility with pre-7.4 releases of PostgreSQL.	runtime	master
resource_cleanup_gangs_on_wait	boolean	on	If a statement is submitted through a resource queue, clean up any idle query executor worker processes before taking a lock on the resource queue.	superuser server start postgresql.conf	global
resource_scheduler	boolean	on	Enables or disables the resource queue workload management feature in Greenplum Database.	superuser server start postgresql.conf	master

Table D.2 Server Configuration Parameters

Parameter	Value Range	Default	Description	Settable by	GPDB
resource_select_only	boolean	on	Sets the types of queries managed by resource queues. If set to on, then SELECT, SELECT INTO, CREATE TABLE AS SELECT, and DECLARE CURSOR commands are evaluated. If set to off INSERT, UPDATE, and DELETE commands will be evaluated as well.	superuser server start postgresql.conf	master
search_path	a comma-separated list of schema names	\$user,public	Specifies the order in which schemas are searched when an object is referenced by a simple name with no schema component. When there are objects of identical names in different schemas, the one found first in the search path is used. The system catalog schema, <i>pg_catalog</i> , is always searched, whether it is mentioned in the path or not. When objects are created without specifying a particular target schema, they will be placed in the first schema listed in the search path. The current effective value of the search path can be examined via the SQL function <i>current_schemas()</i> . <i>current_schemas()</i> shows how the requests appearing in <i>search_path</i> were resolved.	runtime	global
seq_page_cost	floating point	1	Sets the planner's estimate of the cost of a disk page fetch that is part of a series of sequential fetches.	runtime	master
server_encoding	<system dependent>	UTF8	Reports the database encoding (character set). It is determined when the Greenplum Database array is initialized. Ordinarily, clients need only be concerned with the value of <i>client_encoding</i> .	read only	master
server_version	string	8.2.14	Reports the version of PostgreSQL that this release of Greenplum Database is based on.	read only	master
server_version_num	integer	80214	Reports the version of PostgreSQL that this release of Greenplum Database is based on as an integer.	read only	master
shared_buffers	integer > 16K * <i>max_connections</i>	125MB	Sets the amount of memory a Greenplum server instance uses for shared memory buffers. This setting must be at least 128 kilobytes and at least 16 kilobytes times <i>max_connections</i> .	superuser server start postgresql.conf	local

Table D.2 Server Configuration Parameters

Parameter	Value Range	Default	Description	Settable by	GPDB
shared_preload_libraries			A comma-separated list of shared libraries that are to be preloaded at server start. PostgreSQL procedural language libraries can be preloaded in this way, typically by using the syntax '\$libdir/plXXX' where XXX is pgsql, perl, tcl, or python. By preloading a shared library, the library startup time is avoided when the library is first used. If a specified library is not found, the server will fail to start.	superuser server start postgresql.conf	local
silent_mode	boolean	on	Runs the server silently. If this option is set, the server will automatically run in background and any controlling terminals are disassociated. The server's standard output and standard error are redirected to /dev/null, so any messages sent to them will be lost.	superuser server start postgresql.conf	local
sql_inheritance	boolean	on	This controls the inheritance semantics, in particular whether subtables are included by various commands by default. They were not included in versions prior to PostgreSQL 7.1.	superuser server start postgresql.conf	master
ssl	boolean	off	Enables SSL connections.	superuser server start postgresql.conf	master
standard_conforming_strings	boolean	off	Reports whether ordinary string literals ('...') treat backslashes literally, as specified in the SQL standard. The value is currently always off, indicating that backslashes are treated as escapes. It is planned that this will change to on in a future release when string literal syntax changes to meet the standard. Applications may check this parameter to determine how string literals will be processed. The presence of this parameter can also be taken as an indication that the escape string syntax (E'...') is supported.	read only	master
statement_timeout	number of milliseconds	0	Abort any statement that takes over the specified number of milliseconds. 0 turns off the limitation.	runtime	global

Table D.2 Server Configuration Parameters

Parameter	Value Range	Default	Description	Settable by	GPDB
stats_block_level	boolean	off	Enables the collection of block-level statistics on database activity. If enabled, the data that is produced can be accessed via the <i>pg_stat</i> and <i>pg_statio</i> family of system views.	runtime	local
stats_command_string	boolean	on	Enables the collection of statistics on the currently executing command of each session, along with the time at which that command began execution. When enabled, this information is not visible to all users, only to superusers and the user owning the session. This data can be accessed via the <i>pg_stat_activity</i> system view.	runtime	master
stats_queue_level	boolean	off	Collects resource queue statistics on database activity.	runtime	master
stats_reset_server_on_start	boolean	off	If on, collected statistics are zeroed out whenever the server is restarted. If off, statistics are accumulated across server restarts.	superuser server start postgresql.conf	local
stats_row_level	boolean	off	Enables the collection of row-level statistics on database activity. If enabled, the data that is produced can be accessed via the <i>pg_stat</i> and <i>pg_statio</i> family of system views.	superuser server start postgresql.conf	local
stats_start_collector	boolean	on	Controls whether the server should start the statistics-collection subprocess. Note that statistics collection is not global in Greenplum Database. Statistics are only collected at the local segment level, which is of limited usefulness.	superuser server start postgresql.conf	local
superuser_reserved_connections	integer < <i>max_connections</i>	3	Determines the number of connection slots that are reserved for Greenplum Database superusers.	superuser server start postgresql.conf	local
tcp_keepalives_count	number of lost keepalives	0	How many keepalives may be lost before the connection is considered dead. A value of 0 uses the system default. If TCP_KEEPCNT is not supported, this parameter must be 0.	superuser server start postgresql.conf	local
tcp_keepalives_idle	number of seconds	0	Number of seconds between sending keepalives on an otherwise idle connection. A value of 0 uses the system default. If TCP_KEEPIDLE is not supported, this parameter must be 0.	superuser server start postgresql.conf	local

Table D.2 Server Configuration Parameters

Parameter	Value Range	Default	Description	Settable by	GPDB
tcp_keepalives_interval	number of seconds	0	How many seconds to wait for a response to a keepalive before retransmitting. A value of 0 uses the system default. If TCP_KEEPINTVL is not supported, this parameter must be 0.	superuser server start postgresql.conf	local
temp_buffers	integer	1024	Sets the maximum number of temporary buffers used by each database session. These are session-local buffers used only for access to temporary tables. The setting can be changed within individual sessions, but only up until the first use of temporary tables within a session. The cost of setting a large value in sessions that do not actually need a lot of temporary buffers is only a buffer descriptor, or about 64 bytes, per increment. However if a buffer is actually used, an additional 8192 bytes will be consumed.	runtime	global
TimeZone	time zone abbreviation		Sets the time zone for displaying and interpreting time stamps. The default is to use whatever the system environment specifies as the time zone. See Date/Time Keywords in the PostgreSQL documentation.	runtime	global
timezone_abbreviations	string	Default	Sets the collection of time zone abbreviations that will be accepted by the server for date time input. The default is <code>Default</code> , which is a collection that works in most of the world. <code>Australia</code> and <code>India</code> , and other collections can be defined for a particular installation. Possible values are names of configuration files stored in <code>/share/timezonesets/</code> in the installation directory.	runtime	global
transaction_isolation	read committed serializable	read committed	Sets the current transaction's isolation level.	runtime	global
transaction_read_only	boolean	off	Sets the current transaction's read-only status.	runtime	global
transform_null_equals	boolean	off	When on, expressions of the form <code>expr = NULL</code> (or <code>NULL = expr</code>) are treated as <code>expr IS NULL</code> , that is, they return true if <code>expr</code> evaluates to the null value, and false otherwise. The correct SQL-spec-compliant behavior of <code>expr = NULL</code> is to always return null (unknown).	runtime	master

Table D.2 Server Configuration Parameters

Parameter	Value Range	Default	Description	Settable by	GPDB
unix_socket_directory	directory path	unset	Specifies the directory of the Unix-domain socket on which the server is to listen for connections from client applications.	superuser server start postgresql.conf	local
unix_socket_group	Unix group name	unset	Sets the owning group of the Unix-domain socket. By default this is an empty string, which uses the default group for the current user.	superuser server start postgresql.conf	local
unix_socket_permissions	numeric Unix file permission mode (as accepted by the <i>chmod</i> or <i>umask</i> commands)	511	Sets the access permissions of the Unix-domain socket. Unix-domain sockets use the usual Unix file system permission set. Note that for a Unix-domain socket, only write permission matters.	superuser server start postgresql.conf	local
update_process_title	boolean	on	Enables updating of the process title every time a new SQL command is received by the server. The process title is typically viewed by the <code>ps</code> command.	superuser server start postgresql.conf	local
vacuum_cost_delay	milliseconds < 0 (in multiples of 10)	0	The length of time that the process will sleep when the cost limit has been exceeded. 0 disables the cost-based vacuum delay feature.	superuser server start postgresql.conf	local
vacuum_cost_limit	integer > 0	200	The accumulated cost that will cause the vacuuming process to sleep.	superuser server start postgresql.conf	local
vacuum_cost_page_dirty	integer > 0	20	The estimated cost charged when vacuum modifies a block that was previously clean. It represents the extra I/O required to flush the dirty block out to disk again.	superuser server start postgresql.conf	local
vacuum_cost_page_hit	integer > 0	1	The estimated cost for vacuuming a buffer found in the shared buffer cache. It represents the cost to lock the buffer pool, lookup the shared hash table and scan the content of the page.	superuser server start postgresql.conf	local
vacuum_cost_page_miss	integer > 0	10	The estimated cost for vacuuming a buffer that has to be read from disk. This represents the effort to lock the buffer pool, lookup the shared hash table, read the desired block in from the disk and scan its content.	superuser server start postgresql.conf	local

Table D.2 Server Configuration Parameters

Parameter	Value Range	Default	Description	Settable by	GPDB
<code>vacuum_freeze_min_age</code>	integer 0-100000000000	10000000 0	Specifies the cutoff age (in transactions) that VACUUM should use to decide whether to replace transaction IDs with <i>FrozenXID</i> while scanning a table. VACUUM will limit the effective value to half the value of <code>autovacuum_freeze_max_age</code> , so that there is not an unreasonably short time between forced autovacua.	superuser server start postgresql.conf	local
<code>wal_buffers</code>	number of buffers	256kB	Number of disk-page buffers allocated in shared memory for WAL data. The setting need only be large enough to hold the amount of WAL data generated by one typical transaction, since the data is written out to disk at every transaction commit.	superuser server start postgresql.conf	local
<code>wal_sync_method</code>	<code>open_datasync</code> <code>fdasync</code> <code>fsync_writethrough</code> <code>fsync</code> <code>open_sync</code>	<code>fdasync</code>	Method used for forcing WAL updates out to disk if <code>fsynch</code> is enabled. Not all of these choices are available on all platforms. The default is the first method in the list that is supported.	superuser server start postgresql.conf	local
<code>work_mem</code>	number of kilobytes	32MB	Specifies the amount of memory to be used by internal sort operations and hash tables before switching to temporary disk files. Note that for a complex query, several sort or hash operations might be running in parallel; each one will be allowed to use as much memory as this value specifies before it starts to put data into temporary files. Also, several running sessions could be doing such operations concurrently. So the total memory used could be many times the value of <code>work_mem</code> ; it is necessary to keep this fact in mind when choosing the value. Sort operations are used for ORDER BY, DISTINCT, and merge joins. Hash tables are used in hash joins, hash-based aggregation, and hash-based processing of IN subqueries.	runtime	global

E. Initialization Configuration File Reference

A configuration file is needed by the `gpinitssystem` utility to tell it how to initialize and configure the master and segment instances in your Greenplum Database system. This file is referred to as `gp_init_config` in this documentation, but can be named whatever you like. This file is referenced when you run `gpinitssystem` with the `-c` option. An example configuration file can be found in `$GPHOME/docs/cli_help/gp_init_config_example`.

Required Parameters

ARRAY_NAME

A name for the array you are configuring. You can use any name you like. Enclose the name in quotes if the name contains spaces.

Example: `ARRAY_NAME="GreenplumDB"`

MACHINE_LIST_FILE

This specifies the file that contains the list of segment host names that comprise the Greenplum system. The master host is assumed to be the host from which you are running the utility and should not be included in this file. If you have a multi-NIC configuration and have multiple host names configured for each segment host, this file should have ALL configured segment host names. Give the absolute path to the file.

Example: `MACHINE_LIST_FILE=/home/gpadmin/multi_seg_host_file`

SEG_PREFIX

This specifies a prefix that will be used to name the data directories on the master and segment instances. The naming convention for data directories in a Greenplum Database system is `SEG_PREFIXnumber` where *number* starts with 0 for segment instances (the master is always -1). So for example, if you choose the prefix `gp`, your master instance data directory would be named `gp-1`, and the segment instances would be named `gp0`, `gp1`, `gp2`, `gp3`, and so on.

Example: `SEG_PREFIX=gp`

PORT_BASE

This specifies the base port number on which primary segment instances will be started on a segment host. If a host has multiple primary segment instances, the base port number will be incremented by one for each additional segment instance started on that host. Valid values range from 1 through 65535.

Example: `PORT_BASE=10000`

DATA_DIRECTORY

This specifies the data storage location(s) where the utility will create the primary segment data directories. The utility creates a unique data directory for each segment instance. If you want multiple segment instances per host, list a data storage area for each primary segment you want created. The recommended number is one primary segment per CPU. It is OK to list the same data

storage area multiple times if you want your data directories created in the same location. The number of data directory locations specified will determine the number of primary segment instances per host.

You must make sure that the user who runs `gpinitssystem` (for example, the `gpadmin` user) has permissions to write to these directories. You may want to create these directories on the segment hosts before running `gpinitssystem` and `chown` them to the appropriate user.

Example 1: `declare -a DATA_DIRECTORY=(/dbfast1 /dbfast2 /dbfast3 /dbfast4)`

Example 2: `declare -a DATA_DIRECTORY=(/gpdata/primary /gpdata/primary /gpdata/primary /gpdata/primary)`

MASTER_HOSTNAME

The host name of the master instance. This host name must exactly match the configured host name of the machine (run the `hostname` command to determine the correct hostname).

Example 1: `MASTER_HOSTNAME=host1`

Example 2: `MASTER_HOSTNAME=host1.company.com`

MASTER_DIRECTORY

This specifies the location where the data directory will be created on the master host.

You must make sure that the user who runs `gpinitssystem` (for example, the `gpadmin` user) has permissions to write to this directory. You may want to create this directory on the master host before running `gpinitssystem` and `chown` it to the appropriate user.

Example: `MASTER_DIRECTORY=/gpdata`

MASTER_PORT

The port number for the master instance. This is the port number that users and client connections will use when accessing the Greenplum Database system.

Example: `MASTER_PORT=5432`

TRUSTED_SHELL

The shell the `gpinitssystem` utility uses to execute commands on remote hosts. Allowed values are `ssh`. You must set up your trusted host environment before running the `gpinitssystem` utility (you can use `gpssh-exkeys` to do this).

Example: `TRUSTED_SHELL=ssh`

CHECK_POINT_SEGMENTS

Maximum distance between automatic write ahead log (WAL) checkpoints, in log file segments (each segment is normally 16 megabytes). This will set the `checkpoint_segments` parameter in the `postgresql.conf` file for each segment instance in the Greenplum Database system.

Example: `CHECK_POINT_SEGMENTS=8`

ENCODING

The character set encoding to use. Greenplum Database supports the same character sets as PostgreSQL. See “[Character Set Support](#)” on page 52.

Example: `ENCODING=UNICODE`

Optional Parameters

DATABASE_NAME

Optional. The name of a Greenplum Database database to create after the system is initialized. You can always create a database later using the `CREATE DATABASE` command or the `createdb` utility.

Example: `DATABASE_NAME=warehouse`

Optional Parameters for Mirror Segments

Note: You can also deploy mirrors later using `gpaddmirrors`.

MIRROR_PORT_BASE

This specifies the base port number on which mirror segment instances will be started on a segment host. If a host has multiple mirror segment instances, the base port number will be incremented by one for each additional mirror segment instance started on that host. Be sure to use a different number than the primary `PORT_BASE`, with a valid TCP port value from 1 through 65535.

Example: `MIRROR_PORT_BASE=20000`

MIRROR_DATA_DIRECTORY

This specifies the location where the data directory will be created on a host for a mirror segment instance. There must be the same number of data directories declared for mirror segment instances as for primary segment instances (see the `DATA_DIRECTORY` parameter).

You must make sure that the user who runs `gpinitssystem` (for example, the `gpadmin` user) has permissions to write to these directories. You may want to create these directories on the segment hosts before running `gpinitssystem` and `chown` them to the appropriate user.

Example 1: `declare -a MIRROR_DATA_DIRECTORY=(/dbfast5 /dbfast6 /dbfast7 /dbfast8)`

Example 2: `declare -a MIRROR_DATA_DIRECTORY=(/gpdata/mirror /gpdata/mirror /gpdata/mirror /gpdata/mirror)`

F. Greenplum MapReduce Specification

MapReduce is a programming model developed by Google for processing and generating large data sets on an array of commodity servers. Greenplum MapReduce allows programmers who are familiar with the MapReduce paradigm to write map and reduce functions and submit them to the Greenplum Database parallel engine for processing.

In order for Greenplum to be able to process MapReduce functions, the functions need to be defined in a document, which is then passed to the Greenplum MapReduce program, `gpmapreduce`, for execution by the Greenplum Database parallel engine. The Greenplum Database system takes care of the details of distributing the input data, executing the program across a set of machines, handling machine failures, and managing the required inter-machine communication.

This specification describes the document format and schema for defining Greenplum MapReduce jobs.

Greenplum MapReduce Document Format

This section explains some basics of the Greenplum MapReduce document format to help you get started creating your own Greenplum MapReduce documents. Greenplum uses the [YAML 1.1](#) document format and then implements its own schema for defining the various steps of a MapReduce job.

All Greenplum MapReduce files must first declare the version of the YAML specification they are using. After that, three dashes (---) denote the start of a document, and three dots (. . .) indicate the end of a document without starting a new one. Comment lines are prefixed with a pound symbol (#). It is possible to declare multiple Greenplum MapReduce documents in the same file:

```
%YAML 1.1
---
# Begin Document 1
# ...
---
# Begin Document 2
# ...
```

Within a Greenplum MapReduce document, there are three basic types of data structures or *nodes*: *scalars*, *sequences* and *mappings*.

A *scalar* is a basic string of text indented by a space. If you have a scalar input that spans multiple lines, a preceding pipe (|) denotes a *literal* style, where all line breaks are significant. Alternatively, a preceding angle bracket (>) folds a single line break

to a space for subsequent lines that have the same indentation level. If a string contains characters that have reserved meaning, the string must be quoted or the special character must be escaped with a backslash (\).

```
# Read each new line literally
somekey: |
    this value contains two lines
    and each line is read literally
# Treat each new line as a space
anotherkey: >
    this value contains two lines
    but is treated as one continuous line
# This quoted string contains a special character
ThirdKey: "This is a string: not a mapping"
```

A *sequence* is a list with each entry in the list on its own line denoted by a dash and a space (-). Alternatively, you can specify an inline sequence as a comma-separated list within square brackets. A sequence provides a set of data and gives it an order. When you load a list into the Greenplum MapReduce program, the order is kept.

```
# list sequence
- this
- is
- a list
- with
- five scalar values
# inline sequence
[this, is, a list, with, five scalar values]
```

A *mapping* is used to pair up data values with identifiers called *keys*. Mappings use a colon and space (:) for each key: value pair, or can also be specified inline as a comma-separated list within curly braces. The *key* is used as an index for retrieving data from a mapping.

```
# a mapping of items
title: War and Peace
author: Leo Tolstoy
date: 1865
# same mapping written inline
{title: War and Peace, author: Leo Tolstoy, date: 1865}
```

Keys are used to associate meta information with each node and specify the expected node type (*scalar*, *sequence* or *mapping*). See “[Greenplum MapReduce Document Schema](#)” on page 793 for the keys expected by the Greenplum MapReduce program.

The Greenplum MapReduce program processes the nodes of a document in order and uses indentation (spaces) to determine the document hierarchy and the relationships of the nodes to one another. The use of white space is significant. White space should not be used simply for formatting purposes, and tabs should not be used at all.

Greenplum MapReduce Document Schema

Greenplum MapReduce uses the YAML document framework and implements its own YAML schema. The basic structure of a Greenplum MapReduce document is:

```
%YAML 1.1
---
VERSION: 1.0.0.1
DATABASE: dbname
USER: db_username
HOST: master_hostname
PORT: master_port

DEFINE:

- INPUT:
  NAME: input_name
  FILE:
    - hostname:/path/to/file
  GPFDIST:
    - hostname:port:/file_pattern
  TABLE: table_name
  QUERY: SELECT_statement
  EXEC: command_string
  COLUMNS:
    - field_name data_type
  FORMAT: TEXT | CSV
  DELIMITER: delimiter_character
  ESCAPE: escape_character
  NULL: null_string
  QUOTE: csv_quote_character
  ERROR_LIMIT: integer
  ENCODING: database_encoding
```

```

- OUTPUT:
  NAME: output_name
  FILE: file_path_on_client
  TABLE: table_name
  KEYS:
    - column_name
  MODE: REPLACE | APPEND

- MAP:
  NAME: function_name
  FUNCTION: function_definition
  LANGUAGE: perl | python
  PARAMETERS:
    - name type
  RETURNS:
    - name type
  OPTIMIZE: STRICT IMMUTABLE
  MODE: SINGLE | MULTI

- TRANSITION | CONSOLIDATE | FINALIZE:
  NAME: function_name
  FUNCTION: function_definition
  LANGUAGE: perl | python
  PARAMETERS:
    - name type
  RETURNS:
    - name type
  OPTIMIZE: STRICT IMMUTABLE
  MODE: SINGLE | MULTI

- REDUCE:
  NAME: reduce_job_name
  TRANSITION: transition_function_name
  CONSOLIDATE: consolidate_function_name
  FINALIZE: finalize_function_name
  INITIALIZE: value
  KEYS:
    - key_name

```

```

- TASK:
  NAME: task_name
  SOURCE: input_name
  MAP: map_function_name
  REDUCE: reduce_function_name

EXECUTE:

- RUN:
  SOURCE: input_or_task_name
  TARGET: output_name
  MAP: map_function_name
  REDUCE: reduce_function_name
...

```

VERSION

Required. The version of the Greenplum MapReduce YAML specification. Current versions are 1.0.0.1.

DATABASE

Optional. Specifies which database in Greenplum to connect to. If not specified, defaults to the default database or `$PGDATABASE` if set.

USER

Optional. Specifies which database role to use to connect. If not specified, defaults to the current user or `$PGUSER` if set. You must be a Greenplum superuser to run functions written in untrusted Python and Perl. Regular database users can run functions written in trusted Perl. You also must be a database superuser to run MapReduce jobs that contain [FILE](#), [GPFDIST](#) or [EXEC](#) input types.

HOST

Optional. Specifies Greenplum master host name. If not specified, defaults to localhost or `$PGHOST` if set.

PORT

Optional. Specifies Greenplum master port. If not specified, defaults to 5432 or `$PGPORT` if set.

DEFINE

Required. A sequence of definitions for this MapReduce document. The `DEFINE` section must have at least one `INPUT` definition.

INPUT

Required. Defines the input data. Every MapReduce document must have at least one input defined. Multiple input definitions are allowed in a document, but each input definition can specify only one of these access types: a file, a `gpfdist` file distribution program, a table in the database, an SQL command, or an operating system command.

NAME

A name for this input. Names must be unique with regards to the names of other objects in this MapReduce job (such as map function, task, reduce function and output names). Also, names cannot conflict with existing objects in the database (such as tables, functions or views).

FILE

A sequence of one or more input files in the format: `seghostname:/path/to/filename`. You must be a Greenplum Database superuser to run MapReduce jobs with `FILE` input. The file must reside on a Greenplum segment host.

GPFDIST

A sequence of one or more running `gpfdist` file distribution programs in the format: `hostname[:port]/file_pattern`. You must be a Greenplum Database superuser to run MapReduce jobs with `GPFDIST` input, unless the server configuration parameter `gp_external_grant_privileges` is set to `on`.

TABLE

The name of an existing table in the database.

QUERY

An SQL `SELECT` command to run within the database.

EXEC

An operating system command to run on the Greenplum segment hosts. The command is run by all segment instances in the system by default. For example, if you have four segment instances per segment host, the command will be run four times on each host. You must be a Greenplum Database superuser to run MapReduce jobs with `EXEC` input and the server configuration parameter `gp_external_enable_exec` is set to `on`.

COLUMNS

Optional. Columns are specified as: `column_name [data_type]`. If not specified, the default is `value text`. The `DELIMITER` character is what separates two data value fields (columns). A row is determined by a line feed character (`0x0a`).

FORMAT

Optional. Specifies the format of the data - either delimited text (`TEXT`) or comma separated values (`CSV`) format. If the data format is not specified, defaults to `TEXT`.

DELIMITER

Optional for `FILE`, `GPFDIST` and `EXEC` inputs. Specifies a single character that separates data values. The default is a tab character in `TEXT` mode, a comma in `CSV` mode. The delimiter character must only appear between any two data value fields. Do not place a delimiter at the beginning or end of a row.

ESCAPE

Optional for `FILE`, `GPFDIST` and `EXEC` inputs. Specifies the single character that is used for C escape sequences (such as `\n`, `\t`, `\100`, and so on) and for escaping data characters that might otherwise be taken as row or column delimiters. Make sure to choose an escape character that is not used anywhere in your actual data values. The default escape character is a `\` (backslash), however it is possible to specify any other character to represent an escape. It is also possible to disable escaping by specifying the value `'OFF'` as the escape value. This is very useful for data such as web log data that has many embedded backslashes that are not intended to be escapes.

NULL

Optional for `FILE`, `GPFDIST` and `EXEC` inputs. Specifies the string that represents a null value. The default is `\N` in `TEXT` format, and an empty value with no quotations in `CSV` format. You might prefer an empty string even in `TEXT` mode for cases where you do not want to distinguish nulls from empty strings. Any input data item that matches this string will be considered a null value.

QUOTE

Optional for `FILE`, `GPFDIST` and `EXEC` inputs. Specifies the quotation character for `CSV` formatted files. The default is a double quote (`"`). In `CSV` formatted files, data value fields must be enclosed in double quotes if they contain any commas or embedded new lines. Fields that contain double quote characters must be surrounded by double quotes, and the embedded double quotes must each be represented by a pair of consecutive double quotes. It is important to always open and close quotes correctly in order for data rows to be parsed correctly.

ERROR_LIMIT

If the input rows have format errors they will be discarded provided that the error limit count is not reached on any Greenplum segment instance during input processing. If the error limit is not reached, all good rows will be processed and any error rows discarded.

ENCODING

Character set encoding to use for the data. Specify a string constant (such as 'SQL_ASCII'), an integer encoding number, or `DEFAULT` to use the default client encoding. See “[Character Set Support](#)” on page 52.

OUTPUT

Optional. Defines where to output the formatted data of this MapReduce job. If output is not defined, the default is `STDOUT` (standard output of the client). You can send output to a file on the client host or to an existing table in the database.

NAME

A name for this output. The default output name is `STDOUT`. Names must be unique with regards to the names of other objects in this MapReduce job (such as map function, task, reduce function and input names). Also, names cannot conflict with existing objects in the database (such as tables, functions or views).

FILE

Specifies a file location on the MapReduce client machine to output data in the format: */path/to/filename*

TABLE

Specifies the name of a table in the database to output data. If this table does not exist prior to running the MapReduce job, it will be created using the distribution policy specified with `KEYS`.

KEYS

Optional for `TABLE` output. Specifies the column(s) to use as the Greenplum Database distribution key. If the `EXECUTE` task contains a `REDUCE` definition, then the `REDUCE` keys will be used as the table distribution key by default. Otherwise, the first column of the table will be used as the distribution key.

MODE

Optional for `TABLE` output. If not specified, the default is to create the table if it does not already exist, but error out if it does exist. Declaring `APPEND` adds output data to an existing table (provided the table schema matches the output format) without removing any existing data. Declaring `REPLACE` will drop the table if it exists and then recreate it. Both `APPEND` and `REPLACE` will create a new table if one does not exist.

MAP

Required. Each `MAP` function takes data structured in (*key, value*) pairs, processes each pair, and generates zero or more output (*key, value*) pairs. The Greenplum MapReduce framework then collects all pairs with the same key from all output lists and groups them together. This output is then passed to the `REDUCE` task, which is comprised of `TRANSITION` | `CONSOLIDATE` | `FINALIZE` functions.

There is one predefined MAP function named `IDENTITY` that returns (*key, value*) pairs unchanged. Although (*key, value*) are the default parameters, you can specify other prototypes as needed.

TRANSITION | CONSOLIDATE | FINALIZE

`TRANSITION`, `CONSOLIDATE` and `FINALIZE` are all component pieces of `REDUCE`. A `TRANSITION` function is required. `CONSOLIDATE` and `FINALIZE` functions are optional. By default, all take `state` as the first of their input `PARAMETERS`, but other prototypes can be defined as well.

A `TRANSITION` function iterates through each value of a given key and accumulates values in a `state` variable. When the transition function is called on the first value of a key, the `state` is set to the value specified by `INITIALIZE` of a `REDUCE` job (or the default state value for the data type). A transition takes two arguments as input; the current state of the key reduction, and the next value, which then produces a new `state`.

If a `CONSOLIDATE` function is specified, `TRANSITION` processing is performed at the segment-level before redistributing the keys across the Greenplum interconnect for final aggregation (two-phase aggregation). Only the resulting `state` value for a given key is redistributed, resulting in lower interconnect traffic and greater parallelism. `CONSOLIDATE` is handled like a `TRANSITION`, except that instead of `(state + value) => state`, it is `(state + state) => state`.

If a `FINALIZE` function is specified, it takes the final `state` produced by `CONSOLIDATE` (if present) or `TRANSITION` and does any final processing before emitting the final result. `TRANSITION` and `CONSOLIDATE` functions cannot return a set of values. If you need a `REDUCE` job to return a set, then a `FINALIZE` is necessary to transform the final state into a set of output values.

NAME

Required. A name for the function. Names must be unique with regards to the names of other objects in this MapReduce job (such as function, task, input and output names). Also, names cannot conflict with existing objects in the database (such as tables, functions or views).

FUNCTION

Optional. Specifies the full body of the function using the specified `LANGUAGE`. If `FUNCTION` is not specified, then a built-in SQL function is used within this MapReduce script.

LANGUAGE

Required when `FUNCTION` is used. Specifies the implementation language used to interpret the function. This release has language support for `perl` and `python`. `C`, `pgsql`, `R` and `sql` support will be added in a future release.

PARAMETERS

Optional. Function input parameters. The default type is `text`.

MAP **default** - key text, value text

TRANSITION **default** - state text, value text

CONSOLIDATE **default** - state1 text, state2 text (must have exactly two input parameters of the same data type)

FINALIZE **default** - state text (single parameter only)

RETURNS

Optional. The default return type is text.

MAP **default** - key text, value text

TRANSITION **default** - state text (single return value only)

CONSOLIDATE **default** - state text (single return value only)

FINALIZE **default** - value text

OPTIMIZE

Optional optimization parameters for the function:

STRICT - function is not affected by NULL values

IMMUTABLE - function will always return the same value for a given input

MODE

Optional. Specifies the number of rows returned by the function.

MULTI - returns 0 or more rows per input record. The return value of the function must be an array of rows to return, or the function must be written as an iterator using `yield` in Python or `return_next` in Perl. MULTI is the default mode for MAP and FINALIZE functions.

SINGLE - returns exactly one row per input record. SINGLE is the only mode supported for TRANSITION and CONSOLIDATE functions. When used with MAP and FINALIZE functions, SINGLE mode can provide modest performance improvement.

REDUCE

Required. A REDUCE definition names the [TRANSITION](#) | [CONSOLIDATE](#) | [FINALIZE](#) functions that comprise the reduction of (key, value) pairs to the final result set. There are also several predefined REDUCE jobs you can execute, which all operate over a column named `value`:

IDENTITY - returns (key, value) pairs unchanged

SUM - calculates the sum of numeric data

AVG - calculates the average of numeric data

COUNT - calculates the count of input data

MIN - calculates minimum value of numeric data

MAX - calculates maximum value of numeric data

NAME

Required. The name of this `REDUCE` job. Names must be unique with regards to the names of other objects in this MapReduce job (function, task, input and output names). Also, names cannot conflict with existing objects in the database (such as tables, functions or views).

TRANSITION

Required. The name of the `TRANSITION` function.

CONSOLIDATE

Optional. The name of the `CONSOLIDATE` function.

FINALIZE

Optional. The name of the `FINALIZE` function.

INITIALIZE

Optional for `text` and `float` data types. Required for all other data types. The default value for `text` is `' '`. The default value for `float` is `0.0`. Sets the initial `state` value of the `TRANSITION` function.

KEYS

Optional. Defaults to `[key, *]`. When using a multi-column reduce it may be necessary to specify which columns are key columns and which columns are value columns. By default, any input columns that are not passed to the `TRANSITION` function are key columns, and a column named `key` is always a key column even if it is passed to the `TRANSITION` function. The special indicator `*` indicates all columns not passed to the `TRANSITION` function. If this indicator is not present in the list of keys then any unmatched columns are discarded.

TASK

Optional. A `TASK` defines a complete end-to-end `INPUT/MAP/REDUCE` stage within a Greenplum MapReduce job pipeline. It is similar to `EXECUTE` except it is not immediately executed. A task object can be called as `INPUT` to further processing stages.

NAME

Required. The name of this task. Names must be unique with regards to the names of other objects in this MapReduce job (such as map function, reduce function, input and output names). Also, names cannot conflict with existing objects in the database (such as tables, functions or views).

SOURCE

The name of an `INPUT` or another `TASK`.

MAP

Optional. The name of a [MAP](#) function. If not specified, defaults to `IDENTITY`.

REDUCE

Optional. The name of a [REDUCE](#) function. If not specified, defaults to `IDENTITY`.

EXECUTE

Required. `EXECUTE` defines the final `INPUT/MAP/REDUCE` stage within a Greenplum MapReduce job pipeline.

RUN**SOURCE**

Required. The name of an [INPUT](#) or [TASK](#).

TARGET

Optional. The name of an [OUTPUT](#). The default output is `STDOUT`.

MAP

Optional. The name of a [MAP](#) function. If not specified, defaults to `IDENTITY`.

REDUCE

Optional. The name of a [REDUCE](#) function. Defaults to `IDENTITY`.

Example Greenplum MapReduce Document

```
# This example MapReduce job processes documents and looks for keywords in them.
# It takes two database tables as input:
#   - documents (doc_id integer, url text, data text)
#   - keywords (keyword_id integer, keyword text)#
# The documents data is searched for occurrences of keywords and returns results of
# url, data and keyword (a keyword can be multiple words, such as "high performance
# computing")

%YAML 1.1
---

VERSION: 1.0.0.1

# Connect to Greenplum Database using this database and role
DATABASE: webdata
USER: jsmith

# Begin definition section
DEFINE:

  # Declare the input, which selects all columns and rows from the
  # 'documents' and 'keywords' tables.

  - INPUT:
      NAME: doc
      TABLE: documents

  - INPUT:
      NAME: kw
      TABLE: keywords

# Define the map functions to extract terms from documents and keyword
# This example simply splits on white space, but it would be possible
# to make use of a python library like nltk (the natural language toolkit)
# to perform more complex tokenization and word stemming.
```

- MAP:

```

NAME:      doc_map
LANGUAGE: python
FUNCTION: |
    i = 0          # the index of a word within the document
    terms = {}     # a hash of terms and their indexes within the document
    # Lower-case and split the text string on space
    for term in data.lower().split():
        i = i + 1  # increment i (the index)
    # Check for the term in the terms list:
    # if stem word already exists, append the i value to the array entry
    # corresponding to the term. This counts multiple occurrences of the word.
    # If stem word does not exist, add it to the dictionary with position i.
    # For example:
    #   data: "a computer is a machine that manipulates data"
    #   "a" [1, 4]
    #   "computer" [2]
    #   "machine" [3]
    #   ...
    if term in terms:
        terms[term] += ','+str(i)
    else:
        terms[term] = str(i)
    # Return multiple lines for each document. Each line consists of
    # the doc_id, a term and the positions in the data where the term appeared.
    # For example:
    #   (doc_id => 100, term => "a", [1,4]
    #   (doc_id => 100, term => "computer", [2]
    #   ...
    for term in terms:
        yield([doc_id, term, terms[term]])
OPTIMIZE: STRICT IMMUTABLE
PARAMETERS:
    - doc_id integer
    - data text
RETURNS:
    - doc_id integer
    - term text
    - positions text

```

```

# The map function for keywords is almost identical to the one for documents
# but it also counts of the number of terms in the keyword.
- MAP:
  NAME: kw_map
  LANGUAGE: python
  FUNCTION: |
    i = 0
    terms = {}

    for term in keyword.lower().split():
      i = i + 1
      if term in terms:
        terms[term] += ','+str(i)
      else:
        terms[term] = str(i)
    # output 4 values including i (the total count for term in terms):
    yield([keyword_id, i, term, terms[term]])
  OPTIMIZE: STRICT IMMUTABLE
  PARAMETERS:
    - keyword_id integer
    - keyword text
  RETURNS:
    - keyword_id integer
    - nterms integer
    - term text
    - positions text

# A TASK is an object that defines an entire INPUT/MAP/REDUCE stage
# within a Greenplum MapReduce pipeline. It is like EXECUTION, but it is
# executed only when called as input to other processing stages.
# Identify a task called 'doc_prep' which takes in the 'doc' INPUT defined earlier
# and runs the 'doc_map' MAP function which returns doc_id, term, [term_position]
- TASK:
  NAME: doc_prep
  SOURCE: doc
  MAP: doc_map

```

```

# Identify a task called 'kw_prep' which takes in the 'kw' INPUT defined earlier
# and runs the kw_map MAP function which returns kw_id, term, [term_position]
- TASK:
    NAME: kw_prep
    SOURCE: kw
    MAP: kw_map

# One advantage of Greenplum MapReduce is that MapReduce tasks can be
# used as input to SQL operations and SQL can be used to process a MapReduce task.
# This INPUT defines a SQL query that joins the output of the 'doc_prep'
# TASK to that of the 'kw_prep' TASK. Matching terms are output to the 'candidate'
# list (any keyword that shares at least one term with the document).
- INPUT:
    NAME: term_join
    QUERY: |
        SELECT doc.doc_id, kw.keyword_id, kw.term, kw.terms,
               doc.positions as doc_positions,
               kw.positions as kw_positions
        FROM doc_prep doc INNER JOIN kw_prep kw ON (doc.term = kw.term)

# In Greenplum MapReduce, a REDUCE function is comprised of one or more functions.
# A REDUCE has an initial 'state' variable defined for each grouping key. that is
# A TRANSITION function adjusts the state for every value in a key grouping.
# If present, an optional CONSOLIDATE function combines multiple
# 'state' variables. This allows the TRANSITION function to be executed locally at
# the segment-level and only redistribute the accumulated 'state' over
# the network. If present, an optional FINALIZE function can be used to perform
# final computation on a state and emit one or more rows of output from the state.
#
# This REDUCE function is called 'term_reducer' with a TRANSITION function
# called 'term_transition' and a FINALIZE function called 'term_finalizer'

- REDUCE:
    NAME: term_reducer
    TRANSITION: term_transition
    FINALIZE: term_finalizer

```

- TRANSITION:**NAME:** `term_transition`**LANGUAGE:** `python`**PARAMETERS:**

- `state` text
- `term` text
- `nterms` integer
- `doc_positions` text
- `kw_positions` text

FUNCTION: |

```

# 'state' has an initial value of '' and is a colon delimited set
# of keyword positions. keyword positions are comma delimited sets of
# integers. For example, '1,3,2:4:'
# If there is an existing state, split it into the set of keyword positions
# otherwise construct a set of 'nterms' keyword positions - all empty
if state:
    kw_split = state.split(':')
else:
    kw_split = []
    for i in range(0,nterms):
        kw_split.append('')
# 'kw_positions' is a comma delimited field of integers indicating what
# position a single term occurs within a given keyword.
# Splitting based on ',' converts the string into a python list.
# add doc_positions for the current term
for kw_p in kw_positions.split(','):
    kw_split[int(kw_p)-1] = doc_positions
# This section takes each element in the 'kw_split' array and strings
# them together placing a ':' in between each element from the array.
# For example: for the keyword "computer software computer hardware",
# the 'kw_split' array matched up to the document data of
# "in the business of computer software software engineers"
# would look like: ['5', '6,7', '5', '']
# and the outstate would look like: 5:6,7:5:
outstate = kw_split[0]
for s in kw_split[1:]:
    outstate = outstate + ':' + s
return outstate

```

```

- FINALIZE:
  NAME: term_finalizer
  LANGUAGE: python
  RETURNS:
    - count integer
  MODE: MULTI
  FUNCTION: |
    if not state:
        return 0
    kw_split = state.split(':')
    # This function does the following:
    # 1) Splits 'kw_split' on ':'
    #    for example, 1,5,7:2,8 creates '1,5,7' and '2,8'
    # 2) For each group of positions in 'kw_split', splits the set on ','
    #    to create ['1','5','7'] from Set 0: 1,5,7 and
    #    eventually ['2', '8'] from Set 1: 2,8
    # 3) Checks for empty strings
    # 4) Adjusts the split sets by subtracting the position of the set
    #    in the 'kw_split' array
    #    ['1','5','7'] - 0 from each element = ['1','5','7']
    #    ['2', '8'] - 1 from each element = ['1', '7']
    # 5) Resulting arrays after subtracting the offset in step 4 are
    #    intersected and their overlapping values kept:
    #    ['1','5','7'].intersect(['1', '7']) = [1,7]
    # 6) Determines the length of the intersection, which is the number of
    #    times that an entire keyword (with all its pieces) matches in the
    #    document data.
    previous = None
    for i in range(0,len(kw_split)):
        isplit = kw_split[i].split(',')
        if any(map(lambda(x): x == '', isplit)):
            return 0
        adjusted = set(map(lambda(x): int(x)-i, isplit))
        if (previous):
            previous = adjusted.intersection(previous)
        else:
            previous = adjusted
    # return the final count
    if previous:
        return len(previous)
    return 0

```

```
# Define the 'term_match' task which is then executed as part
# of the 'final_output' query. It takes the INPUT 'term_join' defined
# earlier and uses the REDUCE function 'term_reducer' defined earlier

- TASK:
  NAME: term_match
  SOURCE: term_join
  REDUCE: term_reducer

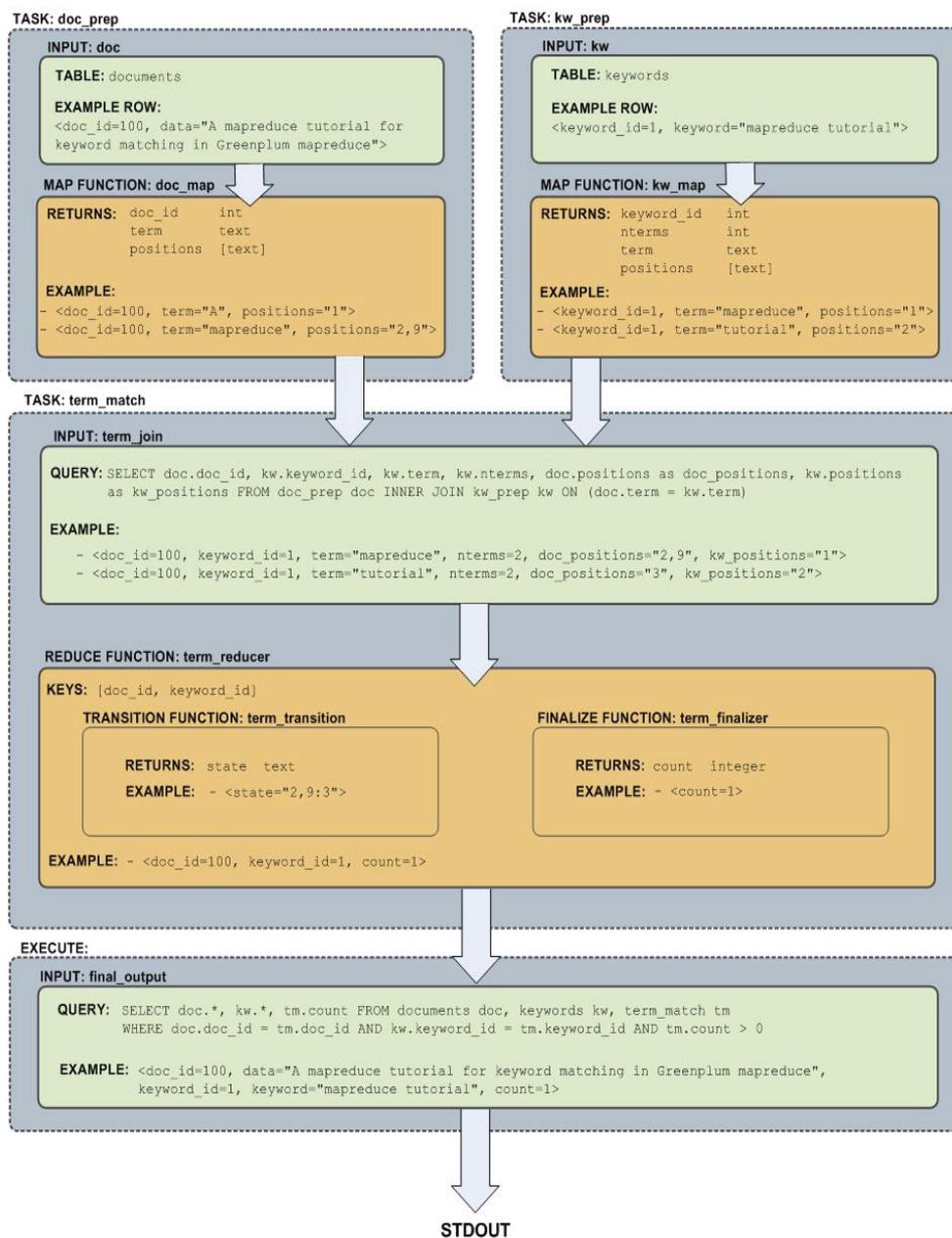
- INPUT:
  NAME: final_output
  QUERY: |
    SELECT doc.*, kw.*, tm.count
    FROM documents doc, keywords kw, term_match tm
    WHERE doc.doc_id = tm.doc_id
      AND kw.keyword_id = tm.keyword_id
      AND tm.count > 0

# Execute this MapReduce job and send output to STDOUT

EXECUTE:
- RUN:
  SOURCE: final_output
  TARGET: STDOUT
```

MapReduce Flow Diagram

The following diagram shows the job flow of the MapReduce job defined in the example:



G. Greenplum Environment Variables

This is a reference of the environment variables to set for Greenplum Database. Set these in your user's startup shell profile (such as `~/.bashrc` or `~/.bash_profile`), or in `/etc/profile` if you want to set them for all users.

Required Environment Variables

GPHOME

This is the installed location of your Greenplum Database software. For example:

```
GPHOME=greenplum-db-3.3.6.0
export GPHOME
```

PATH

Your `PATH` environment variable should point to the location of the Greenplum Database `bin` directory. Solaris users must also add `/usr/sfw/bin` and `/opt/sfw/bin` to their `PATH`. For example:

```
PATH=$GPHOME/bin:$PATH
PATH=$GPHOME/bin:/usr/local/bin:/usr/sbin:/usr/sfw/bin:/opt/sfw/bin:$PATH
export PATH
```

LD_LIBRARY_PATH

The `LD_LIBRARY_PATH` environment variable should point to the location of the Greenplum Database/PostgreSQL library files. For Solaris, this also points to the GNU compiler and readline library files as well (readline libraries may be required for Python support on Solaris). For example:

```
LD_LIBRARY_PATH=$GPHOME/lib
LD_LIBRARY_PATH=$GPHOME/lib:/usr/sfw/lib
export LD_LIBRARY_PATH
```

MASTER_DATA_DIRECTORY

The Greenplum Database management scripts need to know the location of your Greenplum master data directory. The master data directory name is created by the `gpinitssystem` script using the data storage area and naming convention you specified in your initialization configuration file (`gp_init_config`). For example:

```
MASTER_DATA_DIRECTORY=/dbfast1/master/gp-1
export MASTER_DATA_DIRECTORY
```

Optional Environment Variables

The following are standard PostgreSQL environment variables, which are also recognized in Greenplum Database. You may want to add the connection-related environment variables to your profile for convenience, so you do not have to type so many options on the command line for client connections. Note that these environment variables should be set on the Greenplum Database master host only.

PGDATABASE

The name of the default database to use when connecting.

PGHOST

The Greenplum Database master host name.

PGHOSTADDR

The numeric IP address of the master host. This can be set instead of or in addition to `PGHOST` to avoid DNS lookup overhead.

PGPASSWORD

The password used if the server demands password authentication. Use of this environment variable is not recommended for security reasons (some operating systems allow non-root users to see process environment variables via `ps`). Instead consider using the `~/.pgpass` file.

PGPASSFILE

The name of the password file to use for lookups. If not set, it defaults to `~/.pgpass`. See the section about [The Password File](#) in the PostgreSQL documentation for more information.

PGOPTIONS

Sets additional configuration parameters for the Greenplum Database master server.

PGPORT

The port number of the Greenplum Database server on the master host. The default port is 5432.

PGUSER

The Greenplum Database user name used to connect.

PGDATESTYLE

Sets the default style of date/time representation for a session. (Equivalent to `SET datestyle TO ...`)

PGTZ

Sets the default time zone for a session. (Equivalent to `SET timezone TO ...`)

PGCLIENTENCODING

Sets the default client character set encoding for a session. (Equivalent to `SET client_encoding TO ...`)

H. Greenplum Database Data Types

Greenplum Database has a rich set of native data types available to users. Users may also define new data types using the `CREATE TYPE` command. This reference shows all of the built-in data types. In addition to the types listed here, there are also some internally used data types, such as *oid* (object identifier), but those are not documented in this guide.

The following data types are specified by SQL: *bit*, *bit varying*, *boolean*, *character varying*, *varchar*, *character*, *char*, *date*, *double precision*, *integer*, *interval*, *numeric*, *decimal*, *real*, *smallint*, *time* (with or without time zone), and *timestamp* (with or without time zone).

Each data type has an external representation determined by its input and output functions. Many of the built-in types have obvious external formats. However, several types are either unique to PostgreSQL (and Greenplum Database), such as geometric paths, or have several possibilities for formats, such as the date and time types. Some of the input and output functions are not invertible. That is, the result of an output function may lose accuracy when compared to the original input.

Table H.1 Greenplum Database Built-in Data Types

Name ¹	Alias	Size	Range	Description
bigint	int8	8 bytes	-9223372036854775808 to 9223372036854775807	large range integer
bigserial	serial8	8 bytes	1 to 9223372036854775807	large autoincrementing integer
bit [(n)]		<i>n</i> bits	bit string constant	fixed-length bit string
bit varying [(n)]	varbit	actual number of bits	bit string constant	variable-length bit string
boolean	bool	1 byte	true/false, t/f, yes/no, y/n, 1/0	logical boolean (true/false)
box		32 bytes	((x1,y1),(x2,y2))	rectangular box in the plane - not allowed in distribution key columns.
bytea		1 byte + <i>binary string</i>	sequence of octets	variable-length binary string
character [(n)]	char [(n)]	1 byte + <i>n</i>	strings up to <i>n</i> characters in length	fixed-length, blank padded
character varying [(n)]	varchar [(n)]	1 byte + <i>string size</i>	strings up to <i>n</i> characters in length	variable-length with limit
cidr		12 or 24 bytes		IPv4 and IPv6 networks

Table H.1 Greenplum Database Built-in Data Types

Name ¹	Alias	Size	Range	Description
circle		24 bytes	<(x,y),r> (center and radius)	circle in the plane - not allowed in distribution key columns.
date		4 bytes	4713 BC - 5874897 AD	calendar date (year, month, day)
decimal [(p, s)]	numeric [(p, s)]	variable	no limit	user-specified precision, exact
double precision	float8 float	8 bytes	15 decimal digits precision	variable-precision, inexact
inet		12 or 24 bytes		IPv4 and IPv6 hosts and networks
integer	int, int4	4 bytes	-2147483648 to +2147483647	usual choice for integer
interval [(p)]		12 bytes	-178000000 years - 178000000 years	time span
line		32 bytes	((x1,y1),(x2,y2))	infinite line in the plane - not allowed in distribution key columns.
lseg		32 bytes	((x1,y1),(x2,y2))	line segment in the plane - not allowed in distribution key columns.
macaddr		6 bytes		MAC addresses
money		4 bytes	-21474836.48 to +21474836.47	currency amount
path		16+16n bytes	[(x1,y1),...]	geometric path in the plane - not allowed in distribution key columns.
point		16 bytes	(x,y)	geometric point in the plane - not allowed in distribution key columns.
polygon		40+16n bytes	((x1,y1),...)	closed geometric path in the plane - not allowed in distribution key columns.
real	float4	4 bytes	6 decimal digits precision	variable-precision, inexact
serial	serial4	4 bytes	1 to 2147483647	autoincrementing integer
smallint	int2	2 bytes	-32768 to +32767	small range integer
text		1 byte + <i>string size</i>	strings of any length	variable unlimited length
time [(p)] [without time zone]		8 bytes	00:00:00 - 24:00:00	time of day only
time [(p)] with time zone	timetz	12 bytes	00:00:00+1359 - 24:00:00-1359	time of day only, with time zone

Table H.1 Greenplum Database Built-in Data Types

Name¹	Alias	Size	Range	Description
timestamp [(p)] [without time zone]		8 bytes	4713 BC - 5874897 AD	both date and time
timestamp [(p)] with time zone	timestampz	8 bytes	4713 BC - 5874897 AD	both date and time, with time zone

1. For variable length data types (such as char, varchar, text, etc.) if the data is greater than or equal to 127 bytes, the storage overhead is 4 bytes instead of 1.

I. System Catalog Reference

This is a reference of the system catalog tables of Greenplum Database. All system tables prefixed with *gp_* are special system tables added by Greenplum to the standard PostgreSQL system catalogs to implement the parallel features of Greenplum Database. Tables prefixed with *pg_* are standard PostgreSQL system catalog tables, which are also used in Greenplum Database. Note that the global system catalog for Greenplum Database resides on the master instance.

System Tables

- [gp_configuration](#)
- [gp_configuration_history](#)
- [gp_db_interfaces](#)
- [gp_interfaces](#)
- [gp_distribution_policy](#)
- [gpexpand.status](#)
- [gpexpand.status_detail](#)
- [gp_id](#)
- [gp_master_mirroring](#)
- [gp_version_at_initdb](#)
- [pg_aggregate](#)
- [pg_am](#)
- [pg_amop](#)
- [pg_amproc](#)
- [pg_appendonly](#)
- [pg_attrdef](#)
- [pg_attribute](#)
- [pg_authid](#)
- [pg_auth_members](#)
- [pg_autovacuum](#)
- [pg_cast](#)
- [pg_class](#)
- [pg_constraint](#)
- [pg_conversion](#)
- [pg_database](#)
- [pg_depend](#)

- `pg_description`
- `pg_exttable`
- `pg_inherits`
- `pg_index`
- `pg_language`
- `pg_largeobject`
- `pg_listener`
- `pg_namespace`
- `pg_operator`
- `pg_opclass`
- `pg_partition`
- `pg_partition_rule`
- `pg_pltemplate`
- `pg_proc`
- `pg_resqueue`
- `pg_rewrite`
- `pg_shdepend`
- `pg_shdescription`
- `pg_statistic`
- `pg_tablespace`
- `pg_trigger`
- `pg_type`
- `pg_window`

System Views

Greenplum Database also contains the following system views currently not available in PostgreSQL.

- `gp_distributed_log`
- `gp_distributed_xacts`
- `gp_transaction_log`
- `gp_pgdatabase`
- `gpexpand.expansion_progress`
- `pg_partitions`
- `pg_partition_columns`
- `pg_partition_templates`
- `pg_resqueue_status`

- [pg_stat_resqueues](#)

For more information on the standard system views in PostgreSQL (which are also used in Greenplum Database), see the following sections of the PostgreSQL documentation:

- [System Views](#)
- [Statistics Collector Views](#)
- [The Information Schema](#)

gp_configuration

The *gp_configuration* table contains information about each host and instance (segments and masters) in a Greenplum Database system. This table is populated only on the master. This table is defined in the *pg_global* tablespace, meaning it is globally shared across all databases in the system.

Table I.1 pg_catalog.gp_configuration

column	type	references	description
content	smallint		The ID for the portion of data on an instance. A primary segment instance and its mirror will have the same content ID. For a segment the value is from 0- <i>N</i> , where <i>N</i> is the number of segments in Greenplum Database. For the master, the value is -1. The combination of <i>content</i> and <i>definedprimary</i> is the PRIMARY KEY.
definedprimary	boolean		Whether or not this instance was defined as the primary (as opposed to the mirror) at the time the system was initialized.
dbid	smallint		System-assigned ID. The unique identifier of a segment (or master) instance.
isprimary	boolean		Whether or not this instance is currently acting as the primary (as opposed to the mirror).
valid	boolean		Whether or not the instance is currently operational in the Greenplum Database system.
hostname	name		The DNS host name or IP address.
port	integer		The port number that the database server instance (segment or master) is using.
datadir	text		The data directory for the database server instance (segment or master).

gp_configuration_history

The *gp_configuration_history* table contains information about system changes related to fault detection and recovery operations. The *fts_probe* process logs data to this table, as do certain related management utilities such as *gpaddmirrors*, *gprecoverseg*, and *gpinitssystem*. For example, when you add a new segment and mirror segment to the system, records for these events are logged to *gp_configuration_history*.

The event descriptions stored in this table may be helpful for troubleshooting serious system issues in collaboration with Greenplum support technicians.

This table is populated only on the master. This table is defined in the *pg_global* tablespace, meaning it is globally shared across all databases in the system.

Table I.1 pg_catalog.gp_configuration_history

column	type	references	description
time	timestamp with time zone		Timestamp for the event recorded.
dbid	smallint	<i>gp_configuration.dbid</i>	System-assigned ID. The unique identifier of a segment (or master) instance.
desc	text		Text description of the event. For example: GPRECOVERSEG: set valid=t FTSPROBE: fault mode "RW" marking as valid=t isprimary=t persist=t

gp_db_interfaces

The *gp_db_interfaces* table contains information about the relationship of segments to network interfaces. This information, joined with data from *gp_interfaces*, is used by the system to optimize the usage of available network interfaces for various purposes, including fault detection.

Table I.1 gp_db_interfaces

column	type	references	description
dbid	smallint	<i>gp_configuration.dbid</i>	System-assigned ID. The unique identifier of a segment (or master) instance.
interfaceid	smallint	<i>gp_interfaces.interfaceid</i>	System-assigned ID for a network interface.
priority	smallint		Priority of the network interface for this segment.

gp_interfaces

The *gp_interfaces* table contains information about network interfaces on segment hosts. This information, joined with data from *gp_db_interfaces*, is used by the system to optimize the usage of available network interfaces for various purposes, including fault detection.

Table I.1 gp_interfaces

column	type	references	description
interfaceid	smallint		System-assigned ID. The unique identifier of a network interface.
address	name		Hostname address for the segment host containing the network interface. Can be a numeric IP address or a hostname.
status	smallint		Status for the network interface. A value of 0 indicates that the interface is unavailable.

gp_distributed_log

The *gp_distributed_log* view contains status information about distributed transactions and their associated local transactions. A distributed transaction is a transaction that involves modifying data on the segment instances. Greenplum's distributed transaction manager ensures that the segments stay in synch. This view allows you to see the status of distributed transactions.

Table I.1 pg_catalog.gp_distributed_log

column	type	references	description
segment_id	smallint	<i>gp_configuration.content</i>	The content id if the segment. The master is always -1 (no content).
dbid	small_int	<i>gp_configuration.dbid</i>	The unique id of the segment instance.
distributed_xid	xid		The global transaction id.
distributed_id	text		A system assigned ID for a distributed transaction.
status	text		The status of the distributed transaction (Committed or Aborted).
local_transaction	xid		The local transaction ID.

gp_distributed_xacts

The *gp_distributed_xacts* view contains information about Greenplum Database distributed transactions. A distributed transaction is a transaction that involves modifying data on the segment instances. Greenplum’s distributed transaction manager ensures that the segments stay in synch. This view allows you to see the currently active sessions and their associated distributed transactions.

Table I.1 pg_catalog.gp_distributed_xacts

column	type	references	description
distributed_xid	xid		The transaction ID used by the distributed transaction across the Greenplum Database array.
distributed_id	text		The distributed transaction identifier. It has 2 parts — a unique timestamp and the distributed transaction number.
state	text		The current state of this session with regards to distributed transactions.
gp_session_id	int		The ID number of the Greenplum Database session associated with this transaction.
xmin_distributed_snapshot	xid		The minimum distributed transaction number found among all open transactions when this transaction was started. It is used for MVCC distributed snapshot purposes.

gp_distribution_policy

The *gp_distribution_policy* table contains information about Greenplum Database tables and their policy for distributing table data across the segments. This table is populated only on the master. This table is not globally shared, meaning each database has its own copy of this table.

Table I.1 pg_catalog.gp_distribution_policy

column	type	references	description
localoid	oid	<i>pg_class.oid</i>	The table object identifier (OID).
attnums	smallint[]	<i>pg_attribute.attnum</i>	The column number(s) of the distribution column(s).

gpexpand.status

The *gpexpand.status* table contains information about the status of a system expansion operation. Status for specific tables involved in the expansion is stored in *gpexpand.status_detail*.

In a normal expansion operation it is not necessary to modify the data stored in this table. .

Table I.1 gpexpand.status

column	type	references	description
status	text		Tracks the status of an expansion operation. Valid values are: SETUP SETUP DONE EXPANSION STARTED EXPANSION STOPPED COMPLETED
updated	timestamp with time zone		Timestamp of the last change in status.

gpexpand.status_detail

The *gpexpand.status_detail* table contains information about the status of tables involved in a system expansion operation. You can query this table to determine the status of tables being expanded, or to view the start and end time for completed tables.

This table also stores related information about the table such as the oid, disk size, and normal distribution policy and key. Overall status information for the expansion is stored in *gpexpand.status*.

In a normal expansion operation it is not necessary to modify the data stored in this table. .

Table I.1 gp_expansion_status_detail

column	type	references	description
dbname	text		Name of the database to which the table belongs.
fq_name	text		Fully qualified name of the table.
schema_oid	oid		OID for the schema of the database to which the table belongs.
table_oid	oid		OID of the table.
distribution_policy	smallint()		Array of column IDs for the distribution key of the table.
distribution_policy_names	text		Column names for the hash distribution key.
distribution_policy_coloids	text		Column IDs for the distribution keys of the table.
storage_options	text		Not enabled in this release. Do not update this field.
rank	int		Rank determines the order in which tables are expanded. The expansion utility will sort on rank and expand the lowest-ranking tables first.
status	text		Status of expansion for this table. Valid values are: NOT STARTED IN PROGRESS FINISHED
last updated	timestamp with time zone		Timestamp of the last change in status for this table.
expansion started	timestamp with time zone		Timestamp for the start of the expansion of this table. This field is only populated after a table is successfully expanded.

Table I.1 gp_expansion_status_detail

column	type	references	description
expansion finished	timestamp with time zone		Timestamp for the completion of expansion of this table.
source bytes			The size of disk space associated with the source table. Due to table bloat in heap tables and differing numbers of segments after expansion, it is not expected that the final number of bytes will equal the source number. This information is tracked to help provide progress measurement to aid in duration estimation for the end-to-end expansion operation.

gpexpand.expansion_progress

The *gpexpand.expansion_progress* view contains information about the status of a system expansion operation. The view provides calculations of the estimated rate of table redistribution and estimated time to completion.

Status for specific tables involved in the expansion is stored in *gpexpand.status_detail*.

Table I.1 gpexpand.expansion_progress

column	type	references	description
name	text		Name for the data field provided Includes: Bytes Left Bytes Done Estimated Expansion Rate Estimated Time to Completion Tables Expanded Tables Left
value	text		The value for the progress data. For example: Estimated Expansion Rate - 9.75667095996092 MB/s

gp_id

The *gp_id* system catalog table identifies the Greenplum Database system name and number of segments for the system. It also has *local* values for the particular database instance (segment or master) on which the table resides. This table is defined in the *pg_global* tablespace, meaning it is globally shared across all databases in the system.

Table I.1 pg_catalog.gp_id

column	type	references	description
gpname	name		The name of this Greenplum Database system.
numsegments	integer		The number of segments in the Greenplum Database system.
dbid	integer		The unique identifier of this segment (or master) instance.
content	integer		The ID for the portion of data on this segment instance. A primary and its mirror will have the same content ID. For a segment the value is from 0- <i>N</i> , where <i>N</i> is the number of segments in Greenplum Database. For the master, the value is -1.

gp_master_mirroring

The *gp_master_mirroring* table contains state information about the standby master host and its associated write-ahead log (WAL) replication process. If this synchronization process (*gpsyncagent*) fails on the standby master, it may not always be noticeable to users of the system. This catalog is a place where Greenplum Database administrators can check to see if the standby master is current and fully synchronized.

Table I.1 pg_catalog.gp_master_mirroring

column	type	references	description
summary_state	text		The current state of the log replication process between the master and standby master - logs are either 'Synchronized' or 'Not Synchronized'
detail_state	text		If not synchronized, this column will have information about the cause of the error.
log_time	timestampz		This contains the timestamp of the last time the master sent its logs to the standby master.
error_message	text		If not synchronized, this column will have the error message from the failed synchronization attempt.

gp_transaction_log

The *gp_transaction_log* view contains status information about transactions local to a particular segment. This view allows you to see the status of local transactions.

Table I.1 pg_catalog.gp_transaction_log

column	type	references	description
segment_id	smallint	<i>gp_configuration.content</i>	The content id if the segment. The master is always -1 (no content).
dbid	smallint	<i>gp_configuration.dbid</i>	The unique id of the segment instance.
transaction	xid		The local transaction ID.
status	text		The status of the local transaction (Committed or Aborted).

gp_pgdatabase

The *gp_pgdatabase* view shows status information about the Greenplum segment instances and whether they are acting as the mirror or the primary. This view is used internally by the Greenplum fault detection and recovery utilities to determine failed segments when running in *readonly* fault operational mode, as segments are only marked invalid in *gp_configuration* when running in *continue* fault operational mode.

Table I.1 pg_catalog.gp_pgdatabase

column	type	references	description
dbid	smallint	<i>gp_configuration.dbid</i>	System-assigned ID. The unique identifier of a segment (or master) instance.
isprimary	boolean	<i>gp_configuration.isprimary</i>	Whether or not this instance is active. Is it currently acting as the primary segment (as opposed to the mirror).
content	smallint	<i>gp_configuration.content</i>	The ID for the portion of data on an instance. A primary segment instance and its mirror will have the same content ID. For a segment the value is from 0- <i>N</i> , where <i>N</i> is the number of segments in Greenplum Database. For the master, the value is -1.
definedprimary	boolean	<i>gp_configuration.definedprimary</i>	Whether or not this instance was defined as the primary (as opposed to the mirror) at the time the system was initialized.

gp_version_at_initdb

The *gp_version_at_initdb* table is populated on the master and each segment in the Greenplum Database system. It identifies the version of Greenplum Database used when the system was first initialized. This table is defined in the *pg_global* tablespace, meaning it is globally shared across all databases in the system.

Table I.1 pg_catalog.gp_version

column	type	references	description
schemaversion	integer		Schema version number.
productversion	text		Product version number.

pg_aggregate

The *pg_aggregate* table stores information about aggregate functions. An aggregate function is a function that operates on a set of values (typically one column from each row that matches a query condition) and returns a single value computed from all these values. Typical aggregate functions are *sum*, *count*, and *max*. Each entry in *pg_aggregate* is an extension of an entry in *pg_proc*. The *pg_proc* entry carries the aggregate's name, input and output data types, and other information that is similar to ordinary functions.

Table I.1 pg_catalog.pg_aggregate

column	type	references	description
aggfnoid	regproc	<i>pg_proc.oid</i>	Aggregate function OID
aggtransfn	regproc	<i>pg_proc.oid</i>	Transition function OID
aggprelimfn	regproc		Preliminary function OID (zero if none)
aggfinalfn	regproc	<i>pg_proc.oid</i>	Final function OID (zero if none)
agginvtransfn	regproc	<i>pg_proc.oid</i>	The OID in <i>pg_proc</i> of the inverse function of <i>aggtransfn</i>
agginvprelimfn	regproc	<i>pg_proc.oid</i>	The OID in <i>pg_proc</i> of the inverse function of <i>aggprelimfn</i>
aggstortop	oid	<i>pg_operator.oid</i>	Associated sort operator OID (zero if none)
aggtranstype	oid	<i>pg_type.oid</i>	Data type of the aggregate function's internal transition (state) data
agginitval	text		The initial value of the transition state. This is a text field containing the initial value in its external string representation. If this field is NULL, the transition state value starts out NULL

pg_am

The *pg_am* table stores information about index access methods. There is one row for each index access method supported by the system.

Table I.1 pg_catalog.pg_am

column	type	references	description
amname	name		Name of the access method
amstrategies	int2		Number of operator strategies for this access method
amsupport	int2		Number of support routines for this access method
amorderstrategy	int2		Zero if the index offers no sort order, otherwise the strategy number of the strategy operator that describes the sort order
amcanunique	boolean		Does the access method support unique indexes?
amcanmulticol	boolean		Does the access method support multicolumn indexes?
amoptionalkey	boolean		Does the access method support a scan without any constraint for the first index column?
amindexnulls	boolean		Does the access method support null index entries?
amstorage	boolean		Can index storage data type differ from column data type?
amclusterable	boolean		Can an index of this type be clustered on?
aminsert	regproc	<i>pg_proc.oid</i>	“Insert this tuple” function
ambeginscan	regproc	<i>pg_proc.oid</i>	“Start new scan” function
amgettupl	regproc	<i>pg_proc.oid</i>	“Next valid tuple” function
amgetmulti	regproc	<i>pg_proc.oid</i>	“Fetch multiple tuples” function
amrescan	regproc	<i>pg_proc.oid</i>	“Restart this scan” function
amendscan	regproc	<i>pg_proc.oid</i>	“End this scan” function
ammarkpos	regproc	<i>pg_proc.oid</i>	“Mark current scan position” function
amrestrpos	regproc	<i>pg_proc.oid</i>	“Restore marked scan position” function
ambuild	regproc	<i>pg_proc.oid</i>	“Build new index” function
ambulkdelete	regproc	<i>pg_proc.oid</i>	Bulk-delete function
amvacuumcleanup	regproc	<i>pg_proc.oid</i>	Post-VACUUM cleanup function

Table I.1 pg_catalog.pg_am

column	type	references	description
amcostestimate	regproc	<i>pg_proc.oid</i>	Function to estimate cost of an index scan
amoptions	regproc	<i>pg_proc.oid</i>	Function to parse and validate reloptions for an index

pg_amop

The *pg_amop* table stores information about operators associated with index access method operator classes. There is one row for each operator that is a member of an operator class.

Table I.1 pg_catalog.pg_amop

column	type	references	description
amopclaid	oid	<i>pg_opclass.oid</i>	The index operator class this entry is for
amopsubtype	oid	<i>pg_type.oid</i>	Subtype to distinguish multiple entries for one strategy; zero for default
amopstrategy	int2		Operator strategy number
amopreqcheck	boolean		Index hit must be rechecked
amopopr	oid	<i>pg_operator.oid</i>	OID of the operator

pg_amproc

The *pg_amproc* table stores information about support procedures associated with index access method operator classes. There is one row for each support procedure belonging to an operator class.

Table I.1 pg_catalog.pg_amproc

column	type	references	description
amopclaid	oid	<i>pg_opclass.oid</i>	The index operator class this entry is for
amproctype	oid	<i>pg_type.oid</i>	Subtype, if cross-type routine, else zero
amprocnum	int2		Support procedure number
amproc	regproc	<i>pg_proc.oid</i>	OID of the procedure

pg_attrdef

The *pg_attrdef* table stores column default values. The main information about columns is stored in *pg_attribute*. Only columns that explicitly specify a default value (when the table is created or the column is added) will have an entry here.

Table I.1 pg_catalog.pg_attrdef

column	type	references	description
adrelid	oid	<i>pg_class.oid</i>	The table this column belongs to
adnum	int2	<i>pg_attribute.attnum</i>	The number of the column
adbin	text		The internal representation of the column default value
adsrc	text		A human-readable representation of the default value. This field is historical, and is best not used.

pg_attribute

The *pg_attribute* table stores information about table columns. There will be exactly one *pg_attribute* row for every column in every table in the database. (There will also be attribute entries for indexes, and all objects that have *pg_class* entries.) The term attribute is equivalent to column.

Table I.1 pg_catalog.pg_attribute

column	type	references	description
attrelid	oid	<i>pg_class.oid</i>	The table this column belongs to
attname	name		The column name
atttypid	oid	<i>pg_type.oid</i>	The data type of this column
attstattarget	int4		Controls the level of detail of statistics accumulated for this column by <i>ANALYZE</i> . A zero value indicates that no statistics should be collected. A negative value says to use the system default statistics target. The exact meaning of positive values is data type-dependent. For scalar data types, it is both the target number of “most common values” to collect, and the target number of histogram bins to create.
attlen	int2		A copy of <i>pg_type.typelen</i> of this column’s type.
attnum	int2		The number of the column. Ordinary columns are numbered from 1 up. System columns, such as <i>oid</i> , have (arbitrary) negative numbers.
atndims	int4		Number of dimensions, if the column is an array type; otherwise 0. (Presently, the number of dimensions of an array is not enforced, so any nonzero value effectively means it is an array)
attcacheoff	int4		Always -1 in storage, but when loaded into a row descriptor in memory this may be updated to cache the offset of the attribute within the row
atttypmod	int4		Records type-specific data supplied at table creation time (for example, the maximum length of a <i>varchar</i> column). It is passed to type-specific input functions and length coercion functions. The value will generally be -1 for types that do not need it.
attbyval	boolean		A copy of <i>pg_type.typbyval</i> of this column’s type

Table I.1 pg_catalog.pg_attribute

column	type	references	description
attstorage	char		Normally a copy of <i>pg_type.typstorage</i> of this column's type. For TOAST-able data types, this can be altered after column creation to control storage policy.
attalign	char		A copy of <i>pg_type.typalign</i> of this column's type
attnotnull	boolean		This represents a not-null constraint. It is possible to change this column to enable or disable the constraint.
atthasdef	boolean		This column has a default value, in which case there will be a corresponding entry in the <i>pg_attrdef</i> catalog that actually defines the value
attisdropped	boolean		This column has been dropped and is no longer valid. A dropped column is still physically present in the table, but is ignored by the parser and so cannot be accessed via SQL
attislocal	boolean		This column is defined locally in the relation. Note that a column may be locally defined and inherited simultaneously
attinhcount	int4		The number of direct ancestors this column has. A column with a nonzero number of ancestors cannot be dropped nor renamed

pg_auth_members

The *pg_auth_members* system catalog table shows the membership relations between roles. Any non-circular set of relationships is allowed. Because roles are system-wide, *pg_auth_members* is shared across all databases of a Greenplum Database system.

Table I.1 pg_catalog.pg_auth_members

column	type	references	description
roleid	oid	<i>pg_authid.oid</i>	ID of the parent-level (group) role
member	oid	<i>pg_authid.oid</i>	ID of a member role
grantor	oid	<i>pg_authid.oid</i>	ID of the role that granted this membership
admin_option	boolean		True if role member may grant membership to others

pg_authid

The *pg_authid* table contains information about database authorization identifiers (roles). A role subsumes the concepts of users and groups. A user is a role with the *rolcanlogin* flag set. Any role (with or without *rolcanlogin*) may have other roles as members. See *pg_auth_members*.

Since this catalog contains passwords, it must not be publicly readable. *pg_roles* is a publicly readable view on *pg_authid* that blanks out the password field.

Because user identities are system-wide, *pg_authid* is shared across all databases in a Greenplum Database system: there is only one copy of *pg_authid* per system, not one per database.

Table I.1 pg_catalog.pg_authid

column	type	references	description
rolname	name		Role name
rolsuper	boolean		Role has superuser privileges
rolinherit	boolean		Role automatically inherits privileges of roles it is a member of
rolcreaterole	boolean		Role may create more roles
rolcreatedb	boolean		Role may create databases
rolcatupdate	boolean		Role may update system catalogs directly. (Even a superuser may not do this unless this column is true)
rolcanlogin	boolean		Role may log in. That is, this role can be given as the initial session authorization identifier
rolconnlimit	int4		For roles that can log in, this sets maximum number of concurrent connections this role can make. -1 means no limit
rolpassword	text		Password (possibly encrypted); NULL if none
rolvaliduntil	timestamptz		Password expiry time (only used for password authentication); NULL if no expiration
rolconfig	text[]		Session defaults for server configuration parameters

pg_autovacuum

The *pg_autovacuum* system catalog table stores optional per-relation configuration parameters for the autovacuum daemon. If there is an entry here for a particular relation, the given parameters will be used for autovacuuming that table. If no entry is present, the system-wide defaults will be used.

The autovacuum daemon will initiate a `VACUUM` operation on a particular table when the number of updated or deleted tuples exceeds *vac_base_thresh* plus *vac_scale_factor* times the number of live tuples currently estimated to be in the relation. Similarly, it will initiate an `ANALYZE` operation when the number of inserted, updated or deleted tuples exceeds *anl_base_thresh* plus *anl_scale_factor* times the number of live tuples currently estimated to be in the relation.

Also, the autovacuum daemon will perform a `VACUUM` operation to prevent transaction ID wraparound if the table's *pg_class.relfrozensid* field attains an age of more than *freeze_max_age* transactions, whether the table has been changed or not. The system will launch autovacuum to perform such `VACUUMS` even if autovacuum is otherwise disabled.

Any of the numerical fields can contain `-1` to indicate that the system-wide default should be used for this particular value. Observe that the *vac_cost_delay* variable inherits its default value from the *autovacuum_vacuum_cost_delay* configuration parameter, or from *vacuum_cost_delay* if the former is set to a negative value. The same applies to *vac_cost_limit*. Also, autovacuum will ignore attempts to set a per-table *freeze_max_age* larger than the system-wide setting (it can only be set smaller), and the *freeze_min_age* value will be limited to half the system-wide *autovacuum_freeze_max_age* setting.

Table I.1 pg_catalog.pg_autovacuum

column	type	references	description
vacrelid	oid	<i>pg_class.oid</i>	The table this entry is for
enabled	boolean		If false, this table is never autovacuumed
vac_base_thresh	integer		Minimum number of modified tuples before vacuum
vac_scale_factor	float4		Multiplier for <i>reltuples</i> to add to <i>vac_base_thresh</i>
anl_base_thresh	integer		Minimum number of modified tuples before analyze
anl_scale_factor	float4		Multiplier for <i>reltuples</i> to add to <i>anl_base_thresh</i>
vac_cost_delay	integer		Custom <i>vacuum_cost_delay</i> parameter
vac_cost_limit	integer		Custom <i>vacuum_cost_limit</i> parameter
freeze_min_age	integer		Custom <i>vacuum_freeze_min_age</i> parameter
freeze_max_age	integer		Custom <i>autovacuum_freeze_max_age</i> parameter

pg_cast

The catalog *pg_cast* stores data type conversion paths, both built-in paths and those defined with `CREATE CAST`. The cast functions listed in *pg_cast* must always take the cast source type as their first argument type, and return the cast destination type as their result type. A cast function can have up to three arguments. The second argument, if present, must be type `integer`; it receives the type modifier associated with the destination type, or `-1` if there is none. The third argument, if present, must be type `boolean`; it receives `true` if the cast is an explicit cast, `false` otherwise.

It is legitimate to create a *pg_cast* entry in which the source and target types are the same, if the associated function takes more than one argument. Such entries represent ‘length coercion functions’ that coerce values of the type to be legal for a particular type modifier value. Note however that at present there is no support for associating non-default type modifiers with user-created data types, and so this facility is only of use for the small number of built-in types that have type modifier syntax built into the grammar.

When a *pg_cast* entry has different source and target types and a function that takes more than one argument, it represents converting from one type to another and applying a length coercion in a single step. When no such entry is available, coercion to a type that uses a type modifier involves two steps, one to convert between data types and a second to apply the modifier.

Table I.1 `pg_catalog.pg_cast`

column	type	references	description
<code>castsource</code>	<code>oid</code>	<i>pg_type.oid</i>	OID of the source data type.
<code>casttarget</code>	<code>oid</code>	<i>pg_type.oid</i>	OID of the target data type.
<code>castfunc</code>	<code>oid</code>	<i>pg_proc.oid</i>	The OID of the function to use to perform this cast. Zero is stored if the data types are binary compatible (that is, no run-time operation is needed to perform the cast).
<code>castcontext</code>	<code>char</code>		Indicates what contexts the cast may be invoked in. <code>e</code> means only as an explicit cast (using <code>CAST</code> or <code>::</code> syntax). <code>a</code> means implicitly in assignment to a target column, as well as explicitly. <code>i</code> means implicitly in expressions, as well as the other cases.

pg_class

The system catalog table *pg_class* catalogs tables and most everything else that has columns or is otherwise similar to a table (also known as *relations*). This includes indexes (see also *pg_index*), sequences, views, composite types, and TOAST tables. Not all columns are meaningful for all relation types.

Table I.1 pg_catalog.pg_class

column	type	references	description
relname	name		Name of the table, index, view, etc.
relnamespace	oid	<i>pg_namespace.oid</i>	The OID of the namespace (schema) that contains this relation
reltype	oid	<i>pg_type.oid</i>	The OID of the data type that corresponds to this table's row type, if any (zero for indexes, which have no <i>pg_type</i> entry)
relowner	oid	<i>pg_authid.oid</i>	Owner of the relation
relam	oid	<i>pg_am.oid</i>	If this is an index, the access method used (B-tree, Bitmap, hash, etc.)
relfilenode	oid		Name of the on-disk file of this relation; 0 if none.
reltablespace	oid	<i>pg_tablespace.oid</i>	The tablespace in which this relation is stored. If zero, the database's default tablespace is implied. (Not meaningful if the relation has no on-disk file.)
relpages	int4		Size of the on-disk representation of this table in pages (of 32K each). This is only an estimate used by the planner. It is updated by <i>VACUUM</i> , <i>ANALYZE</i> , and a few DDL commands.
reltuples	float4		Number of rows in the table. This is only an estimate used by the planner. It is updated by <i>VACUUM</i> , <i>ANALYZE</i> , and a few DDL commands.
reltoastrelid	oid	<i>pg_class.oid</i>	OID of the TOAST table associated with this table, 0 if none. The TOAST table stores large attributes "out of line" in a secondary table.
reltoastidxid	oid	<i>pg_class.oid</i>	For a TOAST table, the OID of its index. 0 if not a TOAST table.
relaosegidxid	oid		Transaction ID associated with an internal append-only segment file
relaosegreid	oid		Relation OID of an internal append-only segment files and EOF

Table I.1 pg_catalog.pg_class

column	type	references	description
relhasindex	boolean		True if this is a table and it has (or recently had) any indexes. This is set by <code>CREATE INDEX</code> , but not cleared immediately by <code>DROP INDEX</code> . <code>VACUUM</code> will clear if it finds the table has no indexes.
relisshared	boolean		True if this table is shared across all databases in the system. Only certain system catalog tables are shared.
relkind	char		The type of object <i>r</i> = heap or append-only table, <i>x</i> = external table, <i>i</i> = index, <i>s</i> = sequence, <i>v</i> = view, <i>c</i> = composite type, <i>t</i> = TOAST value, <i>o</i> = internal append-only segment files and EOFs, <i>u</i> = uncataloged temporary heap table
relstorage	char		The storage mode of a table <i>a</i> = append-only, <i>h</i> = heap, <i>v</i> = virtual.
relnatts	int2		Number of user columns in the relation (system columns not counted). There must be this many corresponding entries in <i>pg_attribute</i> .
relchecks	int2		Number of check constraints on the table.
reltriggers	int2		Number of triggers on the table.
relukeys	int2		Unused
relfkeys	int2		Unused
relrefs	int2		Unused
relhasoids	boolean		True if an OID is generated for each row of the relation.
relhaspkey	boolean		True if the table has (or once had) a primary key.
relhasrules	boolean		True if table has rules.
relhassubclass	boolean		True if table has (or once had) any inheritance children.

Table I.1 pg_catalog.pg_class

column	type	references	description
relfrozenxid	xid		All transaction IDs before this one have been replaced with a permanent (frozen) transaction ID in this table. This is used to track whether the table needs to be vacuumed in order to prevent transaction ID wraparound or to allow <i>pg_clog</i> to be shrunk. Zero (<code>InvalidTransactionId</code>) if the relation is not a table.
relacl	aclitem[]		Access privileges assigned by <code>GRANT</code> and <code>REVOKE</code> .
reloptions	text[]		Access-method-specific options, as “keyword=value” strings.

pg_constraint

The *pg_constraint* system catalog table stores check, primary key, unique, and foreign key constraints on tables. Column constraints are not treated specially. Every column constraint is equivalent to some table constraint. Not-null constraints are represented in the *pg_attribute* catalog. Check constraints on domains are stored here, too.

Table I.1 pg_catalog.pg_constraint

column	type	references	description
conname	name		Constraint name (not necessarily unique!)
connamespace	oid	<i>pg_namespace.oid</i>	The OID of the namespace (schema) that contains this constraint.
contype	char		c = check constraint, f = foreign key constraint, p = primary key constraint, u = unique constraint.
condeferrable	boolean		Is the constraint deferrable?
condeferred	boolean		Is the constraint deferred by default?
conrelid	oid	<i>pg_class.oid</i>	The table this constraint is on; 0 if not a table constraint.
contypid	oid	<i>pg_type.oid</i>	The domain this constraint is on; 0 if not a domain constraint.
confrelid	oid	<i>pg_class.oid</i>	If a foreign key, the referenced table; else 0.
confupdtype	char		Foreign key update action code.
confdeltype	char		Foreign key deletion action code.
confmatchtype	char		Foreign key match type.
conkey	int2[]	<i>pg_attribute.attnum</i>	If a table constraint, list of columns which the constraint constrains.
confkey	int2[]	<i>pg_attribute.attnum</i>	If a foreign key, list of the referenced columns.
conbin	text		If a check constraint, an internal representation of the expression.
consrc	text		If a check constraint, a human-readable representation of the expression. This is not updated when referenced objects change; for example, it won't track renaming of columns. Rather than relying on this field, it is best to use <code>pg_get_constraintdef()</code> to extract the definition of a check constraint.

pg_conversion

The *pg_conversion* system catalog table describes the available encoding conversion procedures as defined by `CREATE CONVERSION`.

Table I.1 pg_catalog.pg_conversion

column	type	references	description
conname	name		Conversion name (unique within a namespace).
connamespace	oid	<i>pg_namespace.oid</i>	The OID of the namespace (schema) that contains this conversion.
conowner	oid	<i>pg_authid.oid</i>	Owner of the conversion.
conforencoding	int4		Source encoding ID.
contoencoding	int4		Destination encoding ID.
conproc	regproc	<i>pg_proc.oid</i>	Conversion procedure.
condefault	boolean		True if this is the default conversion.

pg_database

The *pg_database* system catalog table stores information about the available databases. Databases are created with the `CREATE DATABASE` SQL command. Unlike most system catalogs, *pg_database* is shared across all databases in the system. There is only one copy of *pg_database* per system, not one per database.

Table I.1 pg_catalog.pg_database

column	type	references	description
datname	name		Database name.
datdba	oid	<i>pg_authid.oid</i>	Owner of the database, usually the user who created it.
encoding	int4		Character encoding for this database. <i>pg_encoding_to_char()</i> can translate this number to the encoding name.
datistemplate	boolean		If true then this database can be used in the <code>TEMPLATE</code> clause of <code>CREATE DATABASE</code> to create a new database as a clone of this one.
datallowconn	boolean		If false then no one can connect to this database. This is used to protect the <code>template0</code> database from being altered.
datconnlimit	int4		Sets the maximum number of concurrent connections that can be made to this database. -1 means no limit.
datlastsysoid	oid		Last system OID in the database; useful particularly to <i>pg_dump/pg_dump</i> .
datfrozenxid	xid		All transaction IDs before this one have been replaced with a permanent (frozen) transaction ID in this database. This is used to track whether the database needs to be vacuumed in order to prevent transaction ID wraparound or to allow <i>pg_clog</i> to be shrunk. It is the minimum of the per-table <i>pg_class.relfrozenxid</i> values.
dattablespace	oid	<i>pg_tablespace.oid</i>	The default tablespace for the database. Within this database, all tables for which <i>pg_class.reltablespace</i> is zero will be stored in this tablespace. All non-shared system catalogs will also be there.

Table I.1 pg_catalog.pg_database

column	type	references	description
datconfig	text[]		Session defaults for user-settable server configuration parameters.
datacl	aclitem[]		Database access privileges as given by GRANT and REVOKE.

pg_depend

The *pg_depend* system catalog table records the dependency relationships between database objects. This information allows `DROP` commands to find which other objects must be dropped by `DROP CASCADE` or prevent dropping in the `DROP RESTRICT` case. See also *pg_shdepend*, which performs a similar function for dependencies involving objects that are shared across a Greenplum system.

In all cases, a *pg_depend* entry indicates that the referenced object may not be dropped without also dropping the dependent object. However, there are several subflavors identified by *deptype*:

- **DEPENDENCY_NORMAL (n)** — A normal relationship between separately-created objects. The dependent object may be dropped without affecting the referenced object. The referenced object may only be dropped by specifying `CASCADE`, in which case the dependent object is dropped, too. Example: a table column has a normal dependency on its data type.
- **DEPENDENCY_AUTO (a)** — The dependent object can be dropped separately from the referenced object, and should be automatically dropped (regardless of `RESTRICT` or `CASCADE` mode) if the referenced object is dropped. Example: a named constraint on a table is made autodependent on the table, so that it will go away if the table is dropped.
- **DEPENDENCY_INTERNAL (i)** — The dependent object was created as part of creation of the referenced object, and is really just a part of its internal implementation. A `DROP` of the dependent object will be disallowed outright (we'll tell the user to issue a `DROP` against the referenced object, instead). A `DROP` of the referenced object will be propagated through to drop the dependent object whether `CASCADE` is specified or not. Example: a trigger that's created to enforce a foreign-key constraint is made internally dependent on the constraint's *pg_constraint* entry.
- **DEPENDENCY_PIN (p)** — There is no dependent object; this type of entry is a signal that the system itself depends on the referenced object, and so that object must never be deleted. Entries of this type are created only by system initialization. The columns for the dependent object contain zeroes.

Table I.1 pg_catalog.pg_depend

column	type	references	description
classid	oid	<i>pg_class.oid</i>	The OID of the system catalog the dependent object is in.
objid	oid	any OID column	The OID of the specific dependent object.
objsubid	int4		For a table column, this is the column number. For all other object types, this column is zero.
refclassid	oid	<i>pg_class.oid</i>	The OID of the system catalog the referenced object is in.
refobjid	oid	any OID column	The OID of the specific referenced object.
refobjsubid	int4		For a table column, this is the referenced column number. For all other object types, this column is zero.
deptype	char		A code defining the specific semantics of this dependency relationship.

pg_description

The *pg_description* system catalog table stores optional descriptions (comments) for each database object. Descriptions can be manipulated with the `COMMENT` command and viewed with `psql`'s `\d` meta-commands. Descriptions of many built-in system objects are provided in the initial contents of *pg_description*. See also *pg_shdescription*, which performs a similar function for descriptions involving objects that are shared across a Greenplum system.

Table I.1 `pg_catalog.pg_description`

column	type	references	description
objoid	oid	any OID column	The OID of the object this description pertains to.
classoid	oid	<i>pg_class.oid</i>	The OID of the system catalog this object appears in
objsubid	int4		For a comment on a table column, this is the column number. For all other object types, this column is zero.
description	text		Arbitrary text that serves as the description of this object.

pg_exttable

The *pg_exttable* system catalog table is used to track external tables and web tables created by the `CREATE EXTERNAL TABLE` command.

Table I.1 pg_catalog.pg_exttable

column	type	references	description
reloid	oid	<i>pg_class.oid</i>	The OID of this external table.
location	text[]		The URI location(s) of the external table files.
command	text		The OS command to execute when the external table is accessed.
fmttype	char		Format of the external table files: <code>t</code> for text, or <code>c</code> for csv.
fmtopts	text		Formatting options of the external table files, such as the field delimiter, null string, escape character, etc.
rejectlimit	integer		The per segment reject limit for rows with errors, after which the load will fail.
rejectlimitype	char		Type of reject limit threshold: <code>r</code> for number of rows.
fmterrtbl	oid	<i>pg_class.oid</i>	The object id of the error table where format errors will be logged.
encoding	text		The client encoding.

pg_index

The *pg_index* system catalog table contains part of the information about indexes. The rest is mostly in *pg_class*.

Table I.1 pg_catalog.pg_index

column	type	references	description
indexrelid	oid	<i>pg_class.oid</i>	The OID of the <i>pg_class</i> entry for this index.
indrelid	oid	<i>pg_class.oid</i>	The OID of the <i>pg_class</i> entry for the table this index is for.
indnatts	int2		The number of columns in the index (duplicates <i>pg_class.relnatts</i>).
indisunique	boolean		If true, this is a unique index.
indisprimary	boolean		If true, this index represents the primary key of the table. (<i>indisunique</i> should always be true when this is true.)
indisclustered	boolean		If true, the table was last clustered on this index via the <code>CLUSTER</code> command.
indisvalid	boolean		If true, the index is currently valid for queries. False means the index is possibly incomplete: it must still be modified by <code>INSERT/UPDATE</code> operations, but it cannot safely be used for queries.
indkey	int2vector	<i>pg_attribute.attnum</i>	This is an array of <i>indnatts</i> values that indicate which table columns this index indexes. For example a value of 1 3 would mean that the first and the third table columns make up the index key. A zero in this array indicates that the corresponding index attribute is an expression over the table columns, rather than a simple column reference.
indclass	oidvector	<i>pg_opclass.oid</i>	For each column in the index key this contains the OID of the operator class to use.

Table I.1 pg_catalog.pg_index

column	type	references	description
indexprs	text		Expression trees (in <code>nodeToString()</code> representation) for index attributes that are not simple column references. This is a list with one element for each zero entry in <code>indkey</code> . NULL if all index attributes are simple references.
indpred	text		Expression tree (in <code>nodeToString()</code> representation) for partial index predicate. NULL if not a partial index.

pg_inherits

The *pg_inherits* system catalog table records information about table inheritance hierarchies. There is one entry for each direct child table in the database. (Indirect inheritance can be determined by following chains of entries.) In Greenplum Database, inheritance relationships are created by both the `INHERITS` clause (standalone inheritance) and the `PARTITION BY` clause (partitioned child table inheritance) of `CREATE TABLE`.

Table I.1 pg_catalog.pg_inherits

column	type	references	description
inhrelid	oid	<i>pg_class.oid</i>	The OID of the child table.
inhparent	oid	<i>pg_class.oid</i>	The OID of the parent table.
inhseqno	int4		If there is more than one direct parent for a child table (multiple inheritance), this number tells the order in which the inherited columns are to be arranged. The count starts at 1.

pg_language

The *pg_language* system catalog table registers languages in which you can write functions or stored procedures. It is populated by `CREATE LANGUAGE`.

Table I.1 pg_catalog.pg_language

column	type	references	description
lanname	name		Name of the language.
lanispl	boolean		This is false for internal languages (such as SQL) and true for user-defined languages. Currently, <code>pg_dump</code> still uses this to determine which languages need to be dumped, but this may be replaced by a different mechanism in the future.
lanpltrusted	boolean		True if this is a trusted language, which means that it is believed not to grant access to anything outside the normal SQL execution environment. Only superusers may create functions in untrusted languages.
lanplcallfoid	oid	<i>pg_proc.oid</i>	For noninternal languages this references the language handler, which is a special function that is responsible for executing all functions that are written in the particular language.
lanvalidator	oid	<i>pg_proc.oid</i>	This references a language validator function that is responsible for checking the syntax and validity of new functions when they are created. Zero if no validator is provided.
lanacl	aclitem[]		Access privileges for the language.

pg_largeobject

The *pg_largeobject* system catalog table holds the data making up ‘large objects’. A large object is identified by an OID assigned when it is created. Each large object is broken into segments or ‘pages’ small enough to be conveniently stored as rows in *pg_largeobject*. The amount of data per page is defined to be `LOBLKSIZE` (which is currently `BLCKSZ/4`, or typically 8K).

Each row of *pg_largeobject* holds data for one page of a large object, beginning at byte offset (*pageno* * `LOBLKSIZE`) within the object. The implementation allows sparse storage: pages may be missing, and may be shorter than `LOBLKSIZE` bytes even if they are not the last page of the object. Missing regions within a large object read as zeroes.

Table I.1 pg_catalog.pg_largeobject

column	type	references	description
loid	oid		Identifier of the large object that includes this page.
pageno	int4		Page number of this page within its large object (counting from zero).
data	bytea		Actual data stored in the large object. This will never be more than <code>LOBLKSIZE</code> bytes and may be less.

pg_listener

The *pg_listener* system catalog table supports the `LISTEN` and `NOTIFY` commands. A listener creates an entry in *pg_listener* for each notification name it is listening for. A notifier scans and updates each matching entry to show that a notification has occurred. The notifier also sends a signal (using the PID recorded in the table) to awaken the listener from sleep.

This table is not currently used in Greenplum Database.

Table I.1 pg_catalog.pg_listener

column	type	references	description
relname	name		Notify condition name. (The name need not match any actual relation in the database.)
listenerpid	int4		PID of the server process that created this entry.
notification	int4		Zero if no event is pending for this listener. If an event is pending, the PID of the server process that sent the notification.

pg_locks

The view *pg_locks* provides access to information about the locks held by open transactions within Greenplum Database.

pg_locks contains one row per active lockable object, requested lock mode, and relevant transaction. Thus, the same lockable object may appear many times, if multiple transactions are holding or waiting for locks on it. However, an object that currently has no locks on it will not appear at all.

There are several distinct types of lockable objects: whole relations (such as tables), individual pages of relations, individual tuples of relations, transaction IDs, and general database objects. Also, the right to extend a relation is represented as a separate lockable object.

Table I.1 pg_catalog.pg_locks

column	type	references	description
locktype	text		Type of the lockable object: relation, extend, page, tuple, transactionid, object, userlock, resource queue, or advisory
database	oid	<i>pg_database.oid</i>	OID of the database in which the object exists, zero if the object is a shared object, or NULL if the object is a transaction ID
relation	oid	<i>pg_class.oid</i>	OID of the relation, or NULL if the object is not a relation or part of a relation
page	integer		Page number within the relation, or NULL if the object is not a tuple or relation page
tuple	smallint		Tuple number within the page, or NULL if the object is not a tuple
transactionid	xid		ID of a transaction, or NULL if the object is not a transaction ID
classid	oid	<i>pg_class.oid</i>	OID of the system catalog containing the object, or NULL if the object is not a general database object
objid	oid	any OID column	OID of the object within its system catalog, or NULL if the object is not a general database object
objsubid	smallint		For a table column, this is the column number (the <i>classid</i> and <i>objid</i> refer to the table itself). For all other object types, this column is zero. NULL if the object is not a general database object
transaction	xid		ID of the transaction that is holding or awaiting this lock

Table I.1 pg_catalog.pg_locks

column	type	references	description
pid	integer		Process ID of the server process holding or awaiting this lock. NULL if the lock is held by a prepared transaction
mode	text		Name of the lock mode held or desired by this process
granted	boolean		True if lock is held, false if lock is awaited

pg_namespace

The *pg_namespace* system catalog table stores namespaces. A namespace is the structure underlying SQL schemas: each namespace can have a separate collection of relations, types, etc. without name conflicts.

Table I.1 pg_catalog.pg_namespace

column	type	references	description
nspname	name		Name of the namespace
nspowner	oid	<i>pg_authid.oid</i>	Owner of the namespace
nspacl	aclitem[]		Access privileges as given by GRANT and REVOKE.

pg_opclass

The *pg_opclass* system catalog table defines index access method operator classes. Each operator class defines semantics for index columns of a particular data type and a particular index access method. Note that there can be multiple operator classes for a given data type/access method combination, thus supporting multiple behaviors. The majority of the information defining an operator class is actually not in its *pg_opclass* row, but in the associated rows in *pg_amop* and *pg_amproc*. Those rows are considered to be part of the operator class definition — this is not unlike the way that a relation is defined by a single *pg_class* row plus associated rows in *pg_attribute* and other tables.

Table I.1 pg_catalog.pg_opclass

column	type	references	description
opcamid	oid	<i>pg_am.oid</i>	Index access method operator class is for.
opcname	name		Name of this operator class
opcnamespace	oid	<i>pg_namespace.oid</i>	Namespace of this operator class
opcowner	oid	<i>pg_authid.oid</i>	Owner of the operator class
opcintype	oid	<i>pg_type.oid</i>	Data type that the operator class indexes.
opcdefault	boolean		True if this operator class is the default for the data type <i>opcintype</i> .
opckeytype	oid	<i>pg_type.oid</i>	Type of data stored in index, or zero if same as <i>opcintype</i> .

pg_operator

The *pg_operator* system catalog table stores information about operators, both built-in and those defined by `CREATE OPERATOR`. Unused column contain zeroes. For example, *oprleft* is zero for a prefix operator.

Table I.1 pg_catalog.pg_operator

column	type	references	description
oprname	name		Name of the operator.
oprnamespace	oid	<i>pg_namespace.oid</i>	The OID of the namespace that contains this operator.
oprowner	oid	<i>pg_authid.oid</i>	Owner of the operator.
oprkind	char		b = infix (both), l = prefix (left), r = postfix (right)
oprcahash	boolean		This operator supports hash joins.
oprleft	oid	<i>pg_type.oid</i>	Type of the left operand.
oprright	oid	<i>pg_type.oid</i>	Type of the right operand.
oprresult	oid	<i>pg_type.oid</i>	Type of the result.
oprcom	oid	<i>pg_operator.oid</i>	Commutator of this operator, if any.
oprnegate		<i>pg_operator.oid</i>	Negator of this operator, if any.
oprlsortop	oid	<i>pg_operator.oid</i>	If this operator supports merge joins, the operator that sorts the type of the left-hand operand (L<L).
oprrsortop l	oid	<i>pg_operator.oid</i>	If this operator supports merge joins, the operator that sorts the type of the right-hand operand (R<R).
oprltcmpop	oid	<i>pg_operator.oid</i>	If this operator supports merge joins, the less-than operator that compares the left and right operand types (L<R).
oprgtcmpop	oid	<i>pg_operator.oid</i>	If this operator supports merge joins, the greater-than operator that compares the left and right operand types (L>R).
oprcode	regproc	<i>pg_proc.oid</i>	Function that implements this operator.
oprrest	regproc	<i>pg_proc.oid</i>	Restriction selectivity estimation function for this operator.
oprjoin	regproc	<i>pg_proc.oid</i>	Join selectivity estimation function for this operator.

pg_partition

The *pg_partition* system catalog table is used to track partitioned tables and their inheritance relationships.

Table I.1 pg_catalog.pg_partition

column	type	references	description
parrelid	oid	<i>pg_class.oid</i>	The object identifier of the table.
parkind	char		The partition type - R for range or L for list.
parlevel	smallint		The level of the partition - 1 for first level under parent table, 2 for second level, and so on.
paristemplate	boolean		Whether or not this table is based on a partition template definition.
parnatts	smallint	<i>pg_attribute.oid</i>	The number of attributes that define this level.
paratts	smallint()		An array of the attribute numbers (as in <i>pg_attribute.attnum</i>) of the attributes that participate in defining this level.
parclass	oidvector	<i>pg_opclass.oid</i>	The operator class identifier(s) of the partition columns.

pg_partitions

The `pg_partitions` system view is used to show the structure of a partitioned table.

Table I.1 pg_catalog.pg_partitions

column	type	references	description
schemaname	name		The name of the schema the partitioned table is in.
tablename	name		The name of the top-level parent table.
partitiontablename	name		The relation name of the partitioned table (this is the table name to use if accessing the partition directly).
partitionname	name		The of the partition (this is the name to use if referring to the partition in an <code>ALTER TABLE</code> command). <code>NULL</code> if the partition was not given a name at create time or generated by an <code>EVERY</code> clause.
parentpartitiontablename	name		The relation name of the parent table one level up from this partition.
parentpartitionname	name		The given name of the parent table one level up from this partition.
partitiontype	text		The type of partition (range or list).
partitionlevel	smallint		The level of this partition in the hierarchy.
partitionrank	bigint		For range partitions, the rank of the partition compared to other partitions of the same level.
partitionposition	smallint		The rule order position of this partition.
partitionlistvalues	text		For list partitions, the list value(s) associated with this partition.
partitionrangestart	text		For range partitions, the start value of this partition.
partitionstartinclusive	boolean		<code>T</code> if the start value is included in this partition. <code>F</code> if it is excluded.
partitionrangeend	text		For range partitions, the end value of this partition.
partitionendinclusive	boolean		<code>T</code> if the end value is included in this partition. <code>F</code> if it is excluded.
partitioneveryclause	text		The <code>EVERY</code> clause (interval) of this partition.

Table I.1 pg_catalog.pg_partitions

column	type	references	description
partitionisdefault	boolean		T if this is a default partition, otherwise F.
partitionboundary	text		The entire partition specification for this partition.

pg_partition_columns

The *pg_partition_columns* system view is used to show the partition key columns of a partitioned table.

Table I.1 pg_catalog.pg_partition_columns

column	type	references	description
schemaname	name		The name of the schema the partitioned table is in.
tablename	name		The table name of the top-level parent table.
columnname	name		The name of the partition key column.
partitionlevel	smallint		The level of this subpartition in the hierarchy.
position_in_partition_key	integer		For list partitions you can have a composite (multi-column) partition key. This shows the position of the column in a composite key.

pg_partition_rule

The *pg_partition_rule* system catalog table is used to track partitioned tables, their check constraints, and data containment rules.

Table I.1 pg_catalog.pg_partition_rule

column	type	references	description
paroid	oid	<i>pg_class.oid</i>	The table object identifier (OID) of the partitioning level of which this part is an instance.
parchildrelid	oid	<i>pg_class.oid</i>	The OID of the partition (child table).
parparentrule	oid	<i>pg_partition_rule.paroid</i>	The OID of the rule associated with the parent table of this partition.
parname	name		The given name of this partition.
parisdefault	boolean		Whether or not this partition is a default partition.
parruleord	smallint		For range partitioned tables, the rank of this partition on this level of the partition hierarchy.
parrangestartincl	boolean		For range partitioned tables, whether or not the starting value is inclusive.
parrangeendincl	boolean		For range partitioned tables, whether or not the ending value is inclusive.
parrangestart	text		For range partitioned tables, the starting value of the range.
parrangeend	text		For range partitioned tables, the ending value of the range.
parrangeevery	text		For range partitioned tables, the interval value of the EVERY clause.
parlistvalues	text		For list partitioned tables, the list of values assigned to this partition.
parreloptions	text		An array describing the storage characteristics of the particular partition.

pg_partition_templates

The `pg_partition_templates` system view is used to show the subpartitions that were created using a subpartition template.

Table I.1 `pg_catalog.pg_partition_templates`

column	type	references	description
<code>schemaname</code>	name		The name of the schema the partitioned table is in.
<code>tablename</code>	name		The table name of the top-level parent table.
<code>partitionname</code>	name		The name of the subpartition (this is the name to use if referring to the partition in an <code>ALTER TABLE</code> command). <code>NULL</code> if the partition was not given a name at create time or generated by an <code>EVERY</code> clause.
<code>partitiontype</code>	text		The type of subpartition (range or list).
<code>partitionlevel</code>	smallint		The level of this subpartition in the hierarchy.
<code>partitionrank</code>	bigint		For range partitions, the rank of the partition compared to other partitions of the same level.
<code>partitionposition</code>	smallint		The rule order position of this subpartition.
<code>partitionlistvalues</code>	text		For list partitions, the list value(s) associated with this subpartition.
<code>partitionrangestart</code>	text		For range partitions, the start value of this subpartition.
<code>partitionstartinclusive</code>	boolean		<code>T</code> if the start value is included in this subpartition. <code>F</code> if it is excluded.
<code>partitionrangeend</code>	text		For range partitions, the end value of this subpartition.
<code>partitionendinclusive</code>	boolean		<code>T</code> if the end value is included in this subpartition. <code>F</code> if it is excluded.
<code>partitioneveryclause</code>	text		The <code>EVERY</code> clause (interval) of this subpartition.
<code>partitionisdefault</code>	boolean		<code>T</code> if this is a default subpartition, otherwise <code>F</code> .
<code>partitionboundary</code>	text		The entire partition specification for this subpartition.

pg_pltemplate

The *pg_pltemplate* system catalog table stores template information for procedural languages. A template for a language allows the language to be created in a particular database by a simple `CREATE LANGUAGE` command, with no need to specify implementation details. Unlike most system catalogs, *pg_pltemplate* is shared across all databases of Greenplum system: there is only one copy of *pg_pltemplate* per system, not one per database. This allows the information to be accessible in each database as it is needed.

There are not currently any commands that manipulate procedural language templates; to change the built-in information, a superuser must modify the table using ordinary `INSERT`, `DELETE`, or `UPDATE` commands.

Table I.1 pg_catalog.pg_pltemplate

column	type	references	description
tmplname	name		Name of the language this template is for
tmpltrusted	boolean		True if language is considered trusted
tmplhandler	text		Name of call handler function
tmplvalidator	text		Name of validator function, or NULL if none
tmpllibrary	text		Path of shared library that implements language
tmplacl	aclitem[]		Access privileges for template (not yet implemented).

pg_proc

The *pg_proc* system catalog table stores information about functions (or procedures), both built-in functions and those defined by `CREATE FUNCTION`. The table contains data for aggregate and window functions as well as plain functions. If *proisagg* is true, there should be a matching row in *pg_aggregate*. If *proiswin* is true, there should be a matching row in *pg_window*.

For compiled functions, both built-in and dynamically loaded, *prosrc* contains the function's C-language name (link symbol). For all other currently-known language types, *prosrc* contains the function's source text. *probin* is unused except for dynamically-loaded C functions, for which it gives the name of the shared library file containing the function.

Table I.1 pg_catalog.pg_proc

column	type	references	description
proname	name		Name of the function.
pronamespace	oid	<i>pg_namespace.oid</i>	The OID of the namespace that contains this function.
proowner	oid	<i>pg_authid.oid</i>	Owner of the function.
prolang	oid	<i>pg_language.oid</i>	Implementation language or call interface of this function.
proisagg	boolean		Function is an aggregate function.
prosecdef	boolean		Function is a security definer (for example, a 'setuid' function).
proisstrict	boolean		Function returns NULL if any call argument is NULL. In that case the function will not actually be called at all. Functions that are not strict must be prepared to handle NULL inputs.
proretset	boolean		Function returns a set (multiple values of the specified data type).
provolatile	char		Tells whether the function's result depends only on its input arguments, or is affected by outside factors. <i>i</i> = <i>immutable</i> (always delivers the same result for the same inputs), <i>s</i> = <i>stable</i> (results (for fixed inputs) do not change within a scan), or <i>v</i> = <i>volatile</i> (results may change at any time or functions with side-effects).
pronargs	int2		Number of arguments.
prorettype	oid	<i>pg_type.oid</i>	Data type of the return value.
proiswin	boolean		Function is neither an aggregate nor a scalar function, but a pure window function.

Table I.1 pg_catalog.pg_proc

column	type	references	description
proargtypes	oidvector	<i>pg_type.oid</i>	An array with the data types of the function arguments. This includes only input arguments (including INOUT arguments), and thus represents the call signature of the function.
proallargtypes	oid[]	<i>pg_type.oid</i>	An array with the data types of the function arguments. This includes all arguments (including OUT and INOUT arguments); however, if all the arguments are IN arguments, this field will be null. Note that subscripting is 1-based, whereas for historical reasons <i>proargtypes</i> is subscripted from 0.
proargmodes	char[]		An array with the modes of the function arguments: i = IN, o = OUT, b = INOUT. If all the arguments are IN arguments, this field will be null. Note that subscripts correspond to positions of <i>proallargtypes</i> not <i>proargtypes</i> .
proargnames	text[]		An array with the names of the function arguments. Arguments without a name are set to empty strings in the array. If none of the arguments have a name, this field will be null. Note that subscripts correspond to positions of <i>proallargtypes</i> not <i>proargtypes</i> .
prosrc	text		This tells the function handler how to invoke the function. It might be the actual source code of the function for interpreted languages, a link symbol, a file name, or just about anything else, depending on the implementation language/call convention.
probin	bytea		Additional information about how to invoke the function. Again, the interpretation is language-specific.
proacl	aclitem[]		Access privileges for the function as given by GRANT/REVOKE.

pg_type

The *pg_type* system catalog table stores information about data types. Base types (scalar types) are created with `CREATE TYPE`, and domains with `CREATE DOMAIN`. A composite type is automatically created for each table in the database, to represent the row structure of the table. It is also possible to create composite types with `CREATE TYPE AS`.

Table I.1 `pg_catalog.pg_type`

column	type	references	description
<code>typname</code>	<code>name</code>		Data type name.
<code>typnamespace</code>	<code>oid</code>	<i>pg_namespace.oid</i>	The OID of the namespace that contains this type.
<code>typowner</code>	<code>oid</code>	<i>pg_authid.oid</i>	Owner of the type.
<code>typelen</code>	<code>int2</code>		For a fixed-size type, <i>typelen</i> is the number of bytes in the internal representation of the type. But for a variable-length type, <i>typelen</i> is negative. -1 indicates a 'varlena' type (one that has a length word), -2 indicates a null-terminated C string.
<code>typbyval</code>	<code>boolean</code>		Determines whether internal routines pass a value of this type by value or by reference. <i>typbyval</i> had better be false if <i>typelen</i> is not 1, 2, or 4 (or 8 on machines where Datum is 8 bytes). Variable-length types are always passed by reference. Note that <i>typbyval</i> can be false even if the length would allow pass-by-value; this is currently true for type <code>float4</code> , for example.
<code>typtype</code>	<code>char</code>		<code>b</code> for a base type, <code>c</code> for a composite type, <code>d</code> for a domain, or <code>p</code> for a pseudo-type.
<code>typisdefined</code>	<code>boolean</code>		True if the type is defined, false if this is a placeholder entry for a not-yet-defined type. When false, nothing except the type name, namespace, and OID can be relied on.
<code>typdelim</code>	<code>char</code>		Character that separates two values of this type when parsing array input. Note that the delimiter is associated with the array element data type, not the array data type.
<code>typreid</code>	<code>oid</code>	<i>pg_class.oid</i>	If this is a composite type, then this column points to the <i>pg_class</i> entry that defines the corresponding table. (For a free-standing composite type, the <i>pg_class</i> entry does not really represent a table, but it is needed anyway for the type's <i>pg_attribute</i> entries to link to.) Zero for non-composite types.

Table I.1 pg_catalog.pg_type

column	type	references	description
typelem	oid	<i>pg_type.oid</i>	If not 0 then it identifies another row in <i>pg_type</i> . The current type can then be subscripted like an array yielding values of type <i>typelem</i> . A true array type is variable length (<i>typlen</i> = -1), but some fixed-length (<i>typlen</i> > 0) types also have nonzero <i>typelem</i> , for example <i>name</i> and <i>point</i> . If a fixed-length type has a <i>typelem</i> then its internal representation must be some number of values of the <i>typelem</i> data type with no other data. Variable-length array types have a header defined by the array subroutines.
typinput	regproc	<i>pg_proc.oid</i>	Input conversion function (text format).
typoutput	regproc	<i>pg_proc.oid</i>	Output conversion function (text format).
typreceive	regproc	<i>pg_proc.oid</i>	Input conversion function (binary format), or 0 if none.
typsend	regproc	<i>pg_proc.oid</i>	Output conversion function (binary format), or 0 if none.
typanalyze	regproc	<i>pg_proc.oid</i>	Custom ANALYZE function, or 0 to use the standard function.
typalign	char		The alignment required when storing a value of this type. It applies to storage on disk as well as most representations of the value inside Greenplum Database. When multiple values are stored consecutively, such as in the representation of a complete row on disk, padding is inserted before a datum of this type so that it begins on the specified boundary. The alignment reference is the beginning of the first datum in the sequence. Possible values are: c = char alignment (no alignment needed). s = short alignment (2 bytes on most machines). i = int alignment (4 bytes on most machines). d = double alignment (8 bytes on many machines, but not all).
typstorage	char		For varlena types (those with <i>typlen</i> = -1) tells if the type is prepared for toasting and what the default strategy for attributes of this type should be. Possible values are: p: Value must always be stored plain. e: Value can be stored in a secondary relation (if relation has one, see <i>pg_class.reltoastrelid</i>). m: Value can be stored compressed inline. x: Value can be stored compressed inline or stored in secondary storage. Note that m columns can also be moved out to secondary storage, but only as a last resort (e and x columns are moved first).
typnotnull	boolean		Represents a not-null constraint on a type. Used for domains only.

Table I.1 pg_catalog.pg_type

column	type	references	description
typbasetype	oid	<i>pg_type.oid</i>	Identifies the type that a domain is based on. Zero if this type is not a domain.
typtypmod	int4		Domains use <i>typtypmod</i> to record the <i>typmod</i> to be applied to their base type (-1 if base type does not use a <i>typmod</i>). -1 if this type is not a domain.
typndims	int4		The number of array dimensions for a domain that is an array (if <i>typbasetype</i> is an array type; the domain's <i>typelem</i> will match the base type's <i>typelem</i>). Zero for types other than array domains.
typdefaultbin	text		If not null, it is the <code>nodeToString()</code> representation of a default expression for the type. This is only used for domains.
typdefault	text		Null if the type has no associated default value. If not null, <i>typdefault</i> must contain a human-readable version of the default expression represented by <i>typdefaultbin</i> . If <i>typdefaultbin</i> is null and <i>typdefault</i> is not, then <i>typdefault</i> is the external representation of the type's default value, which may be fed to the type's input converter to produce a constant.

pg_resqueue

The *pg_resqueue* system catalog table contains information about Greenplum Database resource queues, which are used for the workload management feature. This table is populated only on the master. This table is defined in the *pg_global* tablespace, meaning it is globally shared across all databases in the system.

Table I.1 pg_catalog.pg_resqueue

column	type	references	description
rsqname	name		The name of the resource queue.
rsqcountlimit	real		The active query threshold of the resource queue.
rsqcostlimit	real		The query cost threshold of the resource queue.
rsqovercommit	boolean		Allows queries that exceed the cost threshold to run when the system is idle.
rsqignorecostlimit	real		The query cost limit of what is considered a 'small query'. Queries with a cost under this limit will not be queued and run immediately.

pg_resqueue_status

The *pg_resqueue_status* view allows administrators to see status and activity for a workload management resource queue. It shows how many queries are waiting to run and how many queries are currently active in the system from a particular resource queue.

Table I.1 pg_catalog.pg_resqueue_status

column	type	references	description
rsqname	name	<i>pg_resqueue.rsqname</i>	The name of the resource queue.
rsqcountlimit	real	<i>pg_resqueue.countlimit</i>	The active query threshold of the resource queue. A value of -1 means no limit.
rsqcountvalue	real		The number of active query slots currently being used in the resource queue.
rsqcostlimit	real	<i>pg_resqueue.costlimit</i>	The query cost threshold of the resource queue. A value of -1 means no limit.
rsqcostvalue	real		The total cost of all statements currently in the resource queue.
rsqwaiters	integer		The number of statements currently waiting in the resource queue.
rsqholders	integer		The number of statements currently running on the system from this resource queue.

pg_rewrite

The *pg_rewrite* system catalog table stores rewrite rules for tables and views. *pg_class.relhasrules* must be true if a table has any rules in this catalog.

Table I.1 pg_catalog.pg_rewrite

column	type	references	description
rulename	name		Rule name.
ev_class	oid	<i>pg_class.oid</i>	The table this rule is for.
ev_attr	int2		The column this rule is for (currently, always zero to indicate the whole table).
ev_type	char		Event type that the rule is for: 1 = SELECT, 2 = UPDATE, 3 = INSERT, 4 = DELETE.
is_instead	boolean		True if the rule is an INSTEAD rule.
ev_qual	text		Expression tree (in the form of a <code>nodeToString()</code> representation) for the rule's qualifying condition.
ev_action	text		Query tree (in the form of a <code>nodeToString()</code> representation) for the rule's action.

pg_roles

The view *pg_roles* provides access to information about database roles. This is simply a publicly readable view of *pg_authid* that blanks out the password field. This view explicitly exposes the OID column of the underlying table, since that is needed to do joins to other catalogs.

Table I.1 pg_catalog.pg_roles

column	type	references	description
rolname	name		Role name
rolsuper	bool		Role has superuser privileges
rolinherit	bool		Role automatically inherits privileges of roles it is a member of
rolcreaterole	bool		Role may create more roles
rolcreatedb	bool		Role may create databases
rolcatupdate	bool		Role may update system catalogs directly. (Even a superuser may not do this unless this column is true.)
rolcanlogin	bool		Role may log in. That is, this role can be given as the initial session authorization identifier
rolconnlimit	int4		For roles that can log in, this sets maximum number of concurrent connections this role can make. -1 means no limit
rolpassword	text		Not the password (always reads as *****)
rolvaliduntil	timestamptz		Password expiry time (only used for password authentication); NULL if no expiration
rolconfig	text[]		Session defaults for run-time configuration variables
oid	oid	<i>pg_authid.oid</i>	Object ID of role

pg_shdepend

The *pg_shdepend* system catalog table records the dependency relationships between database objects and shared objects, such as roles. This information allows Greenplum Database to ensure that those objects are unreferenced before attempting to delete them. See also *pg_depend*, which performs a similar function for dependencies involving objects within a single database. Unlike most system catalogs, *pg_shdepend* is shared across all databases of Greenplum system: there is only one copy of *pg_shdepend* per system, not one per database.

In all cases, a *pg_shdepend* entry indicates that the referenced object may not be dropped without also dropping the dependent object. However, there are several subflavors identified by *deptype*:

- **SHARED_DEPENDENCY_OWNER (o)** — The referenced object (which must be a role) is the owner of the dependent object.
- **SHARED_DEPENDENCY_ACL (a)** — The referenced object (which must be a role) is mentioned in the ACL (access control list) of the dependent object.
- **SHARED_DEPENDENCY_PIN (p)** — There is no dependent object; this type of entry is a signal that the system itself depends on the referenced object, and so that object must never be deleted. Entries of this type are created only by system initialization. The columns for the dependent object contain zeroes.

Table I.1 pg_catalog.pg_shdepend

column	type	references	description
dbid	oid	<i>pg_database.oid</i>	The OID of the database the dependent object is in, or zero for a shared object.
classid	oid	<i>pg_class.oid</i>	The OID of the system catalog the dependent object is in.
objid	oid	any OID column	The OID of the specific dependent object.
objsubid	int4		For a table column, this is the column number. For all other object types, this column is zero.
refclassid	oid	<i>pg_class.oid</i>	The OID of the system catalog the referenced object is in (must be a shared catalog).
refobjid	oid	any OID column	The OID of the specific referenced object.
refobjsubid	int4		For a table column, this is the referenced column number. For all other object types, this column is zero.
deptype	char		A code defining the specific semantics of this dependency relationship.

pg_shdescription

The *pg_shdescription* system catalog table stores optional descriptions (comments) for shared database objects. Descriptions can be manipulated with the `COMMENT` command and viewed with `psql`'s `\d` meta-commands. See also *pg_description*, which performs a similar function for descriptions involving objects within a single database. Unlike most system catalogs, *pg_shdescription* is shared across all databases of a Greenplum system: there is only one copy of *pg_shdescription* per system, not one per database.

Table I.1 pg_catalog.pg_shdescription

column	type	references	description
objoid	oid	any OID column	The OID of the object this description pertains to.
classoid	oid	<i>pg_class.oid</i>	The OID of the system catalog this object appears in
description	text		Arbitrary text that serves as the description of this object.

pg_stat_activity

The view *pg_stat_activity* shows one row per server process and details about its associated user session and query. The columns that report data on the current query are available unless the parameter *stats_command_string* has been turned off. Furthermore, these columns are only visible if the user examining the view is a superuser or the same as the user owning the process being reported on.

Table I.1 pg_catalog.pg_stat_activity

column	type	references	description
datid	oid	<i>pg_database.oid</i>	Database OID
datname	name		Database name
procpid	integer		Process ID of the server process
sess_id	integer		Session ID
usesysid	oid	<i>pg_authid.oid</i>	Role OID
username	name		Role name
current_query	text		Current query that process is running
waiting	boolean		True if waiting on a lock, false if not waiting
query_start	timestampz		Time query began execution
backend_start	timestampz		Time backend process was started
client_addr	inet		Client address
client_port	integer		Client port

pg_statistic

The *pg_statistic* system catalog table stores statistical data about the contents of the database. Entries are created by `ANALYZE` and subsequently used by the query planner. There is one entry for each table column that has been analyzed. Note that all the statistical data is inherently approximate, even assuming that it is up-to-date.

pg_statistic also stores statistical data about the values of index expressions. These are described as if they were actual data columns; in particular, *starelid* references the index. No entry is made for an ordinary non-expression index column, however, since it would be redundant with the entry for the underlying table column.

Since different kinds of statistics may be appropriate for different kinds of data, *pg_statistic* is designed not to assume very much about what sort of statistics it stores. Only extremely general statistics (such as nullness) are given dedicated columns in *pg_statistic*. Everything else is stored in slots, which are groups of associated columns whose content is identified by a code number in one of the slot's columns.

pg_statistic should not be readable by the public, since even statistical information about a table's contents may be considered sensitive (for example: minimum and maximum values of a salary column). *pg_stats* is a publicly readable view on *pg_statistic* that only exposes information about those tables that are readable by the current user.

Table I.1 pg_catalog.pg_statistic

column	type	references	description
starelid	oid	<i>pg_class.oid</i>	The table or index that the described column belongs to.
staattnum	int2	<i>pg_attribute.attnum</i>	The number of the described column.
stanullfrac	float4		The fraction of the column's entries that are null.
stawidth	int4		The average stored width, in bytes, of nonnull entries.
stadistinct	float4		The number of distinct nonnull data values in the column. A value greater than zero is the actual number of distinct values. A value less than zero is the negative of a fraction of the number of rows in the table (for example, a column in which values appear about twice on the average could be represented by <code>stadistinct = -0.5</code>). A zero value means the number of distinct values is unknown.
stakindN	int2		A code number indicating the kind of statistics stored in the Nth slot of the <i>pg_statistic</i> row.

Table I.1 pg_catalog.pg_statistic

column	type	references	description
staopN	oid	<i>pg_operator.oid</i>	An operator used to derive the statistics stored in the <i>N</i> th slot. For example, a histogram slot would show the < operator that defines the sort order of the data.
stanumbersN	float4[]		Numerical statistics of the appropriate kind for the <i>N</i> th slot, or NULL if the slot kind does not involve numerical values.
stavaluesN	anyarray		Column data values of the appropriate kind for the <i>N</i> th slot, or NULL if the slot kind does not store any data values. Each array's element values are actually of the specific column's data type, so there is no way to define these columns' type more specifically than <i>anyarray</i> .

pg_stat_resqueues

The *pg_stat_resqueues* view allows administrators to view metrics about a resource queue's workload over time. To allow statistics to be collected for this view, you must enable the *stats_queue_level* server configuration parameter on the Greenplum Database master instance. Enabling the collection of these metrics does incur a small performance penalty, as each statement submitted through a resource queue must be logged in the system catalog tables.

Table I.1 pg_catalog.pg_stat_resqueues

column	type	references	description
queueoid	oid		The OID of the resource queue.
queuename	name		The name of the resource queue.
n_queries_exec	bigint		Number of queries submitted for execution from this resource queue.
n_queries_wait	bigint		Number of queries submitted to this resource queue that had to wait before they could execute.
elapsed_exec	bigint		Total elapsed execution time for statements submitted through this resource queue.
elapsed_wait	bigint		Total elapsed time that statements submitted through this resource queue had to wait before they were executed.

pg_tablespace

The *pg_tablespace* system catalog table stores information about the available tablespaces. Tables can be placed in particular tablespaces to aid administration of disk layout. Unlike most system catalogs, *pg_tablespace* is shared across all databases of a Greenplum system: there is only one copy of *pg_tablespace* per system, not one per database.

Table I.1 pg_catalog.pg_tablespace

column	type	references	description
spcname	name		Tablespace name.
spcowner	oid	<i>pg_authid.oid</i>	Owner of the tablespace, usually the user who created it.
spcllocation	text[]		Location (directory path) of the tablespace on the master instance.
spcacl	aclitem[]		Tablespace access privileges.
spcprilocations	text[]		Location (directory path) of the tablespace on the primary segment instances.
spcmrilocations	text[]		Location (directory path) of the tablespace on the mirror segment instances.

pg_trigger

The *pg_trigger* system catalog table stores triggers on tables.

Table I.1 pg_catalog.pg_trigger

column	type	references	description
tgrelid	oid	<i>pg_class.oid</i>	The table this trigger is on.
tgname	name		Trigger name (must be unique among triggers of same table).
tgfoid	oid	<i>pg_proc.oid</i>	The function to be called.
tgtype	int2		Bit mask identifying trigger conditions.
tgenabled	boolean		True if trigger is enabled.
tgisconstraint	boolean		True if trigger implements a referential integrity constraint.
tgconstrname	name		Referential integrity constraint name.
tgconstrrelid	oid	<i>pg_class.oid</i>	The table referenced by an referential integrity constraint.
tgdeferrable	boolean		True if deferrable.
tginitdeferred	boolean		True if initially deferred.
tnargs	int2		Number of argument strings passed to trigger function.
tgattr	int2vector		Currently unused.
tgargs	bytea		Argument strings to pass to trigger, each NULL-terminated.

pg_window

The *pg_window* table stores information about window functions. Window functions are often used to compose complex OLAP (online analytical processing) queries. Window functions are applied to partitioned result sets within the scope of a single query expression. A window partition is a subset of rows returned by a query, as defined in a special *OVER()* clause. Typical window functions are *rank*, *dense_rank*, and *row_number*. Each entry in *pg_window* is an extension of an entry in *pg_proc*. The *pg_proc* entry carries the window function's name, input and output data types, and other information that is similar to ordinary functions.

Table I.1 pg_catalog.pg_window

column	type	references	description
winfnoid	regproc	<i>pg_proc.oid</i>	The OID in <i>pg_proc</i> of the window function.
winrequireorder	boolean		The window function requires its window specification to have an <i>ORDER BY</i> clause.
winallowframe	boolean		The window function permits its window specification to have a <i>ROWS</i> or <i>RANGE</i> framing clause.
winpeercount	boolean		The peer group row count is required to compute this window function, so the Window node implementation must 'look ahead' as necessary to make this available in its internal state.
wincount	boolean		The partition row count is required to compute this window function.
winfunc	regproc	<i>pg_proc.oid</i>	The OID in <i>pg_proc</i> of a function to compute the value of an immediate-type window function.
winprefunc	regproc	<i>pg_proc.oid</i>	The OID in <i>pg_proc</i> of a preliminary window function to compute the partial value of a deferred-type window function.
winpretype	oid	<i>pg_type.oid</i>	The OID in <i>pg_type</i> of the preliminary window function's result type.

Table I.1 pg_catalog.pg_window

column	type	references	description
winfunc	regproc	pg_proc.oid	The OID in <i>pg_proc</i> of a function to compute the final value of a deferred-type window function from the partition row count and the result of <i>winprefunc</i> .
winkind	char		A character indicating membership of the window function in a class of related functions: w - ordinary window functions n - NTILE functions f - FIRST_VALUE functions l - LAST_VALUE functions g - LAG functions d - LEAD functions

J. SQL 2008 Optional Feature Compliance

The following table is list of features described in the 2008 SQL standard. Features that are supported in Greenplum Database are marked as YES in the ‘Supported’ column, features that are not implemented are marked as NO.

For more information on Greenplum features and SQL compliance, see “[Feature Summary](#)” on page 15.

Table J.1 SQL 2008 Optional Feature Compliance Details

ID	Feature	Supported	Comments
B011	Embedded Ada	NO	
B012	Embedded C	NO	Due to issues with PostgreSQL ecpg
B013	Embedded COBOL	NO	
B014	Embedded Fortran	NO	
B015	Embedded MUMPS	NO	
B016	Embedded Pascal	NO	
B017	Embedded PL/I	NO	
B021	Direct SQL	YES	
B031	Basic dynamic SQL	NO	
B032	Extended dynamic SQL	NO	
B033	Untyped SQL-invoked function arguments	NO	
B034	Dynamic specification of cursor attributes	NO	
B035	Non-extended descriptor names	NO	
B041	Extensions to embedded SQL exception declarations	NO	
B051	Enhanced execution rights	NO	
B111	Module language Ada	NO	
B112	Module language C	NO	
B113	Module language COBOL	NO	
B114	Module language Fortran	NO	
B115	Module language MUMPS	NO	
B116	Module language Pascal	NO	
B117	Module language PL/I	NO	
B121	Routine language Ada	NO	
B122	Routine language C	NO	
B123	Routine language COBOL	NO	
B124	Routine language Fortran	NO	

Table J.1 SQL 2008 Optional Feature Compliance Details

ID	Feature	Supported	Comments
B125	Routine language MUMPS	NO	
B126	Routine language Pascal	NO	
B127	Routine language PL/I	NO	
B128	Routine language SQL	NO	
E011	Numeric data types	YES	
E011-01	INTEGER and SMALLINT data types	YES	
E011-02	DOUBLE PRECISION and FLOAT data types	YES	
E011-03	DECIMAL and NUMERIC data types	YES	
E011-04	Arithmetic operators	YES	
E011-05	Numeric comparison	YES	
E011-06	Implicit casting among the numeric data types	YES	
E021	Character data types	YES	
E021-01	CHARACTER data type	YES	
E021-02	CHARACTER VARYING data type	YES	
E021-03	Character literals	YES	
E021-04	CHARACTER_LENGTH function	YES	Trims trailing spaces from CHARACTER values before counting
E021-05	OCTET_LENGTH function	YES	
E021-06	SUBSTRING function	YES	
E021-07	Character concatenation	YES	
E021-08	UPPER and LOWER functions	YES	
E021-09	TRIM function	YES	
E021-10	Implicit casting among the character string types	YES	
E021-11	POSITION function	YES	
E021-12	Character comparison	YES	
E031	Identifiers	YES	
E031-01	Delimited identifiers	YES	
E031-02	Lower case identifiers	YES	
E031-03	Trailing underscore	YES	
E051	Basic query specification	YES	
E051-01	SELECT DISTINCT	YES	
E051-02	GROUP BY clause	YES	
E051-03	GROUP BY can contain columns not in SELECT list	YES	

Table J.1 SQL 2008 Optional Feature Compliance Details

ID	Feature	Supported	Comments
E051-04	SELECT list items can be renamed	YES	
E051-05	HAVING clause	YES	
E051-06	Qualified * in SELECT list	YES	
E051-07	Correlation names in the FROM clause	YES	
E051-08	Rename columns in the FROM clause	YES	
E061	Basic predicates and search conditions	YES	
E061-01	Comparison predicate	YES	
E061-02	BETWEEN predicate	YES	
E061-03	IN predicate with list of values	YES	
E061-04	LIKE predicate	YES	
E061-05	LIKE predicate ESCAPE clause	YES	
E061-06	NULL predicate	YES	
E061-07	Quantified comparison predicate	YES	
E061-08	EXISTS predicate	YES	Not all uses work in Greenplum
E061-09	Subqueries in comparison predicate	YES	
E061-11	Subqueries in IN predicate	YES	
E061-12	Subqueries in quantified comparison predicate	YES	
E061-13	Correlated subqueries	NO	
E061-14	Search condition	YES	
E071	Basic query expressions	YES	
E071-01	UNION DISTINCT table operator	YES	
E071-02	UNION ALL table operator	YES	
E071-03	EXCEPT DISTINCT table operator	YES	
E071-05	Columns combined via table operators need not have exactly the same data type	YES	
E071-06	Table operators in subqueries	YES	
E081	Basic Privileges	NO	Partial sub-feature support
E081-01	SELECT privilege	YES	
E081-02	DELETE privilege	YES	
E081-03	INSERT privilege at the table level	YES	
E081-04	UPDATE privilege at the table level	YES	
E081-05	UPDATE privilege at the column level	NO	
E081-06	REFERENCES privilege at the table level	NO	
E081-07	REFERENCES privilege at the column level	NO	

Table J.1 SQL 2008 Optional Feature Compliance Details

ID	Feature	Supported	Comments
E081-08	WITH GRANT OPTION	YES	
E081-09	USAGE privilege	YES	
E081-10	EXECUTE privilege	YES	
E091	Set Functions	YES	
E091-01	AVG	YES	
E091-02	COUNT	YES	
E091-03	MAX	YES	
E091-04	MIN	YES	
E091-05	SUM	YES	
E091-06	ALL quantifier	YES	
E091-07	DISTINCT quantifier	YES	
E101	Basic data manipulation	YES	
E101-01	INSERT statement	YES	
E101-03	Searched UPDATE statement	YES	
E101-04	Searched DELETE statement	YES	
E111	Single row SELECT statement	YES	
E121	Basic cursor support	YES	
E121-01	DECLARE CURSOR	YES	
E121-02	ORDER BY columns need not be in select list	YES	
E121-03	Value expressions in ORDER BY clause	YES	
E121-04	OPEN statement	YES	
E121-06	Positioned UPDATE statement	NO	
E121-07	Positioned DELETE statement	NO	
E121-08	CLOSE statement	YES	
E121-10	FETCH statement implicit NEXT	YES	
E121-17	WITH HOLD cursors	YES	
E131	Null value support	YES	
E141	Basic integrity constraints	YES	
E141-01	NOT NULL constraints	YES	
E141-02	UNIQUE constraints of NOT NULL columns	YES	Must be the same as or a superset of the Greenplum distribution key
E141-03	PRIMARY KEY constraints	YES	Must be the same as or a superset of the Greenplum distribution key

Table J.1 SQL 2008 Optional Feature Compliance Details

ID	Feature	Supported	Comments
E141-04	Basic FOREIGN KEY constraint with the NO ACTION default for both referential delete action and referential update action	NO	
E141-06	CHECK constraints	YES	
E141-07	Column defaults	YES	
E141-08	NOT NULL inferred on PRIMARY KEY	YES	
E141-10	Names in a foreign key can be specified in any order	YES	Foreign keys can be declared but are not enforced in Greenplum
E151	Transaction support	YES	
E151-01	COMMIT statement	YES	
E151-02	ROLLBACK statement	YES	
E152	Basic SET TRANSACTION statement	YES	
E152-01	ISOLATION LEVEL SERIALIZABLE clause	YES	
E152-02	READ ONLY and READ WRITE clauses	YES	
E153	Updatable queries with subqueries	NO	
E161	SQL comments using leading double minus	YES	
E171	SQLSTATE support	YES	
E182	Module language	NO	
F021	Basic information schema	YES	
F021-01	COLUMNS view	YES	
F021-02	TABLES view	YES	
F021-03	VIEWS view	YES	
F021-04	TABLE_CONSTRAINTS view	YES	
F021-05	REFERENTIAL_CONSTRAINTS view	YES	
F021-06	CHECK_CONSTRAINTS view	YES	
F031	Basic schema manipulation	YES	
F031-01	CREATE TABLE statement to create persistent base tables	YES	
F031-02	CREATE VIEW statement	YES	
F031-03	GRANT statement	YES	
F031-04	ALTER TABLE statement: ADD COLUMN clause	YES	
F031-13	DROP TABLE statement: RESTRICT clause	YES	
F031-16	DROP VIEW statement: RESTRICT clause	YES	
F031-19	REVOKE statement: RESTRICT clause	YES	
F032	CASCADE drop behavior	YES	

Table J.1 SQL 2008 Optional Feature Compliance Details

ID	Feature	Supported	Comments
F033	ALTER TABLE statement: DROP COLUMN clause	YES	
F034	Extended REVOKE statement	YES	
F034-01	REVOKE statement performed by other than the owner of a schema object	YES	
F034-02	REVOKE statement: GRANT OPTION FOR clause	YES	
F034-03	REVOKE statement to revoke a privilege that the grantee has WITH GRANT OPTION	YES	
F041	Basic joined table	YES	
F041-01	Inner join (but not necessarily the INNER keyword)	YES	
F041-02	INNER keyword	YES	
F041-03	LEFT OUTER JOIN	YES	
F041-04	RIGHT OUTER JOIN	YES	
F041-05	Outer joins can be nested	YES	
F041-07	The inner table in a left or right outer join can also be used in an inner join	YES	
F041-08	All comparison operators are supported (rather than just =)	YES	
F051	Basic date and time	YES	
F051-01	DATE data type (including support of DATE literal)	YES	
F051-02	TIME data type (including support of TIME literal) with fractional seconds precision of at least 0	YES	
F051-03	TIMESTAMP data type (including support of TIMESTAMP literal) with fractional seconds precision of at least 0 and 6	YES	
F051-04	Comparison predicate on DATE, TIME, and TIMESTAMP data types	YES	
F051-05	Explicit CAST between datetime types and character string types	YES	
F051-06	CURRENT_DATE	YES	
F051-07	LOCALTIME	YES	
F051-08	LOCALTIMESTAMP	YES	
F052	Intervals and datetime arithmetic	YES	
F053	OVERLAPS predicate	YES	
F081	UNION and EXCEPT in views	YES	
F111	Isolation levels other than SERIALIZABLE	YES	
F111-01	READ UNCOMMITTED isolation level	NO	Can be declared but is treated as a synonym for READ COMMITTED

Table J.1 SQL 2008 Optional Feature Compliance Details

ID	Feature	Supported	Comments
F111-02	READ COMMITTED isolation level	YES	
F111-03	REPEATABLE READ isolation level	NO	Can be declared but is treated as a synonym for SERIALIZABLE
F121	Basic diagnostics management	NO	
F122	Enhanced diagnostics management	NO	
F123	All diagnostics	NO	
F131-	Grouped operations	YES	
F131-01	WHERE, GROUP BY, and HAVING clauses supported in queries with grouped views	YES	
F131-02	Multiple tables supported in queries with grouped views	YES	
F131-03	Set functions supported in queries with grouped views	YES	
F131-04	Subqueries with GROUP BY and HAVING clauses and grouped views	YES	
F131-05	Single row SELECT with GROUP BY and HAVING clauses and grouped views	YES	
F171	Multiple schemas per user	YES	
F181	Multiple module support	NO	
F191	Referential delete actions	NO	
F200	TRUNCATE TABLE statement	YES	
F201	CAST function	YES	
F202	TRUNCATE TABLE: identity column restart option	NO	
F221	Explicit defaults	YES	
F222	INSERT statement: DEFAULT VALUES clause	YES	
F231	Privilege tables	YES	
F231-01	TABLE_PRIVILEGES view	YES	
F231-02	COLUMN_PRIVILEGES view	YES	
F231-03	USAGE_PRIVILEGES view	YES	
F251	Domain support		
F261	CASE expression	YES	
F261-01	Simple CASE	YES	
F261-02	Searched CASE	YES	
F261-03	NULLIF	YES	
F261-04	COALESCE	YES	
F262	Extended CASE expression	NO	

Table J.1 SQL 2008 Optional Feature Compliance Details

ID	Feature	Supported	Comments
F263	Comma-separated predicates in simple <code>CASE</code> expression	NO	
F271	Compound character literals	YES	
F281	<code>LIKE</code> enhancements	YES	
F291	<code>UNIQUE</code> predicate	NO	
F301	<code>CORRESPONDING</code> in query expressions	NO	
F302	INTERSECT table operator	YES	
F302-01	<code>INTERSECT DISTINCT</code> table operator	YES	
F302-02	<code>INTERSECT ALL</code> table operator	YES	
F304	<code>EXCEPT ALL</code> table operator		
F311	Schema definition statement	YES	Partial sub-feature support
F311-01	<code>CREATE SCHEMA</code>	YES	
F311-02	<code>CREATE TABLE</code> for persistent base tables	YES	
F311-03	<code>CREATE VIEW</code>	YES	
F311-04	<code>CREATE VIEW: WITH CHECK OPTION</code>	NO	
F311-05	<code>GRANT</code> statement	YES	
F312	<code>MERGE</code> statement	NO	
F313	Enhanced <code>MERGE</code> statement	NO	
F321	User authorization	YES	
F341	Usage Tables	NO	
F361	Subprogram support	YES	
F381	Extended schema manipulation	YES	
F381-01	<code>ALTER TABLE</code> statement: <code>ALTER COLUMN</code> clause		Some limitations on altering distribution key columns
F381-02	<code>ALTER TABLE</code> statement: <code>ADD CONSTRAINT</code> clause		
F381-03	<code>ALTER TABLE</code> statement: <code>DROP CONSTRAINT</code> clause		
F382	Alter column data type	YES	Some limitations on altering distribution key columns
F391	Long identifiers	YES	
F392	Unicode escapes in identifiers	NO	
F393	Unicode escapes in literals	NO	
F394	Optional normal form specification	NO	
F401	Extended joined table	YES	
F401-01	<code>NATURAL JOIN</code>	YES	

Table J.1 SQL 2008 Optional Feature Compliance Details

ID	Feature	Supported	Comments
F401-02	FULL OUTER JOIN	YES	
F401-04	CROSS JOIN	YES	
F402	Named column joins for LOBs, arrays, and multisets	NO	
F403	Partitioned joined tables	NO	
F411	Time zone specification	YES	Differences regarding literal interpretation
F421	National character	YES	
F431	Read-only scrollable cursors	YES	Forward scrolling only
01	FETCH with explicit NEXT	YES	
02	FETCH FIRST	NO	
03	FETCH LAST	YES	
04	FETCH PRIOR	NO	
05	FETCH ABSOLUTE	NO	
06	FETCH RELATIVE	NO	
F441	Extended set function support	YES	
F442	Mixed column references in set functions	YES	
F451	Character set definition	NO	
F461	Named character sets	NO	
F471	Scalar subquery values	YES	
F481	Expanded NULL predicate	YES	
F491	Constraint management	YES	
F501	Features and conformance views	YES	
F501-01	SQL_FEATURES view	YES	
F501-02	SQL_SIZING view	YES	
F501-03	SQL_LANGUAGES view	YES	
F502	Enhanced documentation tables	YES	
F502-01	SQL_SIZING_PROFILES view	YES	
F502-02	SQL_IMPLEMENTATION_INFO view	YES	
F502-03	SQL_PACKAGES view	YES	
F521	Assertions	NO	
F531	Temporary tables	YES	Non-standard form
F555	Enhanced seconds precision	YES	
F561	Full value expressions	YES	
F571	Truth value tests	YES	

Table J.1 SQL 2008 Optional Feature Compliance Details

ID	Feature	Supported	Comments
F591	Derived tables	YES	
F611	Indicator data types	YES	
F641	Row and table constructors	NO	
F651	Catalog name qualifiers	YES	
F661	Simple tables	NO	
F671	Subqueries in CHECK	NO	Intentionally omitted
F672	Retrospective check constraints	YES	
F690	Collation support	NO	
F692	Enhanced collation support	NO	
F693	SQL-session and client module collations	NO	
F695	Translation support	NO	
F696	Additional translation documentation	NO	
F701	Referential update actions	NO	
F711	ALTER domain	YES	
F721	Deferrable constraints	NO	
F731	INSERT column privileges	NO	
F741	Referential MATCH types	NO	No partial match
F751	View CHECK enhancements	NO	
F761	Session management	YES	
F762	CURRENT_CATALOG	NO	
F763	CURRENT_SCHEMA	NO	
F771	Connection management	YES	
F781	Self-referencing operations	YES	
F791	Insensitive cursors	YES	
F801	Full set function	YES	
F812	Basic flagging	NO	
F813	Extended flagging	NO	
F831	Full cursor update	NO	
F841	LIKE_REGEX predicate	NO	Non-standard syntax for regex
F842	OCCURENCES_REGEX function	NO	
F843	POSITION_REGEX function	NO	
F844	SUBSTRING_REGEX function	NO	
F845	TRANSLATE_REGEX function	NO	
F846	Octet support in regular expression operators	NO	

Table J.1 SQL 2008 Optional Feature Compliance Details

ID	Feature	Supported	Comments
F847	Nonconstant regular expressions	NO	
F850	Top-level ORDER BY clause in <i>query expression</i>	YES	
F851	Top-level ORDER BY clause in subqueries	NO	
F852	Top-level ORDER BY clause in views	NO	
F855	Nested ORDER BY clause in <i>query expression</i>	NO	
F856	Nested FETCH FIRST clause in <i>query expression</i>	NO	
F857	Top-level FETCH FIRST clause in <i>query expression</i>	NO	
F858	FETCH FIRST clause in subqueries	NO	
F859	Top-level FETCH FIRST clause in views	NO	
F860	FETCH FIRST ROW <i>count</i> in FETCH FIRST clause	NO	
F861	Top-level RESULT OFFSET clause in <i>query expression</i>	NO	
F862	RESULT OFFSET clause in subqueries	NO	
F863	Nested RESULT OFFSET clause in <i>query expression</i>	NO	
F864	Top-level RESULT OFFSET clause in views	NO	
F865	OFFSET ROW <i>count</i> in RESULT OFFSET clause	NO	
S011	Distinct data types	NO	
S023	Basic structured types	NO	
S024	Enhanced structured types	NO	
S025	Final structured types	NO	
S026	Self-referencing structured types	NO	
S027	Create method by specific method name	NO	
S028	Permutable UDT options list	NO	
S041	Basic reference types	NO	
S043	Enhanced reference types	NO	
S051	Create table of type	NO	
S071	SQL paths in function and type name resolution	YES	
S091	Basic array support	NO	Greenplum has arrays, but is not fully standards compliant
S091-01	Arrays of built-in data types	NO	Partially compliant
S091-02	Arrays of distinct types	NO	
S091-03	Array expressions	NO	
S092	Arrays of user-defined types	NO	
S094	Arrays of reference types	NO	

Table J.1 SQL 2008 Optional Feature Compliance Details

ID	Feature	Supported	Comments
S095	Array constructors by query	NO	
S096	Optional array bounds	NO	
S097	Array element assignment	NO	
S098	ARRAY_AGG	NO	ORDER BY clause not supported
S111	ONLY in query expressions	YES	
S151	Type predicate	NO	
S161	Subtype treatment	NO	
S162	Subtype treatment for references	NO	
S201	SQL-invoked routines on arrays	NO	Functions can be passed Greenplum array types
S202	SQL-invoked routines on multisets	NO	
S211	User-defined cast functions	YES	
S231	Structured type locators	NO	
S232	Array locators	NO	
S233	Multiset locators	NO	
S241	Transform functions	NO	
S242	Alter transform statement	NO	
S251	User-defined orderings	NO	
S261	Specific type method	NO	
S271	Basic multiset support	NO	
S272	Multisets of user-defined types	NO	
S274	Multisets of reference types	NO	
S275	Advanced multiset support	NO	
S281	Nested collection types	NO	
S291	Unique constraint on entire row	NO	
S301	Enhanced UNNEST	NO	
S401	Distinct types based on array types	NO	
S402	Distinct types based on distinct types	NO	
S403	MAX_CARDINALITY	NO	
S404	TRIM_ARRAY	NO	
T011	Timestamp in Information Schema	NO	
T021	BINARY and VARBINARY data types	NO	
T022	Advanced support for BINARY and VARBINARY data types	NO	
T023	Compound binary literal	NO	

Table J.1 SQL 2008 Optional Feature Compliance Details

ID	Feature	Supported	Comments
T024	Spaces in binary literals	NO	
T031	BOOLEAN data type	YES	
T041	Basic LOB data type support	NO	
T042	Extended LOB data type support	NO	
T043	Multiplier T	NO	
T044	Multiplier P	NO	
T051	Row types	NO	
T052	MAX and MIN for row types	NO	
T053	Explicit aliases for all-fields reference	NO	
T061	UCS support	NO	
T071	BIGINT data type	YES	
T101	Enhanced nullability determination	NO	
T111	Updatable joins, unions, and columns	NO	
T121	WITH (excluding RECURSIVE) in query expression	NO	
T122	WITH (excluding RECURSIVE) in subquery	NO	
T131	Recursive query	NO	
T132	Recursive query in subquery	NO	
T141	SIMILAR predicate	YES	
T151	DISTINCT predicate	YES	
T152	DISTINCT predicate with negation	NO	
T171	LIKE clause in table definition	YES	
T172	AS subquery clause in table definition	YES	
T173	Extended LIKE clause in table definition	YES	
T174	Identity columns	NO	
T175	Generated columns	NO	
T176	Sequence generator support	NO	
T177	Sequence generator support: simple restart option	NO	
T178	Identity columns: simple restart option	NO	
T191	Referential action RESTRICT	NO	
T201	Comparable data types for referential constraints	NO	
T211	Basic trigger capability	NO	
T211-01	Triggers activated on UPDATE, INSERT, or DELETE of one base table	NO	
T211-02	BEFORE triggers	NO	

Table J.1 SQL 2008 Optional Feature Compliance Details

ID	Feature	Supported	Comments
T211-03	AFTER triggers	NO	
T211-04	FOR EACH ROW triggers	NO	
T211-05	Ability to specify a search condition that must be true before the trigger is invoked	NO	
T211-06	Support for run-time rules for the interaction of triggers and constraints	NO	
T211-07	TRIGGER privilege	YES	
T211-08	Multiple triggers for the same event are executed in the order in which they were created in the catalog	NO	Intentionally omitted
T212	Enhanced trigger capability	NO	
T213	INSTEAD OF triggers	NO	
T231	Sensitive cursors	YES	
T241	START TRANSACTION statement	YES	
T251	SET TRANSACTION statement: LOCAL option	NO	
T261	Chained transactions	NO	
T271	Savepoints	YES	
T272	Enhanced savepoint management	NO	
T281	SELECT privilege with column granularity	NO	
T285	Enhanced derived column names	NO	
T301	Functional dependencies	NO	
T312	OVERLAY function	YES	
T321	Basic SQL-invoked routines	NO	Partial support
T321-01	User-defined functions with no overloading	YES	
T321-02	User-defined stored procedures with no overloading	NO	
T321-03	Function invocation	YES	
T321-04	CALL statement	NO	
T321-05	RETURN statement	NO	
T321-06	ROUTINES view	YES	
T321-07	PARAMETERS view	YES	
T322	Overloading of SQL-invoked functions and procedures	YES	
T323	Explicit security for external routines	YES	
T324	Explicit security for SQL routines	NO	
T325	Qualified SQL parameter references	NO	
T326	Table functions	NO	

Table J.1 SQL 2008 Optional Feature Compliance Details

ID	Feature	Supported	Comments
T331	Basic roles	NO	
T332	Extended roles	NO	
T351	Bracketed SQL comments (<code>/* . . . */</code> comments)	YES	
T431	Extended grouping capabilities	NO	
T432	Nested and concatenated <code>GROUPING SETS</code>	NO	
T433	Multiargument <code>GROUPING</code> function	NO	
T434	<code>GROUP BY DISTINCT</code>	NO	
T441	<code>ABS</code> and <code>MOD</code> functions	YES	
T461	Symmetric <code>BETWEEN</code> predicate	YES	
T471	Result sets return value	NO	
T491	<code>LATERAL</code> derived table	NO	
T501	Enhanced <code>EXISTS</code> predicate	NO	
T511	Transaction counts	NO	
T541	Updatable table references	NO	
T561	Holdable locators	NO	
T571	Array-returning external SQL-invoked functions	NO	
T572	Multiset-returning external SQL-invoked functions	NO	
T581	Regular expression substring function	YES	
T591	<code>UNIQUE</code> constraints of possibly null columns	YES	
T601	Local cursor references	NO	
T611	Elementary OLAP operations	YES	
T612	Advanced OLAP operations	NO	Partially supported
T613	Sampling	NO	
T614	<code>NTILE</code> function	YES	
T615	<code>LEAD</code> and <code>LAG</code> functions	YES	
T616	Null treatment option for <code>LEAD</code> and <code>LAG</code> functions	NO	
T617	<code>FIRST_VALUE</code> and <code>LAST_VALUE</code> function	YES	
T618	<code>NTH_VALUE</code>	NO	Function exists in Greenplum but not all options are supported
T621	Enhanced numeric functions	YES	
T631	<code>N</code> predicate with one list element	NO	
T641	Multiple column assignment	NO	Some syntax variants supported
T651	SQL-schema statements in SQL routines	NO	
T652	SQL-dynamic statements in SQL routines	NO	

Table J.1 SQL 2008 Optional Feature Compliance Details

ID	Feature	Supported	Comments
T653	SQL-schema statements in external routines	NO	
T654	SQL-dynamic statements in external routines	NO	
T655	Cyclically dependent routines	NO	
M001	Datalinks	NO	
M002	Datalinks via SQL/CLI	NO	
M003	Datalinks via Embedded SQL	NO	
M004	Foreign data support	NO	
M005	Foreign schema support	NO	
M006	GetSQLString routine	NO	
M007	TransmitRequest	NO	
M009	GetOpts and GetStatistics routines	NO	
M010	Foreign data wrapper support	NO	
M011	Datalinks via Ada	NO	
M012	Datalinks via C	NO	
M013	Datalinks via COBOL	NO	
M014	Datalinks via Fortran	NO	
M015	Datalinks via M	NO	
M016	Datalinks via Pascal	NO	
M017	Datalinks via PL/I	NO	
M018	Foreign data wrapper interface routines in Ada	NO	
M019	Foreign data wrapper interface routines in C	NO	
M020	Foreign data wrapper interface routines in COBOL	NO	
M021	Foreign data wrapper interface routines in Fortran	NO	
M022	Foreign data wrapper interface routines in MUMPS	NO	
M023	Foreign data wrapper interface routines in Pascal	NO	
M024	Foreign data wrapper interface routines in PL/I	NO	
M030	SQL-server foreign data support	NO	
M031	Foreign data wrapper general routines	NO	
X010	XML type	NO	
X011	Arrays of XML type	NO	
X012	Multisets of XML type	NO	
X013	Distinct types of XML type	NO	
X014	Attributes of XML type	NO	
X015	Fields of XML type	NO	

Table J.1 SQL 2008 Optional Feature Compliance Details

ID	Feature	Supported	Comments
X016	Persistent XML values	NO	
X020	XMLConcat	NO	
X025	XMLCast	NO	
X030	XMLDocument	NO	
X031	XMLElement	NO	
X032	XMLForest	NO	
X034	XMLAgg	NO	
X035	XMLAgg: ORDER BY option	NO	
X036	XMLComment	NO	
X037	XMLPI	NO	
X038	XMLText	NO	
X040	Basic table mapping	NO	
X041	Basic table mapping: nulls absent	NO	
X042	Basic table mapping: null as nil	NO	
X043	Basic table mapping: table as forest	NO	
X044	Basic table mapping: table as element	NO	
X045	Basic table mapping: with target namespace	NO	
X046	Basic table mapping: data mapping	NO	
X047	Basic table mapping: metadata mapping	NO	
X048	Basic table mapping: base64 encoding of binary strings	NO	
X049	Basic table mapping: hex encoding of binary strings	NO	
X051	Advanced table mapping: nulls absent	NO	
X052	Advanced table mapping: null as nil	NO	
X053	Advanced table mapping: table as forest	NO	
X054	Advanced table mapping: table as element	NO	
X055	Advanced table mapping: target namespace	NO	
X056	Advanced table mapping: data mapping	NO	
X057	Advanced table mapping: metadata mapping	NO	
X058	Advanced table mapping: base64 encoding of binary strings	NO	
X059	Advanced table mapping: hex encoding of binary strings	NO	
X060	XMLParse: Character string input and CONTENT option	NO	

Table J.1 SQL 2008 Optional Feature Compliance Details

ID	Feature	Supported	Comments
X061	XMLParse: Character string input and DOCUMENT option	NO	
X065	XMLParse: BLOB input and CONTENT option	NO	
X066	XMLParse: BLOB input and DOCUMENT option	NO	
X068	XMLSerialize: BOM	NO	
X069	XMLSerialize: INDENT	NO	
X070	XMLSerialize: Character string serialization and CONTENT option	NO	
X071	XMLSerialize: Character string serialization and DOCUMENT option	NO	
X072	XMLSerialize: Character string serialization	NO	
X073	XMLSerialize: BLOB serialization and CONTENT option	NO	
X074	XMLSerialize: BLOB serialization and DOCUMENT option	NO	
X075	XMLSerialize: BLOB serialization	NO	
X076	XMLSerialize: VERSION	NO	
X077	XMLSerialize: explicit ENCODING option	NO	
X078	XMLSerialize: explicit XML declaration	NO	
X080	Namespaces in XML publishing	NO	
X081	Query-level XML namespace declarations	NO	
X082	XML namespace declarations in DML	NO	
X083	XML namespace declarations in DDL	NO	
X084	XML namespace declarations in compound statements	NO	
X085	Predefined namespace prefixes	NO	
X086	XML namespace declarations in XMLTable	NO	
X090	XML document predicate	NO	
X091	XML content predicate	NO	
X096	XMLExists	NO	
X100	Host language support for XML: CONTENT option	NO	
X101	Host language support for XML: DOCUMENT option	NO	
X110	Host language support for XML: VARCHAR mapping	NO	
X111	Host language support for XML: CLOB mapping	NO	
X112	Host language support for XML: BLOB mapping	NO	
X113	Host language support for XML: STRIP WHITESPACE option	NO	

Table J.1 SQL 2008 Optional Feature Compliance Details

ID	Feature	Supported	Comments
X114	Host language support for XML: PRESERVE WHITESPACE option	NO	
X120	XML parameters in SQL routines	NO	
X121	XML parameters in external routines	NO	
X131	Query-level XMLBINARY clause	NO	
X132	XMLBINARY clause in DML	NO	
X133	XMLBINARY clause in DDL	NO	
X134	XMLBINARY clause in compound statements	NO	
X135	XMLBINARY clause in subqueries	NO	
X141	IS VALID predicate: data-driven case	NO	
X142	IS VALID predicate: ACCORDING TO clause	NO	
X143	IS VALID predicate: ELEMENT clause	NO	
X144	IS VALID predicate: schema location	NO	
X145	IS VALID predicate outside check constraints	NO	
X151	IS VALID predicate with DOCUMENT option	NO	
X152	IS VALID predicate with CONTENT option	NO	
X153	IS VALID predicate with SEQUENCE option	NO	
X155	IS VALID predicate: NAMESPACE without ELEMENT clause	NO	
X157	IS VALID predicate: NO NAMESPACE with ELEMENT clause	NO	
X160	Basic Information Schema for registered XML Schemas	NO	
X161	Advanced Information Schema for registered XML Schemas	NO	
X170	XML null handling options	NO	
X171	NIL ON NO CONTENT option	NO	
X181	XML(DOCUMENT(UNTYPED)) type	NO	
X182	XML(DOCUMENT(ANY)) type	NO	
X190	XML(SEQUENCE) type	NO	
X191	XML(DOCUMENT(XMLSCHEMA)) type	NO	
X192	XML(CONTENT(XMLSCHEMA)) type	NO	
X200	XMLQuery	NO	
X201	XMLQuery: RETURNING CONTENT	NO	
X202	XMLQuery: RETURNING SEQUENCE	NO	
X203	XMLQuery: passing a context item	NO	

Table J.1 SQL 2008 Optional Feature Compliance Details

ID	Feature	Supported	Comments
X204	XMLQuery: initializing an XQuery variable	NO	
X205	XMLQuery: EMPTY ON EMPTY option	NO	
X206	XMLQuery: NULL ON EMPTY option	NO	
X211	XML 1.1 support	NO	
X221	XML passing mechanism BY VALUE	NO	
X222	XML passing mechanism BY REF	NO	
X231	XML(CONTENT(UNTYPED)) type	NO	
X232	XML(CONTENT(ANY)) type	NO	
X241	RETURNING CONTENT in XML publishing	NO	
X242	RETURNING SEQUENCE in XML publishing	NO	
X251	Persistent XML values of XML(DOCUMENT(UNTYPED)) type	NO	
X252	Persistent XML values of XML(DOCUMENT(ANY)) type	NO	
X253	Persistent XML values of XML(CONTENT(UNTYPED)) type	NO	
X254	Persistent XML values of XML(CONTENT(ANY)) type	NO	
X255	Persistent XML values of XML(SEQUENCE) type	NO	
X256	Persistent XML values of XML(DOCUMENT(XMLSCHEMA)) type	NO	
X257	Persistent XML values of XML(CONTENT(XMLSCHEMA)) type	NO	
X260	XML type: ELEMENT clause	NO	
X261	XML type: NAMESPACE without ELEMENT clause	NO	
X263	XML type: NO NAMESPACE with ELEMENT clause	NO	
X264	XML type: schema location	NO	
X271	XMLValidate: data-driven case	NO	
X272	XMLValidate: ACCORDING TO clause	NO	
X273	XMLValidate: ELEMENT clause	NO	
X274	XMLValidate: schema location	NO	
X281	XMLValidate: with DOCUMENT option	NO	
X282	XMLValidate with CONTENT option	NO	
X283	XMLValidate with SEQUENCE option	NO	
X284	XMLValidate NAMESPACE without ELEMENT clause	NO	
X286	XMLValidate: NO NAMESPACE with ELEMENT clause	NO	

Table J.1 SQL 2008 Optional Feature Compliance Details

ID	Feature	Supported	Comments
X300	XMLTable	NO	
X301	XMLTable: derived column list option	NO	
X302	XMLTable: ordinality column option	NO	
X303	XMLTable: column default option	NO	
X304	XMLTable: passing a context item	NO	
X305	XMLTable: initializing an XQuery variable	NO	
X400	Name and identifier mapping	NO	

Glossary

A

append-only tables

An append-only (AO) table is a storage representation that allows only appending new rows to a table, but does not allow updating or deleting existing rows. This allows for more compact storage on disk because each row does not need to store the [MVCC](#) transaction visibility info. This saves 20 bytes per row. AO tables can also be compressed.

array

The set of physical devices (hosts, servers, network switches, etc.) used to house a [Greenplum Database system](#).

B

bandwidth

Bandwidth is the maximum amount of information that can be transmitted along a channel, such as a network or I/O channel. This data transfer rate is usually measured in megabytes per second (MB/s).

C

catalog

See [system catalog](#).

column-oriented table

Greenplum provides a choice of storage orientation models for a table: row or column. A column-oriented table stores its content on disk by column rather than by row. This storage model has performance advantages for certain types of queries. Only [append-only tables](#) can be column-oriented; [heap tables](#) are always row-oriented.

correlated subquery

A correlated subquery is a nested SELECT statement that refers to a column from an outer SELECT statement. For example:

```
SELECT * FROM product WHERE exists (SELECT * FROM sale WHERE
qty>0 AND pn = product.pn);
```

D

data directory

The data directory is the file system location on disk where database data is stored. The **master** data directory contains the global **system catalog** only — no user data is stored on the master. The data directory on the **segment instances** has user data for that segment plus a local copy of the system catalog. The data directory contains several subdirectories, control files, and configuration files as well.

distributed

Certain database objects in Greenplum Database, such as tables and indexes, are distributed. They are divided into equal parts and spread out among the **segment instances** based on a hashing algorithm. To the end-user and client software, however, a distributed object appears as a conventional database object.

distribution key

In a Greenplum table that uses **hash distribution**, one or more columns are used as the distribution key, meaning those columns are used to divide the data among all of the **segments**. The distribution key should be the primary key of the table or a unique column or set of columns.

distribution policy

The distribution policy determines how to divide the rows of a table among the Greenplum **segments**. Greenplum Database provides two types of distribution policy: **hash distribution** and **random distribution**.

DDL

Data Definition Language. A subset of SQL commands used for defining the structure of a database.

DML

Database Manipulation Language. SQL commands that store, manipulate, and retrieve data from tables. INSERT, UPDATE, DELETE, and SELECT are DML commands.

G

gang

For each **slice** of the query plan there is at least one **query executor** worker process assigned. During query execution, each segment will have a number of processes working on the query in parallel. Related processes that are working on the same portion of the query plan on different segments are referred to as **gangs**.

Greenplum Database

Greenplum Database is the industry's first massively parallel processing (MPP) database server based on open-source technology. It is explicitly designed to support business intelligence (BI) applications and large, multi-terabyte data warehouses. Greenplum Database is based on PostgreSQL.

Greenplum Database system

An associated set of [segment instances](#) and a [master instance](#) running on an [array](#), which can be composed of one or more [hosts](#).

Greenplum instance

The process that serves a database. An instance of Greenplum Database is comprised of a [master instance](#) and two or more [segment instances](#), however users and administrators always connect to the database via the [master instance](#).

GUC

GUC is a PostgreSQL acronym that stands for *global user configuration*, and is an obsolete term in Greenplum Database. However, the term *GUC* may still occasionally appear in Greenplum Database error messages and other materials. A GUC is equivalent to a *global* server configuration parameter. See also, postgresql.conf.

H

hash distribution

With hash distribution, one or more table columns is used as the [distribution key](#) for the table. The distribution key is used by a hashing algorithm to assign each row to a particular [segment](#). Keys of the same value will always hash to the same segment.

heap tables

Whenever you create a table without specifying a storage structure, the default is a heap storage structure. In a heap structure, the table is an unordered collection of data that allows multiple copies or *versions* of a row. Heap tables have row-level versioning information and allow updates and deletes. See also [append-only tables](#) and [multiversion concurrency control](#).

host

A host represents a physical machine or compute node in a Greenplum Database system. In Greenplum Database, one host is designated as the [master](#). The other hosts in the system have one or more [segments](#) on them.

I

interconnect

The interconnect is the networking layer of Greenplum Database. When a user connects to a database and issues a query, processes are created on each of the [segments](#) to handle the work of that query. The interconnect refers to the inter-process communication between the segments and [master](#), as well as the network infrastructure on which this communication relies. The interconnect typically uses a standard Gigabit Ethernet switching fabric.

I/O

Input/Output (I/O) refers to the transfer of data to and from a system or device using a communication channel.

J

JDBC

Java Database Connectivity is an application program interface (API) specification for connecting programs written in Java to data in a database management system (DBMS). The application program interface lets you encode access request statements in SQL that are then passed to the program that manages the database.

M

master

The master is the entry point to a Greenplum Database system. It is the database listener process ([postmaster](#)) that accepts client connections and dispatches the SQL commands issued by the users of the system.

The master is where the global [system catalog](#) resides. However, the master does not contain any user data. User data resides only on the [segments](#). The master does the work of authenticating user connections, parsing and planning the incoming SQL commands, distributing the query plan to the segments for execution, coordinating the results returned by each of the segments, and presenting the final results to the user.

master instance

The database process that serves the Greenplum master. See [master](#).

mirror

A mirror is a backup copy of a [segment](#) (or master) that is stored on a different host than the primary copy. Mirrors are useful for maintaining operations if a host in your Greenplum Database system fails. Mirroring is an optional feature of Greenplum Database. Mirror segments are evenly distributed among other hosts in the array. If a host that holds a primary segment fails, Greenplum Database will switch to the mirror or secondary host.

motion node

A motion node is a portion of a query execution plan that indicates data movement between the various database instances of Greenplum Database (segments and the master). Some operations, such as joins, require [segments](#) to send and receive tuples to one another in order to satisfy the operation. A motion node can also indicate data movement from the segments back up to the [master](#).

MPP

Massive Parallel Processing.

multiversion concurrency control

Unlike traditional database systems which use locks for concurrency control, Greenplum Database (as does PostgreSQL) maintains data consistency by using a multiversion model (multiversion concurrency control or MVCC). This means that while querying a database, each transaction sees a snapshot of data which protects the transaction from viewing inconsistent data that could be caused by (other) concurrent updates on the same data rows. This provides transaction isolation for each database session.

MVCC, by eschewing explicit locking methodologies of traditional database systems, minimizes lock contention in order to allow for reasonable performance in multiuser environments. The main advantage to using the MVCC model of concurrency control rather than locking is that in MVCC locks acquired for querying (reading) data do not conflict with locks acquired for writing data, and so reading never blocks writing and writing never blocks reading.

MVCC

See [multiversion concurrency control](#).

O**ODBC**

Open Database Connectivity, a standard database access method that makes it possible to access any data from any client application, regardless of which database management system (DBMS) is handling the data. ODBC manages this by inserting a middle layer, called a database driver, between a client application and the DBMS. The purpose of this layer is to translate the application's data queries into commands that the DBMS understands.

OLAP

Online Analytical Processing (OLAP) is a category of technologies for collecting, managing, processing and presenting multidimensional data for analysis and management. OLAP leverages existing data from a relational schema or data warehouse (data source) by placing key performance indicators (measures) into context (dimensions). As of release 3.1, OLAP functions are supported in Greenplum Database. In practice, OLAP functions allow application developers to compose analytic business queries more easily and more efficiently. For example, moving averages and moving sums can be calculated over various intervals; aggregations and ranks can be reset as selected column values change; and complex ratios can be expressed in simple terms.

P

partitioned

Partitioning is a way to logically divide the data in a table for better performance and easier maintenance. In Greenplum Database, partitioning is a procedure that creates multiple sub-tables (or child tables) from a single large table (or parent table). The primary purpose is to improve performance by scanning only the relevant data needed to satisfy a query. Note that partitioned tables are also [distributed](#).

Perl DBI

Perl Database Interface (DBI) is an API for connecting programs written in Perl to database management systems (DBMS). Perl DBI (DataBase Interface) is the most common database interface for the Perl programming language.

PostgreSQL

PostgreSQL is a SQL compliant, open source relational database management system (RDBMS). Greenplum Database uses a modified version of PostgreSQL as its underlying database server. For more information on PostgreSQL go to <http://www.postgresql.org>.

postgresql.conf

The server configuration file that configures various aspects of the database server. This configuration file is located in the [data directory](#) of the database instance. In Greenplum Database, the [master](#) and each [segment](#) instance has its own `postgresql.conf` file.

postgres process

The `postgres` executable is the actual [PostgreSQL](#) server process that processes queries. The database listener `postgres` process (also known as the [postmaster](#)) creates other `postgres` subprocesses as needed to handle client connections.

postmaster

The `postmaster` server program starts the `postgres` database server listener process that accepts client connections. In Greenplum Database, a `postgres` database listener process runs on the Greenplum master instance and on each segment instance.

psql

This is the interactive terminal to [PostgreSQL](#) and Greenplum Database. You can use psql to access a database and issue SQL commands. For more information on psql, see “psql” on page 730.

Q**QD**

See [query dispatcher](#).

QE

See [query executor](#).

query dispatcher

The query dispatcher (QD) is a process that is initiated when users connect to the [master](#) and issue SQL commands. This process represents a user session and is responsible for sending the query plan to the segments and coordinating the results it gets back. The query dispatcher process spawns one or more [query executor](#) processes to assist in the execution of SQL commands.

query executor

A query executor process (QE) is associated with a query dispatcher (QD) process and operates on its behalf. Query executor processes run on the [segment instances](#) and execute their slice of the query plan on a [segment](#).

query plan

A query plan is the set of operations that Greenplum Database will perform to produce the answer to a given query. Each node or step in the plan represents a database operation such as a table scan, join, aggregation or sort. Plans are read and executed from bottom to top. Greenplum Database supports an additional plan node type called a [motion node](#). See also [slice](#).

R**rack**

A type of shelving to which computer components can be attached vertically, one on top of the other. Components are normally screwed into front-mounted, tapped metal strips with holes which are spaced so as to accommodate the height of devices of various U-sizes. Racks usually have their height denominated in U-units.

RAID

Redundant Array of Independent (or Inexpensive) Disks. RAID is a system of using multiple hard drives for sharing or replicating data among the drives. The benefit of RAID is increased data integrity, fault-tolerance and/or performance. Multiple hard drives are grouped and seen by the OS as one logical hard drive.

RAM

Random Access Memory. The main memory of a computer system used for storing programs and data. RAM provides temporary read/write storage while hard disks offer semi-permanent storage.

random distribution

With random distribution, table rows are sent to the [segments](#) as they come in, cycling across the segments in a round-robin fashion. Rows with columns having the same values will not necessarily be located on the same segment. Although a random distribution ensures even data distribution, there are performance advantages to choosing a [hash distribution](#) policy whenever possible.

S**segment**

A segment represents a portion of data in a Greenplum database. User-defined tables and their indexes are [distributed](#) across the available number of [segment instances](#) in the Greenplum Database system. Each segment instance contains a distinct portion of the user data. A primary segment instance and its [mirror](#) both store the same segment of data.

segment instance

The segment instance is the database server process ([postmaster](#)) that serves [segments](#). Users do not connect to segment instances directly, but through the [master](#).

slice

In order to achieve maximum parallelism during query execution, Greenplum divides the work of the [query plan](#) into *slices*. A slice is a portion of the plan that can be worked on independently at the segment level. A query plan is sliced wherever a [motion node](#) occurs in the plan, one slice on each side of the motion. Plans that do not require data movement (such as catalog lookups on the master) are known as single-slice plans.

star schema

A relational database schema often used in data warehousing. The star schema is organized around a central table (fact table) joined to a few smaller tables (dimension tables) using foreign key references. The fact table contains raw numeric items that represent relevant business facts (price, number of units sold, etc.).

system catalog

The system catalogs are the place where a relational database management system stores schema metadata, such as information about tables and columns, and internal bookkeeping information. The system catalog in Greenplum Database is the same as the PostgreSQL catalog with some additional tables to support the distributed nature of the Greenplum system and databases. In Greenplum Database, the [master](#) contains the global system catalog tables. The [segments](#) also maintain their own local copy of the system catalog.

T**TPC-H**

The Transaction Processing Performance Council (TPC) is a third-party organization that provides database benchmark tools for the industry. **TPC-H** is their ad-hoc, decision support benchmark. This benchmark illustrates decision support systems that examine large volumes of data, execute queries with a high degree of complexity, and give answers to critical business questions. The TPC-H toolkit is used for Greenplum Database functional and performance testing.

tuple

A tuple is another name for a row or record in a relational database table.

W**WAL**

Write-Ahead Logging (WAL) is a standard approach to transaction logging. WAL's central concept is that changes to data files (where tables and indexes reside) are logged before they are written to permanent storage. Data pages do not need to be flushed to disk on every transaction commit. In the event of a crash, data changes not yet applied to the database can be recovered from the log. A major benefit of using WAL is a significantly reduced number of disk writes.

Index

Symbols

.pgpass: 79

A

ABORT: 286
 access and permissions: 67
 active statement
 cost: 85
 count: 85
 active threshold: 88
 add_missing_from: 756
 administration utilities: 10
 aggregate functions: 142
 user-defined: 350
 ALTER AGGREGATE: 287
 ALTER CONVERSION: 289
 ALTER DATABASE: 290
 ALTER DOMAIN: 292
 ALTER FUNCTION: 294
 ALTER GROUP: 297
 ALTER INDEX: 298
 ALTER LANGUAGE: 300
 ALTER OPERATOR: 301
 ALTER OPERATOR CLASS: 302
 ALTER RESOURCE QUEUE: 303
 ALTER ROLE: 305
 ALTER SCHEMA: 308
 ALTER SEQUENCE: 309
 ALTER TABLE: 312
 ALTER TABLESPACE: 324
 ALTER TRIGGER: 325
 ALTER TYPE: 326
 ALTER USER: 327
 ANALYZE: 123, 227, 328
 APIs: 82
 append-only tables: 101
 parameters: 180, 181
 architecture
 Greenplum Performance Monitor: 11
 Greenplum system: 6
 network: 33
 archive host backup: 202
 array
 configuration parameters: 181
 functions: 142
 in Greenplum Database: 916
 operators: 142
 ARRAY_NAME: 788
 array_nulls: 756
 Ascential Datastage: 83
 attributes
 resource queue attributes: 85
 role attributes: 68
 attrnums: 826
 authentication parameters: 173
 authentication_timeout: 756

automatic database backup: 199
 automatic vacuuming parameters: 179
 autovacuum: 756
 autovacuum_analyze_scale_factor: 756
 autovacuum_analyze_threshold: 756
 autovacuum_freeze_max_age: 756
 autovacuum_naptime: 757
 autovacuum_vacuum_cost_delay: 757
 autovacuum_vacuum_cost_limit: 757
 autovacuum_vacuum_scale_factor: 757
 autovacuum_vacuum_threshold: 757

B

backout_gpinitssystem: 60
 backslash_quote: 757
 backup
 database: 195, 197
 non-parallel: 196
 parallel: 195
 backup master: 184
 configuring: 187
 installing: 40
 bashrc: 43
 BEGIN: 330
 bgwriter_all_maxpages: 757
 bgwriter_all_percent: 758
 bgwriter_delay: 758
 bgwriter_lru_maxpages: 758
 bgwriter_lru_percent: 758
 bigint: 814
 bigserial: 814
 binary functions: 142
 binary operators: 142
 bit: 814
 bit string functions: 142
 bit string operators: 142
 bit varying: 814
 bitmap: 121
 bitmap index: 97
 bitmap indexes: 119, 121
 block_size: 758
 bonjour_name: 758
 boolean: 814
 box: 814
 Business Objects: 83
 bytea: 814

C

calculate storage capacity: 36
 cast, user-defined: 354
 catalog: 916
 char: 814
 character: 814
 character sets: 52, 630
 character varying: 814
 check constraints: 99, 107

- check_function_bodies: 758
- CHECK_POINT_SEGMENTS: 789
- checking for locks: 238
- CHECKPOINT: 332
- checkpoint_segments: 758
- checkpoint_timeout: 758
- checkpoint_warning: 759
- cidr: 814
- circle: 815
- client: 77
 - connection parameters: 180
 - connections: 77
 - interfaces: 82
 - third-party: 83
 - troubleshooting: 84
- client applications: 77
- client tools
 - clusterdb: 694
 - createdb: 696
 - createlang: 698
 - createuser: 718
 - dropdb: 700
 - droplang: 702
 - dropuser: 704
 - ecpg: 706
 - pg_config: 708
 - pg_dump: 711
 - pg_dumpall: 721
 - pg_restore: 725
 - psql: 730
 - reindexdb: 751
 - vacuumdb: 753
- client_encoding: 759
- client_min_messages: 759
- CLOSE: 333
- CLUSTER: 334
- clusterdb: 694
- Cognos: 83
- collecting statistics: 328
- column constraints: 99
- column-oriented tables: 101
- columns
 - data types: 98
 - storage by: 101
- commands
 - clusterdb: 694
 - createdb: 696
 - createlang: 698
 - createuser: 718
 - dropdb: 700
 - droplang: 702
 - dropuser: 704
 - ecpg: 706
 - gp_dump: 583
 - gp_restore: 588
 - gpactivatestandby: 592
 - gpaddmirrors: 595
 - gpchecknet: 599
 - gpcheckos: 602
 - gpcheckperf: 604
 - gpcrondump: 607
 - gpdrestore: 612
 - gpdeletesystem: 615
 - gpdetective: 617
 - gpexpandsystem: 619
 - gpfdist: 623
 - gpinitstandby: 625
 - gpinitssystem: 628
 - gpload: 633
 - gplogfilter: 643
 - gmapreduce: 647
 - gprebuildsystem: 654
 - gprecoverseg: 657
 - gpscp: 661
 - gpsizecalc: 663
 - gpskew: 666
 - gpssh: 669
 - gpssh-exkeys: 671
 - gpstart: 674
 - gpstate: 676
 - gpstop: 679
 - gpsyncmaster: 564
 - initdb: 564
 - ipcclean: 564
 - pg_config: 708
 - pg_controldata: 564
 - pg_ctl: 564
 - pg_dump: 711
 - pg_dumpall: 721
 - pg_resetxlog: 564
 - pg_restore: 725
 - postgres: 564
 - postmaster: 564
 - psql: 730
 - reindexdb: 751
 - vacuumdb: 753
- COMMENT: 337
- COMMIT: 340
- commit_delay: 759
- commit_siblings: 759
- comparison operators: 142
- compressed tables: 101
- concurrency control: 125
 - limiting concurrent connections: 75
 - lock management parameters: 180
 - lock modes: 125
 - resource queues: 85
- conditional expressions: 142
- config_file: 759
- configuration: 180
 - array configuration parameters: 181
 - array setup: 788
 - connection parameters: 77
 - environment variables: 77
 - global parameters: 171
 - Greenplum Database server: 755
 - Greenplum system: 170

- hardware: 30
 - hardware example: 31
 - Linux parameter settings: 35
 - local parameters: 171
 - master parameters: 171
 - network layout: 32
 - parameter categories: 172
 - redundant network switch: 33
 - required OS system settings: 34
 - Solaris parameter settings: 35
 - viewing mirror configuration: 222
 - viewing server settings: 222
 - workload management: 87
 - configuration parameters
 - initialization: 788
 - server: 755
 - configuring
 - environment variables: 43
 - fault operational mode: 188
 - Greenplum Database: 59
 - master host: 42
 - mirroring: 186
 - standby master: 187
 - connecting: 77
 - connection parameters: 77, 173
 - database: 73, 77
 - ETL and BI tools: 83
 - limiting concurrent connections: 75
 - troubleshooting: 84
 - with psql: 79
 - constraint_exclusion: 760
 - constraints
 - check: 99, 107
 - not-null constraints: 99
 - unique constraints: 99
 - content: 820, 831
 - contention: 235, 238
 - continue mode: 188
 - conversions: 357
 - COPY: 341
 - non-parallel data loading: 163
 - correlated subquery: 916
 - supported syntax: 536
 - cost threshold: 88
 - cpu_index_tuple_cost: 760
 - cpu_operator_cost: 760
 - cpu_tuple_cost: 760
 - CREATE AGGREGATE: 350
 - CREATE CAST: 354
 - CREATE CONVERSION: 357
 - CREATE DATABASE: 359
 - CREATE DOMAIN: 361
 - CREATE EXTERNAL TABLE: 159, 363
 - CREATE EXTERNAL WEB TABLE: 363
 - CREATE FUNCTION: 375
 - CREATE GROUP: 381
 - CREATE INDEX: 382
 - CREATE LANGUAGE: 386
 - CREATE OPERATOR: 389
 - CREATE OPERATOR CLASS: 394
 - CREATE RESOURCE QUEUE: 399
 - CREATE ROLE: 402
 - CREATE RULE: 406
 - CREATE SCHEMA: 409
 - CREATE SEQUENCE: 411
 - CREATE TABLE: 415
 - CREATE TABLE AS: 426
 - CREATE TABLESPACE: 430
 - CREATE TRIGGER: 433
 - CREATE TYPE: 436
 - CREATE USER: 443
 - CREATE VIEW: 444
 - createdb: 696
 - createlang: 698
 - createuser: 718
 - creating
 - databases: 359
 - functions: 375
 - indexes: 382
 - roles: 402
 - rules: 406
 - schemas: 409
 - sequences: 411
 - tables: 415
 - cron: 607, 612
 - cursor_tuple_fraction: 760
 - cursors
 - closing: 333
 - fetching data from: 488
 - moving position: 504
 - opening: 448
 - custom_variable_classes: 760
 - customer support: 240
- ## D
- data
 - distribution: 236
 - redundancy: 183
 - data directory: 917
 - master: 43
 - segments: 47
 - data loading: 341, 363, 633
 - error isolation mode: 159, 160
 - from external tables: 160
 - from web tables: 163
 - non-parallel: 163
 - parallel: 10, 153
 - performance tips: 164
 - with COPY: 163
 - with gpfdist: 10
 - data types
 - bigint: 814
 - bigserial: 814
 - bit: 814
 - bit varying: 814
 - boolean: 814
 - box: 814
 - bytea: 814

- char: 814
- character: 814
- character varying: 814
- choosing: 98
- cidr: 814
- circle: 815
- date: 815
- decimal: 815
- double precision: 815
- float4: 815
- float8: 815
- formatting: 142
- inet: 815
- int, int4: 815
- int2: 815
- int8: 814
- integer: 815
- interval: 815
- line: 815
- lseg: 815
- macaddr: 815
- money: 815
- numeric: 815
- path: 815
- point: 815
- polygon: 815
- real: 815
- serial: 815
- serial4: 815
- serial8: 814
- smallint: 815
- text: 815
- time: 815
- timestamp: 816
- timestampz: 816
- timetz: 815
- user-defined: 436
- varbit: 814
- varchar: 814
- DATA_DIRECTORY: 788
- data_directory: 760
- database
 - access and permissions: 67
 - administration: 94
 - analyze: 227
 - backup: 195, 197
 - backup automatically: 196, 199
 - client
 - applications: 77
 - connections: 73, 77
 - creating: 359
 - design, optimizing: 236
 - log files: 227, 229
 - maintenance: 107
 - management: 94
 - migration: 195
 - rebuild: 203
 - reindex: 227
 - restore: 195, 196, 200
 - restore from archive host: 202
 - schema: 96
 - statistics: 235
 - templates: 94
 - vacuum: 227
- DATABASE_NAME: 790
- datadir: 820
- Datastage: 83
- date: 815
- date functions: 142
- date operators: 142
- date range partitioning: 109
- DateStyle: 760
- db_user_namespace: 761
- dbid: 820, 821, 831
- DDL: 917
- deadlock_timeout: 761
- DEALLOCATE: 447
- debug_assertions: 761
- debug_pretty_print: 761
- debug_print_parse: 761
- debug_print_plan: 761
- debug_print_prelim_plan: 761
- debug_print_rewritten: 761
- debug_print_slice_table: 761
- debugging: 240
- decimal: 815
- DECLARE: 448
- default_statistics_target: 762
- default_tablespace: 762
- default_transaction_isolation: 762
- default_transaction_read_only: 762
- default_with_oids: 762
- definedprimary: 820
- DELETE: 451
- delimiter character: 155
 - designated: 155
- demos
 - removing: 65
- directories
 - data directory: 760, 788
 - docs: 41
- disk
 - capacity: 36
 - failure: 234
 - layout: 31
 - usage: 222, 234
- dispatcher: 922
- distributed: 917
 - database system architecture: 12
 - table skew: 223
 - tables: 12, 98
 - tables, altering: 105
 - tables, distribution policy: 100
 - tables, optimizing: 236
- distribution key: 105, 223
- distribution policy: 13
 - hash: 13

- random: 13
- DML: 917
 - INSERT command: 126
 - TRUNCATE command: 128
 - UPDATE command: 127
- docs: 41
- document
 - MapReduce: 791
- domains: 361
- double precision: 815
- DROP AGGREGATE: 454
- DROP CAST: 455
- DROP CONVERSION: 456
- DROP DATABASE: 457
- DROP DOMAIN: 458
- DROP EXTERNAL TABLE: 459
- DROP FUNCTION: 460
- DROP GROUP: 462
- DROP INDEX: 463
- DROP LANGUAGE: 464
- DROP OPERATOR: 465
- DROP OPERATOR CLASS: 467
- DROP OWNED: 469
- DROP RESOURCE QUEUE: 471
- DROP ROLE: 473
- DROP RULE: 474
- DROP SCHEMA: 475
- DROP SEQUENCE: 476
- DROP TABLE: 477
- DROP TABLESPACE: 478
- DROP TRIGGER: 479
- DROP TYPE: 480
- DROP USER: 481
- DROP VIEW: 482
- dropdb: 700
- droplang: 702
- dropuser: 704
- dynamic_library_path: 762

E

- ecpg: 706
- effective_cache_size: 762
- enable_bitmapscan: 763
- enable_groupagg: 763
- enable_hashagg: 763
- enable_hashjoin: 763
- enable_indexscan: 763
- enable_mergejoin: 763
- enable_nestloop: 763
- enable_seqscan: 763
- enable_sort: 764
- enable_tidscan: 764
- ENCODING: 789
- encoding: 52, 630
- encoding conversions: 357
- END: 483
- environment variables: 43, 77, 812, 814
- error isolation mode: 159, 160
 - with COPY: 163
- error reporting and logging parameters: 178
- errors: 189

- error messages: 239
- escape reserved characters: 155
- escape_string_warning: 764
- etc/hosts: 84
- ETL
 - with gpfdist: 156
- EXECUTE: 484
- executor: 922
- EXPLAIN: 485
- EXPLAIN ANALYZE: 235
- explain_pretty_print: 764
- external tables: 153, 363
 - accessing with gpfdist: 154
 - defining: 159
 - error isolation mode: 159, 160
 - format: 155
 - parameters: 180
- external_pid_file: 764
- extra_float_digits: 764

F

- failover: 8
- failures: 185
 - diagnosing: 189
 - of master: 193
 - of segments: 190
 - recovering from: 190
- fault operational mode: 188
- faults: 185
- FETCH: 488
- file location parameters: 173
- file server, gpfdist: 156
- float4: 815
- float8: 815
- foreign keys: 100
- from_collapse_limit: 764
- fsynch: 764
- full_page_writes: 765
- functions
 - aggregate: 142
 - array: 142
 - binary: 142
 - bit string: 142
 - data type formatting: 142
 - date: 142
 - geometric: 142
 - MapReduce: 791
 - mathematical: 142
 - network address: 142
 - sequence: 142
 - set returning: 142
 - string: 142
 - system administration: 143
 - system information: 143
 - time: 142
 - user defined: 141
 - user-defined: 375

G

- gangs: 28
- gather motion: 26, 27
- geometric functions: 142
- geometric operators: 142
- gin_fuzzy_search_limit: 765
- global parameters: 170
 - setting: 171
- global system catalog: 7, 923
- gp_adjust_selectivity_for_outerjoins: 765
- gp_analyze_relative_error: 765
- gp_autostats_mode: 766
- gp_autostats_on_change_threshold: 766
- gp_cached_segworkers_threshold: 766
- gp_command_count: 766
- gp_configuration: 820, 821, 822, 823, 827, 828, 830
- gp_connections_per_thread: 767
- gp_crondump: 196
- gp_debug_linger: 767
- gp_distributed_log: 824
- gp_distributed_xacts: 825
- gp_distribution_policy: 826
- gp_dump: 195, 197, 201, 583
- gp_enable_adaptive_nestloop: 767
- gp_enable_agg_distinct: 767
- gp_enable_agg_distinct_pruning: 767
- gp_enable_fallback_plan: 767
- gp_enable_fast_sri: 767
- gp_enable_gpperfmon: 767
- gp_enable_multiphase_agg: 768
- gp_enable_predicate_propagation: 768
- gp_enable_preunique: 768
- gp_enable_sequential_window_plans: 768
- gp_enable_sort_distinct: 768
- gp_enable_sort_limit: 769
- gp_external_enable_exec: 769
- gp_external_grant_privileges: 769
- gp_external_max_segs: 769
- gp_fault_action: 769
- gp_fts_probe_threadcount: 769
- gp_gpperfmon_send_interval: 769
- gp_hashagg_compress_spill_files: 770
- gp_hashjoin_tuples_per_bucket: 770
- gp_id: 831
- gp_init_config_example: 59
- gp_interconnect_hash_multiplier: 770
- gp_interconnect_queue_depth: 770
- gp_interconnect_setup_timeout: 770
- gp_interconnect_type: 770
- gp_log_format: 771
- gp_log_gang: 771
- gp_log_interconnect: 771
- gp_max_csv_line_length: 771
- gp_max_local_distributed_cache: 771
- gp_max_packet_size: 771
- gp_motion_cost_per_row: 771
- gp_reject_percent_threshold: 771
- gp_reraise_signal: 772
- gp_restore: 200, 588
- gp_role: 772
- gp_safefswritesize: 772
- gp_segment_connect_timeout: 772
- gp_segments_for_planner: 772
- gp_session_id: 772
- gp_set_proc_affinity: 772
- gp_set_read_only: 772
- gp_statistics_pullup_from_child_partition: 772
- gp_statistics_use_fkeys: 773
- gp_transaction_log: 833
- gp_use_dispatch_agent: 773
- gp_version_at_initdb: 835
- gp_vmem_protect_gang_cache_limit: 773
- gp_vmem_protect_limit: 773
- gpactivatestandby: 592
- gpaddmirrors: 595
- gpadmin user: 42, 67, 73
- gpchecknet: 238, 599
- gpchecknet utility: 39
- gpcheckos: 602
- gpcheckperf: 238, 604
- gpcheckperf utility: 37
- gpcrondump: 197, 607
- gpdrestore: 197, 201, 612
- gpdeletesystem: 615
- gpdemo: 62
- gpdetective: 240, 617
- gpexpandsystem: 619
- gpfdist: 10, 154, 623
 - installing: 158
 - troubleshooting: 158
 - using: 156
- GPHOME: 812
- gpinitstandby: 625
- gpinitssystem: 628
 - troubleshooting: 60
- gpload: 633
- gplogfilter: 643
- gpmapreduce: 647
- gpname: 831
- gpperfmon_port: 773
- gpbuildsystem: 654
- gprecoverseg: 657
- gpscp: 661
- gpsizecalc: 222, 663
- gpskew: 223, 666
- gpssh: 669
- gpssh-exkeys: 671
- gpstart: 229, 674
- gpstate: 221, 238, 676
- gpstop: 679
- gpsyncmaster: 564
- GRANT: 492
- Greenplum Database: 57
 - array: 916
 - connecting to: 77
 - demo programs: 62
 - environment: 43
 - installing: 40
 - starting: 167
 - stopping: 168
 - super user account: 42
- greenplum_path.sh: 43
- groups

creating: 69
gzip compression: 101

H

hardware
 disk layout: 31
 performance validation: 37
 platforms: 30
 platforms example: 31
hardware issues: 234
hash distribution: 13
hash key: 12
hba_file: 773
heap tables: 101
high availability: 9
historical data: 109
host: 918
host failure: 234
host_file: 788

I

ident_file: 773
indexes: 121, 382
 bitmap: 119
 examining usage: 122
 managing: 123
 operator classes: 394
 types: 121
 using with Greenplum: 119
inet: 815
Informatica: 83
inheritance: 107
initdb: 564
initializing: 57
INSERT command: 126, 497
installer: 41
 files: 41
installing
 backup master host: 40
 Greenplum Database: 40
 master host: 40
 segment hosts: 44
int: 815
int2: 815
int4: 815
int8: 814
integer: 815
integer_datetimes: 774
interconnect: 12
 about: 7
 redundancy: 9
interface
 database: 77
 network: 33
interval: 815
IntervalStyle: 774
ipcclean: 564
isprimary: 820

J

JDBC: 77, 82
jetpack: 226
join order: 236
join_collapse_limit: 774

K

key: 109, 110
key exchange: 44
krb_caseins_users: 774
krb_server_hostname: 774
krb_server_keyfile: 774
krb_srvname: 774

L

lc_collate: 775
lc_ctype: 775
lc_messages: 775
lc_monetary: 775
lc_numeric: 775
lc_time: 775
LD_LIBRARY_PATH: 812
line: 815
list partitioning: 108
listen_addresses: 775
loader: 633
loading data: 159, 341, 363
 error isolation mode: 159, 160
 from external tables: 153, 160
 from web tables: 163
 non-parallel: 163
 parallel: 10, 153
 performance tips: 164
 with COPY: 163
 with gpfdist: 10, 156
loading partitioned tables: 112
local parameters: 170
 setting: 171
local_preload_libraries: 776
locale: 630
locales: 50
localization: 50
localoid: 826
LOCK: 500
lock: 238
 management parameters: 180
 modes: 125
 row-level: 235
 table-level: 235
log files
 error messages: 239
 format of: 224
 managing: 229
 viewing: 224
log rotation: 229
log_autostats: 776
log_connections: 776
log_disconnections: 776

log_dispatch_stats: 776
 log_duration: 776
 log_error_verbosity: 776
 log_executor_stats: 776
 log_filename: 776
 log_hostname: 776
 log_min_duration_statement: 777
 log_min_error_statement: 777
 log_min_messages: 777
 log_parser_stats: 777
 log_planner_stats: 777
 log_rotation_age: 777
 log_rotation_size: 778
 log_statement: 778
 log_statement_stats: 778
 log_timezone: 778
 log_truncate_on_rotation: 778
 logical operators: 142
 logs

- on segments: 190
- parameters: 178
- searching: 643
- write ahead parameters: 175

 lseg: 815

M

macaddr: 815
 MACHINE_LIST_FILE: 788
 maintenance: 107, 109, 114, 123
 maintenance_work_mem: 779
 management scripts

- gp_dump: 583
- gp_restore: 588
- gpactivatestandby: 592
- gpaddmirrors: 595
- gpchecknet: 599
- gpcheckos: 602
- gpcheckperf: 604
- gpcrondump: 607
- gpdbrestore: 612
- gpdeletesystem: 615
- gpdetective: 617
- gpexpandsystem: 619
- gpfdist: 623
- gpinitstandby: 625
- gpinitssystem: 628
- gpload: 633
- gpmareduce: 647
- gpbuildsystem: 654
- gpcoverseg: 657
- gpscp: 661
- gpsizecalc: 663
- gpskew: 666
- gpssh: 669
- gpssh-exkeys: 671
- gpstart: 674
- gpstate: 676
- gpstop: 679

MapReduce

- document format: 791
- document schema: 793
- interface: 647
- specification: 791

 master: 12

- backup: 40
- configuring: 42
- data directory: 43
- installing: 40
- mirroring: 184
- recovery: 193

 master instance: 919
 master parameters: 170

- setting: 171

 master port

- changing the default: 173
- initial configuration: 789

 MASTER_DATA_DIRECTORY: 61, 812
 MASTER_DIRECTORY: 789
 MASTER_HOSTNAME: 789
 MASTER_PORT: 789
 mathematical functions: 142
 mathematical operators: 142
 max_appendonly_tables: 779
 max_connections: 84, 779
 max_connections parameter: 75
 max_files_per_process: 779
 max_fsm_pages: 779
 max_fsm_relations: 780
 max_function_args: 780
 max_identifier_length: 780
 max_index_keys: 780
 max_locks_per_transaction: 780
 max_prepared_transactions: 780
 max_resource_portals_per_transaction: 780
 max_resource_queues: 781
 max_stack_depth: 781
 metadata and log capacity: 36
 Microsoft SQL Server: 83
 Microstrategy: 83
 mirror: 919
 MIRROR_DATA_DIRECTORY: 790
 mirroring: 8

- of data: 183
- of master: 184
- of segments: 183
- setting up: 186
- viewing mirror configuration: 222

 MIVP: 63
 money: 815
 monitor performance: 11
 monitoring: 10, 221

- disk space usage: 222
- skew: 223
- system state: 221

 motion: 27

- gather: 26
- redistribute: 26

 motion node: 920

motion, defined: 26
 MOVE: 504
 MPP: 920
 Multiversion Concurrency Control: 125
 MVCC: 125, 227

N

network
 address functions: 142
 address operators: 142
 configuration: 33
 failure: 234
 infrastructure: 7
 layout: 32
 protocols: 7
 validating network performance: 39
 network protocol
 TCP: 7
 UDP: 7
 node, query plan: 26
 not-null constraints: 99
 numeric: 815
 numeric range partitioning: 110
 numsegments: 831

O

object privileges: 70
 ODBC: 77, 82
 operator classes: 394
 operators
 array: 142
 binary: 142
 bit string: 142
 comparison: 142
 date: 142
 geometric: 142
 logical: 142
 mathematical: 142
 network address: 142
 sequence: 142
 string: 142
 time: 142
 operators, user-defined: 389
 OS system settings: 34
 overcommit: 88

P

parameters
 append-only table: 180
 append-only tables: 181
 array configuration: 181
 automatic vacuuming: 179
 categories: 172
 client connection: 180
 connection and authentication: 173
 connection parameters: 77

error reporting and logging: 178
 external table: 180
 file location: 173
 lock management: 180
 master, global and local: 170
 query tuning: 176
 runtime statistics collection: 178
 setting level: 171
 system resource and consumption: 174
 version compatibility: 181
 workload management: 87, 180
 write ahead log: 175
 partition: 109, 110
 by date range: 109
 by numeric range: 110
 existing table: 112
 key: 109, 110
 list: 108
 range: 108
 strategy: 108
 strategy, verifying: 113
 tables: 107
 viewing: 114
 viewing design: 113
 partitioned: 921
 password file: 79
 password, default superuser: 629
 password_encryption: 781
 PATH: 812
 path: 815
 pattern matching: 142
 performance
 benchmarks: 232
 checking with gpcheckperf: 37
 defining: 231
 factors: 231
 hardware: 37
 hardware baseline: 232
 issues and causes: 108, 234
 troubleshooting: 238
 validating network performance: 39
 performance metrics: 221
 performance monitor: 11
 architecture: 11
 performance tuning
 OS system settings: 34
 Perl DBI: 82
 permissions: 492
 database: 67
 pg_bitmapindex: 97
 pg_catalog: 97
 pg_config: 708
 pg_controldata: 564
 pg_ctl: 564
 pg_dump: 711
 pg_dumpall: 721
 pg_hba.conf: 84
 editing: 74
 fields: 73

pg_partition_columns: 113, 114
 pg_partition_templates: 113, 114
 pg_partitions: 113, 114
 pg_resetxlog: 564
 pg_restore: 725
 PGCLIENTENCODING: 813
 PGDATABASE: 813
 PGDATESTYLE: 813
 PGHOST: 813
 PGHOSTADDR: 813
 pgjdbc: 82
 pgodbc: 82
 PGOPTIONS: 813
 PGPASSFILE: 813
 PGPASSWORD: 813
 pgperl: 82
 PGPORT: 813
 PGTZ: 813
 PGUSER: 813
 point: 815
 polygon: 815
 port: 781

- master, changing the default: 173
- master, setting before initialization: 789
- segments, setting before initialization: 788

 PORT_BASE: 788
 postgres: 564
 postgres user: 42
 PostgreSQL: 921
 postmaster: 564, 921
 preinstallation: 30
 PREPARE: 506
 prepared statements: 506
 primary key: 13
 privileges

- managing object privileges: 70

 privileges: 492
 procedural languages: 386
 productversion: 835
 psql: 77, 730, 922
 public key exchange: 671
 pygresql: 82

Q

QD: 922
 QE: 922
 query

- cost: 88, 235
- dispatcher: 28, 922
- execution plan: 922
- executor: 28, 922
- monitoring active: 11
- parsing: 26
- planner: 235
- status check: 239
- troubleshooting: 239
- tuning parameters: 176
- worker processes example: 28

 query plans: 26, 485

- about: 26

example: 26, 27
 parallel: 26
 statistics: 153

R

rack: 922
 RAID: 31, 922
 random distribution: 13
 random_page_cost: 781
 range partitioning: 108
 read uncommitted: 129
 read-committed: 128
 read-only mode: 188
 read-write mode: 188
 real: 815
 REASSIGN OWNED: 509
 rebuild database: 203
 recovery: 185

- of master: 193
- of segments: 190

 redistribute motion: 26, 27
 redundancy: 8, 183
 referential integrity: 100
 regex_flavor: 781
 REINDEX: 510
 reindex: 227
 reindexdb: 751
 RELEASE SAVEPOINT: 512
 repeatable read: 129
 requirements

- network: 34
- shared memory: 34

 reserved characters: 155
 RESET: 513
 resource contention: 238
 resource queues: 85, 399

- altering: 89
- creating: 87
- exempt users: 86
- viewing status: 90
- with roles: 89

 resource_cleanup_gangs_on_wait: 781
 resource_scheduler: 781
 resource_select_only: 86, 782
 resource queues

- creating: 88
- viewing status: 89

 restarting Greenplum Database: 168
 restore database: 195, 196, 200

- non-parallel: 197
- with different configuration: 202

 REVOKE: 514
 rewrite rules: 406
 roles: 67

- creating: 68, 402
- creating group membership: 69
- list of attributes: 68
- resource queues: 85

 ROLLBACK: 517

ROLLBACK TO SAVEPOINT: 518
 rotating log files: 229
 row comparisons: 142
 row-level lock: 235
 Rows: 127
 runtime statistics collection parameters: 178

S

SAS: 83
 SAVEPOINT: 520
 schema: 409
 inheritance: 107
 managing: 96
 MapReduce YAML document schema: 793
 pg_catalog: 97
 star: 12, 923
 system-level: 97
 schemaversion: 835
 scp, group session: 661
 search_path: 782
 SEG_PREFIX: 788
 segment: 12, 923
 about: 7
 data directories: 47
 failures: 189
 installing: 44
 mirroring: 183
 primary per host: 7
 recovery: 190
 segment instance: 923
 segment ports
 initial configuration: 788
 SELECT: 522
 SELECT INTO: 538
 seq_page_cost: 782
 sequence functions: 142
 sequence operators: 142
 sequences: 411
 creating and using: 118
 serial: 815
 serial4: 815
 serial8: 814
 serializable: 128
 server configuration: 755
 server logs: 224
 server programs: 564
 gpsyncmaster: 564
 initdb: 564
 ipcclean: 564
 pg_controldata: 564
 pg_ctl: 564
 pg_resetxlog: 564
 postgres: 564
 postmaster: 564
 server_encoding: 782
 server_version: 782
 server_version_num: 782
 sessions: 77
 SET: 540
 set returning functions: 142
 SET ROLE: 542
 SET SESSION AUTHORIZATION: 544
 SET TRANSACTION: 546
 setup: 59
 shared memory: 34
 shared_buffers: 782
 shared_preload_libraries: 783
 SHOW: 549
 silent_mode: 783
 skew: 223
 correcting skew: 105
 slice: 27
 slice, defined: 26
 smallint: 815
 sql_inheritance: 783
 ssh: 669
 group session: 669
 key exchange: 44, 671
 ssl: 783
 standard_conforming_strings: 783
 standby master: 184
 configuring: 187
 installing: 40
 star schema: 923
 START TRANSACTION: 550
 starting Greenplum Database: 167
 statement_timeout: 783
 statistics: 328
 database: 235
 query planner: 153
 stats_block_level: 784
 stats_command_string: 784
 stats_queue_level: 784
 stats_reset_server_on_start: 784
 stats_row_level: 784
 stats_start_collector: 784
 stopping Greenplum Database: 168
 storage capacity: 36
 stored procedures: 375
 string functions: 142
 string operators: 142
 subquery expressions: 142
 superuser: 67, 73
 role attribute: 86
 superuser password: 629
 superuser_reserved_connections: 784
 support, customer: 240
 switch configuration: 33
 system catalog: 923
 system catalogs
 global: 7
 gp_configuration: 820, 821, 822, 823, 827, 828, 830
 gp_distributed_log: 824
 gp_distributed_xacts: 825
 gp_distribution_policy: 826
 gp_id: 831
 gp_transaction_log: 833
 gp_version_at_initdb: 835
 system requirements

- CPU: 14
- disk: 14
- filesystem: 14
- memory: 14
- network: 14
- OS: 14
- software and utilities: 14
- system resource and consumption parameters: 174
- system resources and performance: 231
- system state: 221
- system utilization metrics: 11, 221
- system-levels schemas: 97

T

tables

- append-only: 101
- column-oriented: 101
- compressed append-only: 101
- creating: 415
- distributed: 98
- distributed: 105
- distribution policy: 100
- external: 153
- heap type: 101
- loading partitioned: 112
- locking: 235, 500
- maintaining partitioned: 114
- partition existing: 112
- partition strategy: 108
- partitioning: 107
- row-oriented: 101
- storage model: 101
- web: 153

- tablespaces: 430

- TCP: 7

- tcp_keepalives_count: 784

- tcp_keepalives_idle: 784

- tcp_keepalives_interval: 785

- temp_buffers: 785

- templates, database: 94

- text: 815

- The: 31

- time: 815

- time functions: 142

- time operators: 142

- time with time zone: 815

- timestamp: 816

- timestamp with time zone: 816

- timestamptz: 816

- timetz: 815

- TimeZone: 785

- timezone_abbreviations: 785

- TPC-H: 924

- TPCH: 64

- transaction_isolation: 785

- transaction_read_only: 785

- transactions

- committing: 340

- isolation level: 128, 330

- management: 235

- rolling back: 517

- savepoints: 520

- starting: 330

- working with: 128

- transform_null_equals: 785

- triggers: 433

- troubleshooting: 617

- client connections: 84

- data loading errors: 160

- gpfdist: 158

- performance problems: 238

- TRUNCATE: 128, 552

- trusted hosts: 44

- setup: 671

- TRUSTED_SHELL: 789

- tuple: 924

U

- UDP: 7

- unique constraints: 99

- unix_socket_directory: 786

- unix_socket_group: 786

- unix_socket_permissions: 786

- UPDATE: 553

- UPDATE command: 127

- update_process_title: 786

- user data size: 36

- user limits: 34

- users

- Greenplum super user: 42

- utilities

- administration: 10

- clusterdb: 694

- createdb: 696

- createlang: 698

- createuser: 718

- dropdb: 700

- droplang: 702

- dropuser: 704

- ecpg: 706

- gp_dump: 583

- gp_restore: 588

- gpactivatestandby: 592

- gpaddmirrors: 595

- gpchecknet: 599

- gpcheckos: 602

- gpcheckperf: 604

- gpcrondump: 607

- gpdrestore: 612

- gpdeletesystem: 615

- gpdetective: 617

- gpexpandsystem: 619

- gpfdist: 623

- gpinitstandby: 625

- gpinitssystem: 628

- gpload: 633

- gplogfilter: 643
- gmapreduce: 647
- gprebuildsystem: 654
- gprecoverseg: 657
- gpscp: 661
- gpsizecalc: 663
- gpskew: 666
- gpssh: 669
- gpssh-exkeys: 671
- gpstart: 674
- gpstate: 676
- gpstop: 679
- pg_config: 708
- pg_dump: 711
- pg_dumpall: 721
- pg_restore: 725
- psql: 730
- reindexdb: 751
- vacuumdb: 753

V

- VACUUM: 557
- vacuum: 227
- vacuum_cost_delay: 786
- vacuum_cost_limit: 786
- vacuum_cost_page_dirty: 786
- vacuum_cost_page_hit: 786
- vacuum_cost_page_miss: 786
- vacuum_freeze_min_age: 787
- vacuumdb: 753
- vacuuming parameters: 179
- valid: 820
- VALUES: 560
- varbit: 814
- varchar: 814
- version compatibility parameters: 181
- viewing
 - partition design: 113
 - partitions: 114
 - query plans: 485
- views: 444
 - creating and managing: 124

W

- wal_buffers: 787
- wal_synch_method: 787
- web tables: 153, 363
 - creating and using: 160, 161
 - defining command-based: 160, 162
 - defining URL-based: 163
 - loading data: 163
- work_mem: 787
- worker processes (query): 28
- workload and system performance: 231
- workload management: 85, 180, 235, 399
 - parameters: 87, 180
- write ahead log parameters: 175