

**EDB Postgres™ Enterprise Guide**

**EDB Postgres™ Advanced Server 9.5  
formerly Postgres Plus Advanced Server 9.5**

February 24, 2016

EDB Postgres™ Enterprise Guide  
by EnterpriseDB® Corporation  
Copyright © 2014 - 2016 EnterpriseDB Corporation

EnterpriseDB Corporation, 34 Crosby Drive, Suite 100, Bedford, MA 01730, USA  
**T**  +1 781 357 3390 **F**  +1 978 589 5701 **E** info@enterprisedb.com www.enterprisedb.com

Table of Contents

[1 Introduction 20](#_Toc444155334)

[1.1 What’s New 23](#_Toc444155335)

[1.2 Typographical Conventions Used in this Guide 24](#_Toc444155336)

[1.3 Other Conventions Used in this Guide 25](#_Toc444155337)

[1.4 About the Examples Used in this Guide 26](#_Toc444155338)

[1.4.1.1 Sample Database Description 26](#_Toc444155339)

[2 Database Administration 35](#_Toc444155340)

[2.1 Configuration Parameters 35](#_Toc444155341)

[2.1.1 Setting Configuration Parameters 36](#_Toc444155342)

[2.1.2 Summary of Configuration Parameters 38](#_Toc444155343)

[2.1.3 Configuration Parameters by Functionality 56](#_Toc444155344)

[2.1.3.1 Top Performance Related Parameters 57](#_Toc444155345)

[2.1.3.2 Resource Usage / Memory 66](#_Toc444155346)

[2.1.3.3 Resource Usage / EDB Resource Manager 70](#_Toc444155347)

[2.1.3.4 Query Tuning 72](#_Toc444155348)

[2.1.3.5 Query Tuning / Planner Method Configuration 72](#_Toc444155349)

[2.1.3.6 Reporting and Logging / What to Log 75](#_Toc444155350)

[2.1.3.7 EnterpriseDB Auditing Settings 76](#_Toc444155351)

[2.1.3.8 Client Connection Defaults / Locale and Formatting 81](#_Toc444155352)

[2.1.3.9 Client Connection Defaults / Statement Behavior 81](#_Toc444155353)

[2.1.3.10 Client Connection Defaults / Other Defaults 82](#_Toc444155354)

[2.1.3.11 Compatibility Options 84](#_Toc444155355)

[2.1.3.12 Customized Options 92](#_Toc444155356)

[2.1.3.13 Ungrouped 101](#_Toc444155357)

[2.2 Controlling the Audit Logs 104](#_Toc444155358)

[2.2.1 Auditing Configuration Parameters 104](#_Toc444155359)

[2.3 Unicode Collation Algorithm 108](#_Toc444155360)

[2.3.1 Basic Unicode Collation Algorithm Concepts 109](#_Toc444155361)

[2.3.2 International Components for Unicode 110](#_Toc444155362)

[2.3.2.1 Locale Collations 110](#_Toc444155363)

[2.3.2.2 Collation Attributes 111](#_Toc444155364)

[2.3.3 Creating an ICU Collation 114](#_Toc444155365)

[2.3.3.1 CREATE COLLATION 114](#_Toc444155366)

[2.3.3.2 CREATE DATABASE 116](#_Toc444155367)

[2.3.3.3 initdb 117](#_Toc444155368)

[2.3.4 Using a Collation 119](#_Toc444155369)

[2.4 Profile Management 123](#_Toc444155370)

[2.4.1 Creating a New Profile 124](#_Toc444155371)

[2.4.1.1 Creating a Password Function 127](#_Toc444155372)

[2.4.2 Altering a Profile 130](#_Toc444155373)

[2.4.3 Dropping a Profile 131](#_Toc444155374)

[2.4.4 Associating a Profile with an Existing Role 132](#_Toc444155375)

[2.4.5 Unlocking a Locked Account 134](#_Toc444155376)

[2.4.6 Creating a New Role Associated with a Profile 136](#_Toc444155377)

[2.4.7 Backing up Profile Management Functions 138](#_Toc444155378)

[3 Enhanced SQL Features 139](#_Toc444155379)

[3.1 Synonyms 139](#_Toc444155380)

[3.2 Hierarchical Queries 143](#_Toc444155381)

[3.2.1 Defining the Parent/Child Relationship 144](#_Toc444155382)

[3.2.2 Selecting the Root Nodes 144](#_Toc444155383)

[3.2.3 Organization Tree in the Sample Application 144](#_Toc444155384)

[3.2.4 Node Level 146](#_Toc444155385)

[3.2.5 Ordering the Siblings 147](#_Toc444155386)

[3.2.6 Retrieving the Root Node with CONNECT\_BY\_ROOT 148](#_Toc444155387)

[3.2.7 Retrieving a Path with SYS\_CONNECT\_BY\_PATH 152](#_Toc444155388)

[3.3 Extended Functions and Operators 154](#_Toc444155389)

[3.3.1 Logical Operators 154](#_Toc444155390)

[3.3.2 Comparison Operators 155](#_Toc444155391)

[3.3.3 Mathematical Functions and Operators 157](#_Toc444155392)

[3.3.4 String Functions and Operators 159](#_Toc444155393)

[3.3.5 Pattern Matching String Functions 162](#_Toc444155394)

[3.3.5.1 REGEXP\_COUNT 162](#_Toc444155395)

[3.3.5.2 REGEXP\_INSTR 163](#_Toc444155396)

[3.3.5.3 REGEXP\_SUBSTR 165](#_Toc444155397)

[3.3.6 Pattern Matching Using the LIKE Operator 168](#_Toc444155398)

[3.3.7 Data Type Formatting Functions 169](#_Toc444155399)

[3.3.7.1 IMMUTABLE TO\_CHAR(TIMESTAMP, format) Function 173](#_Toc444155400)

[3.3.8 Date/Time Functions and Operators 176](#_Toc444155401)

[3.3.8.1 ADD\_MONTHS 177](#_Toc444155402)

[3.3.8.2 EXTRACT 178](#_Toc444155403)

[3.3.8.3 MONTHS\_BETWEEN 179](#_Toc444155404)

[3.3.8.4 NEXT\_DAY 180](#_Toc444155405)

[3.3.8.5 NEW\_TIME 181](#_Toc444155406)

[3.3.8.6 ROUND 181](#_Toc444155407)

[3.3.8.7 TRUNC 186](#_Toc444155408)

[3.3.8.8 CURRENT DATE/TIME 189](#_Toc444155409)

[3.3.8.9 NUMTODSINTERVAL 190](#_Toc444155410)

[3.3.8.10 NUMTOYMINTERVAL 190](#_Toc444155411)

[3.3.9 Sequence Manipulation Functions 192](#_Toc444155412)

[3.3.10 Conditional Expressions 193](#_Toc444155413)

[3.3.10.1 CASE 193](#_Toc444155414)

[3.3.10.2 COALESCE 194](#_Toc444155415)

[3.3.10.3 NULLIF 195](#_Toc444155416)

[3.3.10.4 NVL 195](#_Toc444155417)

[3.3.10.5 NVL2 195](#_Toc444155418)

[3.3.10.6 GREATEST and LEAST 196](#_Toc444155419)

[3.3.11 Aggregate Functions 197](#_Toc444155420)

[3.3.12 Subquery Expressions 199](#_Toc444155421)

[3.3.12.1 EXISTS 199](#_Toc444155422)

[3.3.12.2 IN 200](#_Toc444155423)

[3.3.12.3 NOT IN 200](#_Toc444155424)

[3.3.12.4 ANY/SOME 200](#_Toc444155425)

[3.3.12.5 ALL 201](#_Toc444155426)

[3.3.13 Uniform Resource Locator Functions 202](#_Toc444155427)

[3.3.13.1 EDB\_GET\_URL\_AS\_BYTEA 202](#_Toc444155428)

[3.3.13.2 EDB\_GET\_URL\_AS\_TEXT 203](#_Toc444155429)

[3.4 Table Partitioning 205](#_Toc444155430)

[3.4.1 Selecting a Partition Type 206](#_Toc444155431)

[3.4.2 Using Partition Pruning 207](#_Toc444155432)

[3.4.3 Example - Partition Pruning 210](#_Toc444155433)

[3.5 Partitioning Command Syntax 213](#_Toc444155434)

[3.5.1 CREATE TABLE…PARTITION BY 213](#_Toc444155435)

[3.5.1.1 Example - PARTITION BY LIST 216](#_Toc444155436)

[3.5.1.2 Example - PARTITION BY RANGE 217](#_Toc444155437)

[3.5.1.3 Example - PARTITION BY HASH 218](#_Toc444155438)

[3.5.1.4 Example - PARTITION BY RANGE, SUBPARTITION BY LIST 219](#_Toc444155439)

[3.5.2 ALTER TABLE...ADD PARTITION 222](#_Toc444155440)

[3.5.2.1 Example - Adding a Partition to a LIST Partitioned Table 224](#_Toc444155441)

[3.5.2.2 Example - Adding a Partition to a RANGE Partitioned Table 225](#_Toc444155442)

[3.5.3 ALTER TABLE… ADD SUBPARTITION 227](#_Toc444155443)

[3.5.3.1 Example - Adding a Subpartition to a LIST-RANGE Partitioned Table 229](#_Toc444155444)

[3.5.3.2 Example - Adding a Subpartition to a RANGE-LIST Partitioned Table 230](#_Toc444155445)

[3.5.4 ALTER TABLE...SPLIT PARTITION 232](#_Toc444155446)

[3.5.4.1 Example - Splitting a LIST Partition 234](#_Toc444155447)

[3.5.4.2 Example - Splitting a RANGE Partition 236](#_Toc444155448)

[3.5.5 ALTER TABLE...SPLIT SUBPARTITION 239](#_Toc444155449)

[3.5.5.1 Example - Splitting a LIST Subpartition 241](#_Toc444155450)

[3.5.5.2 Example - Splitting a RANGE Subpartition 243](#_Toc444155451)

[3.5.6 ALTER TABLE… EXCHANGE PARTITION 247](#_Toc444155452)

[3.5.6.1 Example - Exchanging a Table for a Partition 248](#_Toc444155453)

[3.5.7 ALTER TABLE… MOVE PARTITION 251](#_Toc444155454)

[3.5.7.1 Example - Moving a Partition to a Different Tablespace 252](#_Toc444155455)

[3.5.8 ALTER TABLE… RENAME PARTITION 254](#_Toc444155456)

[3.5.8.1 Example - Renaming a Partition 255](#_Toc444155457)

[3.5.9 DROP TABLE 256](#_Toc444155458)

[3.5.10 ALTER TABLE… DROP PARTITION 257](#_Toc444155459)

[3.5.10.1 Example - Deleting a Partition 257](#_Toc444155460)

[3.5.11 ALTER TABLE… DROP SUBPARTITION 259](#_Toc444155461)

[3.5.11.1 Example - Deleting a Subpartition 259](#_Toc444155462)

[3.5.12 TRUNCATE TABLE 261](#_Toc444155463)

[3.5.12.1 Example - Emptying a Table 261](#_Toc444155464)

[3.5.13 ALTER TABLE… TRUNCATE PARTITION 264](#_Toc444155465)

[3.5.13.1 Example - Emptying a Partition 265](#_Toc444155466)

[3.5.14 ALTER TABLE… TRUNCATE SUBPARTITION 267](#_Toc444155467)

[3.5.14.1 Example - Emptying a Subpartition 267](#_Toc444155468)

[3.6 Handling Stray Values in a LIST or RANGE Partitioned Table 270](#_Toc444155469)

[3.7 Specifying Multiple Partitioning Keys in a RANGE Partitioned Table 274](#_Toc444155470)

[3.8 Retrieving Information about a Partitioned Table 276](#_Toc444155471)

[3.8.1 Table Partitioning Views - Reference 277](#_Toc444155472)

[3.8.1.1 ALL\_PART\_TABLES 277](#_Toc444155473)

[3.8.1.2 ALL\_TAB\_PARTITIONS 278](#_Toc444155474)

[3.8.1.3 ALL\_TAB\_SUBPARTITIONS 279](#_Toc444155475)

[3.8.1.4 ALL\_PART\_KEY\_COLUMNS 280](#_Toc444155476)

[3.8.1.5 ALL\_SUBPART\_KEY\_COLUMNS 280](#_Toc444155477)

[4 Security 281](#_Toc444155478)

[4.1 Protecting Against SQL Injection Attacks 281](#_Toc444155479)

[4.1.1 SQL/Protect Overview 282](#_Toc444155480)

[4.1.1.1 Types of SQL Injection Attacks 282](#_Toc444155481)

[4.1.1.2 Monitoring SQL Injection Attacks 283](#_Toc444155482)

[4.1.2 Configuring SQL/Protect 286](#_Toc444155483)

[4.1.2.1 Selecting Roles to Protect 289](#_Toc444155484)

[4.1.2.2 Monitoring Protected Roles 291](#_Toc444155485)

[4.1.3 Common Maintenance Operations 297](#_Toc444155486)

[4.1.3.1 Adding a Role to the Protected Roles List 297](#_Toc444155487)

[4.1.3.2 Removing a Role From the Protected Roles List 297](#_Toc444155488)

[4.1.3.3 Setting the Types of Protection for a Role 298](#_Toc444155489)

[4.1.3.4 Removing a Relation From the Protected Relations List 299](#_Toc444155490)

[4.1.3.5 Deleting Statistics 299](#_Toc444155491)

[4.1.3.6 Deleting Offending Queries 300](#_Toc444155492)

[4.1.3.7 Disabling and Enabling Monitoring 301](#_Toc444155493)

[4.1.4 Backing Up and Restoring a SQL/Protect Database 303](#_Toc444155494)

[4.1.4.1 Object Identification Numbers in SQL/Protect Tables 303](#_Toc444155495)

[4.1.4.2 Backing Up the Database 304](#_Toc444155496)

[4.1.4.3 Restoring From the Backup Files 304](#_Toc444155497)

[4.2 EDB\*Wrap 309](#_Toc444155498)

[4.2.1 Using EDB\*Wrap to Obfuscate Source Code 310](#_Toc444155499)

[4.3 Virtual Private Database 314](#_Toc444155500)

[5 EDB Resource Manager 315](#_Toc444155501)

[5.1 Creating and Managing Resource Groups 316](#_Toc444155502)

[5.1.1 CREATE RESOURCE GROUP 316](#_Toc444155503)

[5.1.2 ALTER RESOURCE GROUP 317](#_Toc444155504)

[5.1.3 DROP RESOURCE GROUP 319](#_Toc444155505)

[5.1.4 Assigning a Process to a Resource Group 319](#_Toc444155506)

[5.1.5 Removing a Process from a Resource Group 320](#_Toc444155507)

[5.1.6 Monitoring Processes in Resource Groups 321](#_Toc444155508)

[5.2 CPU Usage Throttling 323](#_Toc444155509)

[5.2.1 Setting the CPU Rate Limit for a Resource Group 323](#_Toc444155510)

[5.2.2 Example – Single Process in a Single Group 324](#_Toc444155511)

[5.2.3 Example – Multiple Processes in a Single Group 325](#_Toc444155512)

[5.2.4 Example – Multiple Processes in Multiple Groups 327](#_Toc444155513)

[5.3 Dirty Buffer Throttling 330](#_Toc444155514)

[5.3.1 Setting the Dirty Rate Limit for a Resource Group 330](#_Toc444155515)

[5.3.2 Example – Single Process in a Single Group 332](#_Toc444155516)

[5.3.3 Example – Multiple Processes in a Single Group 333](#_Toc444155517)

[5.3.4 Example – Multiple Processes in Multiple Groups 335](#_Toc444155518)

[5.4 System Catalogs 339](#_Toc444155519)

[5.4.1 edb\_all\_resource\_groups 339](#_Toc444155520)

[5.4.2 edb\_resource\_group 339](#_Toc444155521)

[6 Database Utilities 340](#_Toc444155522)

[6.1 EDB\*Loader 340](#_Toc444155523)

[6.1.1 Data Loading Methods 341](#_Toc444155524)

[6.1.2 General Usage 342](#_Toc444155525)

[6.1.3 Building the EDB\*Loader Control File 343](#_Toc444155526)

[6.1.4 Invoking EDB\*Loader 358](#_Toc444155527)

[6.1.4.1 Exit Codes 364](#_Toc444155528)

[6.1.5 Direct Path Load 365](#_Toc444155529)

[6.1.6 Parallel Direct Path Load 366](#_Toc444155530)

[6.1.7 Remote Loading 369](#_Toc444155531)

[6.1.8 Updating a Table with a Conventional Path Load 370](#_Toc444155532)

[6.1.9 Loading Empty Strings with EDB\*Loader 372](#_Toc444155533)

[6.2 EDB\*Plus 373](#_Toc444155534)

[6.2.1 Starting EDB\*Plus 373](#_Toc444155535)

[6.2.2 Command Summary 376](#_Toc444155536)

[6.2.2.1 ACCEPT 376](#_Toc444155537)

[6.2.2.2 APPEND 376](#_Toc444155538)

[6.2.2.3 CHANGE 376](#_Toc444155539)

[6.2.2.4 CLEAR 377](#_Toc444155540)

[6.2.2.5 COLUMN 377](#_Toc444155541)

[6.2.2.6 CONNECT 380](#_Toc444155542)

[6.2.2.7 DEFINE 381](#_Toc444155543)

[6.2.2.8 DEL 382](#_Toc444155544)

[6.2.2.9 DESCRIBE 383](#_Toc444155545)

[6.2.2.10 DISCONNECT 383](#_Toc444155546)

[6.2.2.11 EDIT 383](#_Toc444155547)

[6.2.2.12 EXECUTE 384](#_Toc444155548)

[6.2.2.13 EXIT 384](#_Toc444155549)

[6.2.2.14 GET 385](#_Toc444155550)

[6.2.2.15 HELP 385](#_Toc444155551)

[6.2.2.16 HOST 386](#_Toc444155552)

[6.2.2.17 INPUT 386](#_Toc444155553)

[6.2.2.18 LIST 386](#_Toc444155554)

[6.2.2.19 PASSWORD 387](#_Toc444155555)

[6.2.2.20 PAUSE 387](#_Toc444155556)

[6.2.2.21 PRINT 388](#_Toc444155557)

[6.2.2.22 PROMPT 388](#_Toc444155558)

[6.2.2.23 QUIT 388](#_Toc444155559)

[6.2.2.24 REMARK 388](#_Toc444155560)

[6.2.2.25 SAVE 389](#_Toc444155561)

[6.2.2.26 SET 389](#_Toc444155562)

[6.2.2.27 SHOW 394](#_Toc444155563)

[6.2.2.28 SPOOL 394](#_Toc444155564)

[6.2.2.29 START 395](#_Toc444155565)

[6.2.2.30 UNDEFINE 395](#_Toc444155566)

[6.2.2.31 WHENEVER SQLERROR 395](#_Toc444155567)

[6.3 libpq C Library 397](#_Toc444155568)

[6.3.1 Using libpq with EnterpriseDB SPL 397](#_Toc444155569)

[6.3.2 REFCURSOR Support 397](#_Toc444155570)

[6.3.3 Array Binding 404](#_Toc444155571)

[6.3.3.1 PQBulkStart 404](#_Toc444155572)

[6.3.3.2 PQexecBulk 404](#_Toc444155573)

[6.3.3.3 PQBulkFinish 405](#_Toc444155574)

[6.3.3.4 PQexecBulkPrepared 405](#_Toc444155575)

[6.3.3.5 Example Code (Using PQBulkStart, PQexecBulk, PQBulkFinish) 406](#_Toc444155576)

[6.3.3.6 Example Code (Using PQexecBulkPrepared) 407](#_Toc444155577)

[6.4 ECPGPlus 408](#_Toc444155578)

[6.4.1 C-preprocessor Directives 409](#_Toc444155579)

[6.4.2 Supported C Data Types 411](#_Toc444155580)

[6.4.3 Type Codes 413](#_Toc444155581)

[6.4.4 The SQLDA Structure 414](#_Toc444155582)

[6.4.5 ECPGPlus Statements 418](#_Toc444155583)

[6.4.5.1 ALLOCATE DESCRIPTOR 418](#_Toc444155584)

[6.4.5.2 CALL 419](#_Toc444155585)

[6.4.5.3 CLOSE 420](#_Toc444155586)

[6.4.5.4 COMMIT 420](#_Toc444155587)

[6.4.5.5 CONNECT 421](#_Toc444155588)

[6.4.5.6 DEALLOCATE DESCRIPTOR 423](#_Toc444155589)

[6.4.5.7 DECLARE CURSOR 424](#_Toc444155590)

[6.4.5.8 DECLARE DATABASE 424](#_Toc444155591)

[6.4.5.9 DECLARE STATEMENT 425](#_Toc444155592)

[6.4.5.10 DELETE 425](#_Toc444155593)

[6.4.5.11 DESCRIBE 427](#_Toc444155594)

[6.4.5.12 DESCRIBE DESCRIPTOR 428](#_Toc444155595)

[6.4.5.13 DISCONNECT 429](#_Toc444155596)

[6.4.5.14 EXECUTE 430](#_Toc444155597)

[6.4.5.15 EXECUTE DESCRIPTOR 431](#_Toc444155598)

[6.4.5.16 EXECUTE...END EXEC 432](#_Toc444155599)

[6.4.5.17 EXECUTE IMMEDIATE 432](#_Toc444155600)

[6.4.5.18 FETCH 433](#_Toc444155601)

[6.4.5.19 FETCH DESCRIPTOR 434](#_Toc444155602)

[6.4.5.20 GET DESCRIPTOR 435](#_Toc444155603)

[6.4.5.21 INSERT 436](#_Toc444155604)

[6.4.5.22 OPEN 437](#_Toc444155605)

[6.4.5.23 OPEN DESCRIPTOR 438](#_Toc444155606)

[6.4.5.24 PREPARE 439](#_Toc444155607)

[6.4.5.25 ROLLBACK 440](#_Toc444155608)

[6.4.5.26 SAVEPOINT 441](#_Toc444155609)

[6.4.5.27 SELECT 441](#_Toc444155610)

[6.4.5.28 SET CONNECTION 442](#_Toc444155611)

[6.4.5.29 SET DESCRIPTOR 444](#_Toc444155612)

[6.4.5.30 UPDATE 446](#_Toc444155613)

[6.4.5.31 WHENEVER 447](#_Toc444155614)

[7 Open Client Library 449](#_Toc444155615)

[7.1 Comparison with Oracle Call Interface 449](#_Toc444155616)

[7.2 Compiling and Linking a Program 450](#_Toc444155617)

[7.3 Ref Cursor Support 452](#_Toc444155618)

[7.4 OCL Function Reference 455](#_Toc444155619)

[7.4.1 Connect, Authorize and Initialize Functions 455](#_Toc444155620)

[7.4.1.1 Using the tnsnames.ora File 455](#_Toc444155621)

[7.4.2 Handle and Descriptor Functions 456](#_Toc444155622)

[7.4.2.1 EDB\_ATTR\_EMPTY\_STRINGS 456](#_Toc444155623)

[7.4.2.2 EDB\_ATTR\_HOLDABLE 457](#_Toc444155624)

[7.4.2.3 EDB\_ATTR\_STMT\_LVL\_TX 458](#_Toc444155625)

[7.4.3 Bind, Define and Describe Functions 458](#_Toc444155626)

[7.4.4 Statement Functions 459](#_Toc444155627)

[7.4.5 Transaction Functions 459](#_Toc444155628)

[7.4.6 XA Functions 459](#_Toc444155629)

[7.4.6.1 xaoSvcCtx 459](#_Toc444155630)

[7.4.7 Date and Datetime Functions 460](#_Toc444155631)

[7.4.8 Interval Functions 461](#_Toc444155632)

[7.4.9 Number Functions 461](#_Toc444155633)

[7.4.10 String Functions 462](#_Toc444155634)

[7.4.11 Cartridge Services and File I/O Interface Functions 463](#_Toc444155635)

[7.4.12 LOB Functions 463](#_Toc444155636)

[7.4.13 Miscellaneous Functions 463](#_Toc444155637)

[7.4.14 Supported Data Types 463](#_Toc444155638)

[7.5 Debugger 465](#_Toc444155639)

[7.5.1 Configuring the Debugger 465](#_Toc444155640)

[7.5.2 Starting the Debugger 466](#_Toc444155641)

[7.5.3 The View Data Options Window 468](#_Toc444155642)

[7.5.4 Main Debugger Window 470](#_Toc444155643)

[7.5.4.1 The Program Body Pane 471](#_Toc444155644)

[7.5.4.2 The Stack Pane 472](#_Toc444155645)

[7.5.4.3 The Output Pane 475](#_Toc444155646)

[7.5.4.4 The Status Bar 476](#_Toc444155647)

[7.5.5 Debugging a Program 477](#_Toc444155648)

[7.5.5.1 Stepping Through the Code 477](#_Toc444155649)

[7.5.5.2 Using Breakpoints 478](#_Toc444155650)

[7.5.5.3 Setting a Global Breakpoint for In-Context Debugging 481](#_Toc444155651)

[7.5.5.4 Exiting the Debugger 487](#_Toc444155652)

[8 Performance Analysis and Tuning 488](#_Toc444155653)

[8.1 Dynatune 488](#_Toc444155654)

[8.1.1 edb\_dynatune 488](#_Toc444155655)

[8.1.2 edb\_dynatune\_profile 489](#_Toc444155656)

[8.2 Infinite Cache 490](#_Toc444155657)

[8.2.1 Installing Infinite Cache 494](#_Toc444155658)

[8.2.2 Configuring the Infinite Cache Server 496](#_Toc444155659)

[8.2.2.1 Modifying Infinite Cache Settings 496](#_Toc444155660)

[8.2.2.2 Enabling Infinite Cache 496](#_Toc444155661)

[8.2.2.3 Controlling the Infinite Cache Server 499](#_Toc444155662)

[8.2.3 Dynamically Modifying Infinite Cache Server Nodes 500](#_Toc444155663)

[8.2.4 Controlling the edb-icache Daemons 501](#_Toc444155664)

[8.2.4.1 Command Line Options 501](#_Toc444155665)

[8.2.4.2 edb-icache-tool 503](#_Toc444155666)

[8.2.5 Warming the edb-icache Servers 505](#_Toc444155667)

[8.2.5.1 The edb\_icache\_warm() Function 505](#_Toc444155668)

[8.2.5.2 Using the edb\_icache\_warm Utility 506](#_Toc444155669)

[8.2.6 Retrieving Statistics from Infinite Cache 507](#_Toc444155670)

[8.2.6.1 Using edb\_icache\_stats() 507](#_Toc444155671)

[8.2.6.2 edb\_icache\_server\_list 508](#_Toc444155672)

[8.2.7 Retrieving Table Statistics 510](#_Toc444155673)

[8.2.7.1 pg\_statio\_all\_tables 510](#_Toc444155674)

[8.2.7.2 pg\_statio\_sys\_tables 512](#_Toc444155675)

[8.2.7.3 pg\_statio\_user\_tables 512](#_Toc444155676)

[8.2.7.4 pg\_statio\_all\_indexes 512](#_Toc444155677)

[8.2.7.5 pg\_statio\_sys\_indexes 514](#_Toc444155678)

[8.2.7.6 pg\_statio\_user\_indexes 514](#_Toc444155679)

[8.2.8 edb\_icache\_server\_enable() 515](#_Toc444155680)

[8.2.9 Infinite Cache Log Entries 516](#_Toc444155681)

[8.2.10 Allocating Memory to the Cache Servers 516](#_Toc444155682)

[8.3 Index Advisor 517](#_Toc444155683)

[8.3.1 Index Advisor Components 518](#_Toc444155684)

[8.3.2 Index Advisor Configuration 519](#_Toc444155685)

[8.3.3 Using Index Advisor 522](#_Toc444155686)

[8.3.3.1 Using the pg\_advise\_index Utility 522](#_Toc444155687)

[8.3.3.2 Using Index Advisor at the psql Command Line 524](#_Toc444155688)

[8.3.4 Reviewing the Index Advisor Recommendations 526](#_Toc444155689)

[8.3.4.1 Using the show\_index\_recommendations() Function 526](#_Toc444155690)

[8.3.4.2 Querying the index\_advisor\_log Table 527](#_Toc444155691)

[8.3.4.3 Querying the index\_recommendations View 529](#_Toc444155692)

[8.3.5 Limitations 530](#_Toc444155693)

[8.4 SQL Profiler 532](#_Toc444155694)

[8.5 Query Optimization Hints 533](#_Toc444155695)

[8.5.1 Default Optimization Modes 535](#_Toc444155696)

[8.5.2 Access Method Hints 537](#_Toc444155697)

[8.5.3 Specifying a Join Order 541](#_Toc444155698)

[8.5.4 Joining Relations Hints 542](#_Toc444155699)

[8.5.5 Global Hints 545](#_Toc444155700)

[8.5.6 Using the APPEND Optimizer Hint 548](#_Toc444155701)

[8.5.7 Conflicting Hints 549](#_Toc444155702)

[8.6 DBMS\_PROFILER 550](#_Toc444155703)

[8.6.1 Querying the DBMS\_PROFILER Tables and View 552](#_Toc444155704)

[8.6.2 DBMS\_PROFILER Functions and Procedures 559](#_Toc444155705)

[8.6.2.1 FLUSH\_DATA 559](#_Toc444155706)

[8.6.2.2 GET\_VERSION 560](#_Toc444155707)

[8.6.2.3 INTERNAL\_VERSION\_CHECK 560](#_Toc444155708)

[8.6.2.4 PAUSE\_PROFILER 560](#_Toc444155709)

[8.6.2.5 RESUME\_PROFILER 561](#_Toc444155710)

[8.6.2.6 START\_PROFILER 561](#_Toc444155711)

[8.6.2.7 STOP\_PROFILER 562](#_Toc444155712)

[8.6.3 DBMS\_PROFILER - Reference 563](#_Toc444155713)

[8.6.3.1 PLSQL\_PROFILER\_RUNS 563](#_Toc444155714)

[8.6.3.2 PLSQL\_PROFILER\_UNITS 563](#_Toc444155715)

[8.6.3.3 PLSQL\_PROFILER\_DATA 564](#_Toc444155716)

[8.6.3.4 PLSQL\_PROFILER\_RAWDATA 564](#_Toc444155717)

[8.7 Dynamic Runtime Instrumentation Tools Architecture (DRITA) 569](#_Toc444155718)

[8.7.1 Configuring and Using DRITA 569](#_Toc444155719)

[8.8 DRITA Functions 571](#_Toc444155720)

[8.8.1 get\_snaps() 571](#_Toc444155721)

[8.8.2 sys\_rpt() 571](#_Toc444155722)

[8.8.3 sess\_rpt() 572](#_Toc444155723)

[8.8.4 sessid\_rpt() 573](#_Toc444155724)

[8.8.5 sesshist\_rpt() 575](#_Toc444155725)

[8.8.6 purgesnap() 576](#_Toc444155726)

[8.8.7 truncsnap() 577](#_Toc444155727)

[8.9 Simulating Statspack AWR Reports 578](#_Toc444155728)

[8.9.1 edbreport() 578](#_Toc444155729)

[8.9.2 stat\_db\_rpt() 586](#_Toc444155730)

[8.9.3 stat\_tables\_rpt() 587](#_Toc444155731)

[8.9.4 statio\_tables\_rpt() 589](#_Toc444155732)

[8.9.5 stat\_indexes\_rpt() 591](#_Toc444155733)

[8.9.6 statio\_indexes\_rpt() 593](#_Toc444155734)

[8.10 Performance Tuning Recommendations 595](#_Toc444155735)

[8.11 Event Descriptions 596](#_Toc444155736)

[8.12 Catalog Views 598](#_Toc444155737)

[8.12.1 edb$system\_waits 598](#_Toc444155738)

[8.12.2 edb$session\_waits 599](#_Toc444155739)

[8.12.3 edb$session\_wait\_history 600](#_Toc444155740)

[9 Built-In Utility Packages 602](#_Toc444155741)

[9.1 DBMS\_ALERT 604](#_Toc444155742)

[9.1.1 REGISTER 605](#_Toc444155743)

[9.1.2 REMOVE 605](#_Toc444155744)

[9.1.3 REMOVEALL 606](#_Toc444155745)

[9.1.4 SIGNAL 606](#_Toc444155746)

[9.1.5 WAITANY 606](#_Toc444155747)

[9.1.6 WAITONE 608](#_Toc444155748)

[9.1.7 Comprehensive Example 609](#_Toc444155749)

[9.2 DBMS\_CRYPTO 612](#_Toc444155750)

[9.2.1 DECRYPT 613](#_Toc444155751)

[9.2.2 ENCRYPT 615](#_Toc444155752)

[9.2.3 HASH 617](#_Toc444155753)

[9.2.4 MAC 618](#_Toc444155754)

[9.2.5 RANDOMBYTES 619](#_Toc444155755)

[9.2.6 RANDOMINTEGER 619](#_Toc444155756)

[9.2.7 RANDOMNUMBER 620](#_Toc444155757)

[9.3 DBMS\_JOB 621](#_Toc444155758)

[9.3.1 BROKEN 622](#_Toc444155759)

[9.3.2 CHANGE 623](#_Toc444155760)

[9.3.3 INTERVAL 624](#_Toc444155761)

[9.3.4 NEXT\_DATE 624](#_Toc444155762)

[9.3.5 REMOVE 625](#_Toc444155763)

[9.3.6 RUN 625](#_Toc444155764)

[9.3.7 SUBMIT 626](#_Toc444155765)

[9.3.8 WHAT 627](#_Toc444155766)

[9.4 DBMS\_LOB 628](#_Toc444155767)

[9.4.1 APPEND 629](#_Toc444155768)

[9.4.2 COMPARE 629](#_Toc444155769)

[9.4.3 CONVERTTOBLOB 630](#_Toc444155770)

[9.4.4 CONVERTTOCLOB 632](#_Toc444155771)

[9.4.5 COPY 633](#_Toc444155772)

[9.4.6 ERASE 634](#_Toc444155773)

[9.4.7 GET\_STORAGE\_LIMIT 635](#_Toc444155774)

[9.4.8 GETLENGTH 635](#_Toc444155775)

[9.4.9 INSTR 635](#_Toc444155776)

[9.4.10 READ 636](#_Toc444155777)

[9.4.11 SUBSTR 637](#_Toc444155778)

[9.4.12 TRIM 638](#_Toc444155779)

[9.4.13 WRITE 638](#_Toc444155780)

[9.4.14 WRITEAPPEND 639](#_Toc444155781)

[9.5 DBMS\_LOCK 640](#_Toc444155782)

[9.5.1 SLEEP 640](#_Toc444155783)

[9.6 DBMS\_MVIEW 641](#_Toc444155784)

[9.6.1 GET\_MV\_DEPENDENCIES 642](#_Toc444155785)

[9.6.2 REFRESH 642](#_Toc444155786)

[9.6.3 REFRESH\_ALL\_MVIEWS 644](#_Toc444155787)

[9.6.4 REFRESH\_DEPENDENT 646](#_Toc444155788)

[9.7 DBMS\_OUTPUT 648](#_Toc444155789)

[9.7.1 CHARARR 648](#_Toc444155790)

[9.7.2 DISABLE 649](#_Toc444155791)

[9.7.3 ENABLE 649](#_Toc444155792)

[9.7.4 GET\_LINE 650](#_Toc444155793)

[9.7.5 GET\_LINES 652](#_Toc444155794)

[9.7.6 NEW\_LINE 653](#_Toc444155795)

[9.7.7 PUT 653](#_Toc444155796)

[9.7.8 PUT\_LINE 655](#_Toc444155797)

[9.7.9 SERVEROUTPUT 655](#_Toc444155798)

[9.8 DBMS\_PIPE 657](#_Toc444155799)

[9.8.1 CREATE\_PIPE 658](#_Toc444155800)

[9.8.2 NEXT\_ITEM\_TYPE 659](#_Toc444155801)

[9.8.3 PACK\_MESSAGE 661](#_Toc444155802)

[9.8.4 PURGE 661](#_Toc444155803)

[9.8.5 RECEIVE\_MESSAGE 663](#_Toc444155804)

[9.8.6 REMOVE\_PIPE 663](#_Toc444155805)

[9.8.7 RESET\_BUFFER 665](#_Toc444155806)

[9.8.8 SEND\_MESSAGE 666](#_Toc444155807)

[9.8.9 UNIQUE\_SESSION\_NAME 667](#_Toc444155808)

[9.8.10 UNPACK\_MESSAGE 667](#_Toc444155809)

[9.8.11 Comprehensive Example 668](#_Toc444155810)

[9.9 DBMS\_PROFILER 671](#_Toc444155811)

[9.9.1 FLUSH\_DATA 672](#_Toc444155812)

[9.9.2 GET\_VERSION 672](#_Toc444155813)

[9.9.3 INTERNAL\_VERSION\_CHECK 672](#_Toc444155814)

[9.9.4 PAUSE\_PROFILER 673](#_Toc444155815)

[9.9.5 RESUME\_PROFILER 673](#_Toc444155816)

[9.9.6 START\_PROFILER 673](#_Toc444155817)

[9.9.7 STOP\_PROFILER 674](#_Toc444155818)

[9.10 DBMS\_RANDOM 675](#_Toc444155819)

[9.10.1 INITIALIZE 675](#_Toc444155820)

[9.10.2 NORMAL 676](#_Toc444155821)

[9.10.3 RANDOM 676](#_Toc444155822)

[9.10.4 SEED 677](#_Toc444155823)

[9.10.5 SEED 677](#_Toc444155824)

[9.10.6 STRING 678](#_Toc444155825)

[9.10.7 TERMINATE 679](#_Toc444155826)

[9.10.8 VALUE 679](#_Toc444155827)

[9.10.9 VALUE 679](#_Toc444155828)

[9.11 DBMS\_RLS 681](#_Toc444155829)

[9.11.1 ADD\_POLICY 684](#_Toc444155830)

[9.11.2 DROP\_POLICY 691](#_Toc444155831)

[9.11.3 ENABLE\_POLICY 691](#_Toc444155832)

[9.12 DBMS\_SCHEDULER 693](#_Toc444155833)

[9.12.1 Using Calendar Syntax to Specify a Repeating Interval 694](#_Toc444155834)

[9.12.2 CREATE\_JOB 696](#_Toc444155835)

[9.12.3 CREATE\_PROGRAM 698](#_Toc444155836)

[9.12.4 CREATE\_SCHEDULE 700](#_Toc444155837)

[9.12.5 DEFINE\_PROGRAM\_ARGUMENT 701](#_Toc444155838)

[9.12.6 DISABLE 703](#_Toc444155839)

[9.12.7 DROP\_JOB 704](#_Toc444155840)

[9.12.8 DROP\_PROGRAM 705](#_Toc444155841)

[9.12.9 DROP\_PROGRAM\_ARGUMENT 706](#_Toc444155842)

[9.12.10 DROP\_SCHEDULE 707](#_Toc444155843)

[9.12.11 ENABLE 707](#_Toc444155844)

[9.12.12 EVALUATE\_CALENDAR\_STRING 708](#_Toc444155845)

[9.12.13 RUN\_JOB 710](#_Toc444155846)

[9.12.14 SET\_JOB\_ARGUMENT\_VALUE 710](#_Toc444155847)

[9.13 DBMS\_SESSION 712](#_Toc444155848)

[9.13.1 SET\_ROLE 712](#_Toc444155849)

[9.14 DBMS\_SQL 713](#_Toc444155850)

[9.14.1 BIND\_VARIABLE 714](#_Toc444155851)

[9.14.2 BIND\_VARIABLE\_CHAR 715](#_Toc444155852)

[9.14.3 BIND VARIABLE RAW 716](#_Toc444155853)

[9.14.4 CLOSE\_CURSOR 716](#_Toc444155854)

[9.14.5 COLUMN\_VALUE 717](#_Toc444155855)

[9.14.6 COLUMN\_VALUE\_CHAR 718](#_Toc444155856)

[9.14.7 COLUMN VALUE RAW 719](#_Toc444155857)

[9.14.8 DEFINE\_COLUMN 720](#_Toc444155858)

[9.14.9 DEFINE\_COLUMN\_CHAR 722](#_Toc444155859)

[9.14.10 DEFINE COLUMN RAW 722](#_Toc444155860)

[9.14.11 DESCRIBE COLUMNS 723](#_Toc444155861)

[9.14.12 EXECUTE 724](#_Toc444155862)

[9.14.13 EXECUTE\_AND\_FETCH 724](#_Toc444155863)

[9.14.14 FETCH\_ROWS 726](#_Toc444155864)

[9.14.15 IS\_OPEN 728](#_Toc444155865)

[9.14.16 LAST\_ROW\_COUNT 728](#_Toc444155866)

[9.14.17 OPEN\_CURSOR 729](#_Toc444155867)

[9.14.18 PARSE 730](#_Toc444155868)

[9.15 DBMS\_UTILITY 732](#_Toc444155869)

[9.15.1 LNAME\_ARRAY 733](#_Toc444155870)

[9.15.2 UNCL\_ARRAY 733](#_Toc444155871)

[9.15.3 ANALYZE\_DATABASE, ANALYZE SCHEMA and ANALYZE PART\_OBJECT 733](#_Toc444155872)

[9.15.4 CANONICALIZE 736](#_Toc444155873)

[9.15.5 COMMA\_TO\_TABLE 737](#_Toc444155874)

[9.15.6 DB\_VERSION 738](#_Toc444155875)

[9.15.7 EXEC\_DDL\_STATEMENT 739](#_Toc444155876)

[9.15.8 FORMAT\_CALL\_STACK 740](#_Toc444155877)

[9.15.9 GET\_CPU\_TIME 740](#_Toc444155878)

[9.15.10 GET\_DEPENDENCY 741](#_Toc444155879)

[9.15.11 GET\_HASH\_VALUE 742](#_Toc444155880)

[9.15.12 GET\_PARAMETER\_VALUE 743](#_Toc444155881)

[9.15.13 GET\_TIME 744](#_Toc444155882)

[9.15.14 NAME\_TOKENIZE 744](#_Toc444155883)

[9.15.15 TABLE\_TO\_COMMA 747](#_Toc444155884)

[9.16 UTL\_ENCODE 749](#_Toc444155885)

[9.16.1 BASE64\_DECODE 749](#_Toc444155886)

[9.16.2 BASE64\_ENCODE 750](#_Toc444155887)

[9.16.3 MIMEHEADER\_DECODE 751](#_Toc444155888)

[9.16.4 MIMEHEADER\_ENCODE 752](#_Toc444155889)

[9.16.5 QUOTED\_PRINTABLE\_DECODE 753](#_Toc444155890)

[9.16.6 QUOTED\_PRINTABLE\_ENCODE 754](#_Toc444155891)

[9.16.7 TEXT\_DECODE 755](#_Toc444155892)

[9.16.8 TEXT\_ENCODE 756](#_Toc444155893)

[9.16.9 UUDECODE 757](#_Toc444155894)

[9.16.10 UUENCODE 759](#_Toc444155895)

[9.17 UTL\_FILE 761](#_Toc444155896)

[9.17.1 Setting File Permissions with utl\_file.umask 762](#_Toc444155897)

[9.17.2 FCLOSE 764](#_Toc444155898)

[9.17.3 FCLOSE\_ALL 764](#_Toc444155899)

[9.17.4 FCOPY 764](#_Toc444155900)

[9.17.5 FFLUSH 766](#_Toc444155901)

[9.17.6 FOPEN 767](#_Toc444155902)

[9.17.7 FREMOVE 768](#_Toc444155903)

[9.17.8 FRENAME 768](#_Toc444155904)

[9.17.9 GET\_LINE 770](#_Toc444155905)

[9.17.10 IS\_OPEN 771](#_Toc444155906)

[9.17.11 NEW\_LINE 772](#_Toc444155907)

[9.17.12 PUT 773](#_Toc444155908)

[9.17.13 PUT\_LINE 775](#_Toc444155909)

[9.17.14 PUTF 776](#_Toc444155910)

[9.18 UTL\_HTTP 778](#_Toc444155911)

[9.18.1 HTML\_PIECES 781](#_Toc444155912)

[9.18.2 REQ 781](#_Toc444155913)

[9.18.3 RESP 781](#_Toc444155914)

[9.18.4 BEGIN\_REQUEST 782](#_Toc444155915)

[9.18.5 END\_REQUEST 782](#_Toc444155916)

[9.18.6 END\_RESPONSE 783](#_Toc444155917)

[9.18.7 GET\_BODY\_CHARSET 783](#_Toc444155918)

[9.18.8 GET\_FOLLOW\_REDIRECT 784](#_Toc444155919)

[9.18.9 GET\_HEADER 784](#_Toc444155920)

[9.18.10 GET\_HEADER\_BY\_NAME 786](#_Toc444155921)

[9.18.11 GET\_HEADER\_COUNT 787](#_Toc444155922)

[9.18.12 GET\_RESPONSE 787](#_Toc444155923)

[9.18.13 GET\_RESPONSE\_ERROR\_CHECK 787](#_Toc444155924)

[9.18.14 GET\_TRANSFER\_TIMEOUT 788](#_Toc444155925)

[9.18.15 READ\_LINE 788](#_Toc444155926)

[9.18.16 READ\_RAW 790](#_Toc444155927)

[9.18.17 READ\_TEXT 791](#_Toc444155928)

[9.18.18 REQUEST 792](#_Toc444155929)

[9.18.19 REQUEST\_PIECES 792](#_Toc444155930)

[9.18.20 SET\_BODY\_CHARSET 793](#_Toc444155931)

[9.18.21 SET\_FOLLOW\_REDIRECT 793](#_Toc444155932)

[9.18.22 SET\_HEADER 794](#_Toc444155933)

[9.18.23 SET\_RESPONSE\_ERROR\_CHECK 794](#_Toc444155934)

[9.18.24 SET\_TRANSFER\_TIMEOUT 795](#_Toc444155935)

[9.18.25 WRITE\_LINE 796](#_Toc444155936)

[9.18.26 WRITE\_RAW 797](#_Toc444155937)

[9.18.27 WRITE\_TEXT 798](#_Toc444155938)

[9.19 UTL\_MAIL 799](#_Toc444155939)

[9.19.1 SEND 799](#_Toc444155940)

[9.19.2 SEND\_ATTACH\_RAW 800](#_Toc444155941)

[9.19.3 SEND\_ATTACH\_VARCHAR2 802](#_Toc444155942)

[9.20 UTL\_RAW 804](#_Toc444155943)

[9.20.1 CAST\_TO\_RAW 804](#_Toc444155944)

[9.20.2 CAST\_TO\_VARCHAR2 805](#_Toc444155945)

[9.20.3 CONCAT 806](#_Toc444155946)

[9.20.4 CONVERT 807](#_Toc444155947)

[9.20.5 LENGTH 808](#_Toc444155948)

[9.20.6 SUBSTR 809](#_Toc444155949)

[9.21 UTL\_SMTP 810](#_Toc444155950)

[9.21.1 CONNECTION 811](#_Toc444155951)

[9.21.2 REPLY/REPLIES 811](#_Toc444155952)

[9.21.3 CLOSE\_DATA 811](#_Toc444155953)

[9.21.4 COMMAND 811](#_Toc444155954)

[9.21.5 COMMAND\_REPLIES 812](#_Toc444155955)

[9.21.6 DATA 813](#_Toc444155956)

[9.21.7 EHLO 813](#_Toc444155957)

[9.21.8 HELO 814](#_Toc444155958)

[9.21.9 HELP 814](#_Toc444155959)

[9.21.10 MAIL 815](#_Toc444155960)

[9.21.11 NOOP 815](#_Toc444155961)

[9.21.12 OPEN\_CONNECTION 816](#_Toc444155962)

[9.21.13 OPEN\_DATA 816](#_Toc444155963)

[9.21.14 QUIT 817](#_Toc444155964)

[9.21.15 RCPT 817](#_Toc444155965)

[9.21.16 RSET 818](#_Toc444155966)

[9.21.17 VRFY 818](#_Toc444155967)

[9.21.18 WRITE\_DATA 818](#_Toc444155968)

[9.21.19 Comprehensive Example 819](#_Toc444155969)

[9.22 UTL\_URL 821](#_Toc444155970)

[9.22.1 ESCAPE 821](#_Toc444155971)

[9.22.2 UNESCAPE 823](#_Toc444155972)

[10 Expanded Catalog Views 825](#_Toc444155973)

[10.1 ALL\_ALL\_TABLES 825](#_Toc444155974)

[10.2 ALL\_CONS\_COLUMNS 825](#_Toc444155975)

[10.3 ALL\_CONSTRAINTS 826](#_Toc444155976)

[10.4 ALL\_DB\_LINKS 826](#_Toc444155977)

[10.5 ALL\_IND\_COLUMNS 827](#_Toc444155978)

[10.6 ALL\_INDEXES 827](#_Toc444155979)

[10.7 ALL\_JOBS 828](#_Toc444155980)

[10.8 ALL\_OBJECTS 828](#_Toc444155981)

[10.9 ALL\_PART\_KEY\_COLUMNS 829](#_Toc444155982)

[10.10 ALL\_PART\_TABLES 830](#_Toc444155983)

[10.11 ALL\_POLICIES 831](#_Toc444155984)

[10.12 ALL\_SEQUENCES 832](#_Toc444155985)

[10.13 ALL\_SOURCE 832](#_Toc444155986)

[10.14 ALL\_SUBPART\_KEY\_COLUMNS 833](#_Toc444155987)

[10.15 ALL\_SYNONYMS 833](#_Toc444155988)

[10.16 ALL\_TAB\_COLUMNS 834](#_Toc444155989)

[10.17 ALL\_TAB\_PARTITIONS 835](#_Toc444155990)

[10.18 ALL\_TAB\_SUBPARTITIONS 836](#_Toc444155991)

[10.19 ALL\_TABLES 837](#_Toc444155992)

[10.20 ALL\_TRIGGERS 837](#_Toc444155993)

[10.21 ALL\_TYPES 838](#_Toc444155994)

[10.22 ALL\_USERS 838](#_Toc444155995)

[10.23 ALL\_VIEW\_COLUMNS 839](#_Toc444155996)

[10.24 ALL\_VIEWS 839](#_Toc444155997)

[10.25 DBA\_ALL\_TABLES 840](#_Toc444155998)

[10.26 DBA\_CONS\_COLUMNS 840](#_Toc444155999)

[10.27 DBA\_CONSTRAINTS 841](#_Toc444156000)

[10.28 DBA\_DB\_LINKS 841](#_Toc444156001)

[10.29 DBA\_IND\_COLUMNS 842](#_Toc444156002)

[10.30 DBA\_INDEXES 842](#_Toc444156003)

[10.31 DBA\_JOBS 843](#_Toc444156004)

[10.32 DBA\_OBJECTS 843](#_Toc444156005)

[10.33 DBA\_PART\_KEY\_COLUMNS 844](#_Toc444156006)

[10.34 DBA\_PART\_TABLES 845](#_Toc444156007)

[10.35 DBA\_POLICIES 846](#_Toc444156008)

[10.36 DBA\_PROFILES 847](#_Toc444156009)

[10.37 DBA\_ROLE\_PRIVS 847](#_Toc444156010)

[10.38 DBA\_ROLES 847](#_Toc444156011)

[10.39 DBA\_SEQUENCES 848](#_Toc444156012)

[10.40 DBA\_SOURCE 848](#_Toc444156013)

[10.41 DBA\_SUBPART\_KEY\_COLUMNS 848](#_Toc444156014)

[10.42 DBA\_SYNONYMS 849](#_Toc444156015)

[10.43 DBA\_TAB\_COLUMNS 849](#_Toc444156016)

[10.44 DBA\_TAB\_PARTITIONS 850](#_Toc444156017)

[10.45 DBA\_TAB\_SUBPARTITIONS 851](#_Toc444156018)

[10.46 DBA\_TABLES 852](#_Toc444156019)

[10.47 DBA\_TRIGGERS 852](#_Toc444156020)

[10.48 DBA\_TYPES 853](#_Toc444156021)

[10.49 DBA\_USERS 854](#_Toc444156022)

[10.50 DBA\_VIEW\_COLUMNS 855](#_Toc444156023)

[10.51 DBA\_VIEWS 855](#_Toc444156024)

[10.52 USER\_ALL\_TABLES 856](#_Toc444156025)

[10.53 USER\_CONS\_COLUMNS 856](#_Toc444156026)

[10.54 USER\_CONSTRAINTS 857](#_Toc444156027)

[10.55 USER\_DB\_LINKS 857](#_Toc444156028)

[10.56 USER\_IND\_COLUMNS 858](#_Toc444156029)

[10.57 USER\_INDEXES 858](#_Toc444156030)

[10.58 USER\_JOBS 859](#_Toc444156031)

[10.59 USER\_OBJECTS 859](#_Toc444156032)

[10.60 USER\_PART\_KEY\_COLUMNS 860](#_Toc444156033)

[10.61 USER\_PART\_TABLES 861](#_Toc444156034)

[10.62 USER\_POLICIES 862](#_Toc444156035)

[10.63 USER\_ROLE\_PRIVS 863](#_Toc444156036)

[10.64 USER\_SEQUENCES 863](#_Toc444156037)

[10.65 USER\_SOURCE 864](#_Toc444156038)

[10.66 USER\_SUBPART\_KEY\_COLUMNS 864](#_Toc444156039)

[10.67 USER\_SYNONYMS 864](#_Toc444156040)

[10.68 USER\_TAB\_COLUMNS 865](#_Toc444156041)

[10.69 USER\_TAB\_PARTITIONS 866](#_Toc444156042)

[10.70 USER\_TAB\_SUBPARTITIONS 867](#_Toc444156043)

[10.71 USER\_TABLES 868](#_Toc444156044)

[10.72 USER\_TRIGGERS 868](#_Toc444156045)

[10.73 USER\_TYPES 869](#_Toc444156046)

[10.74 USER\_USERS 869](#_Toc444156047)

[10.75 USER\_VIEW\_COLUMNS 870](#_Toc444156048)

[10.76 USER\_VIEWS 870](#_Toc444156049)

[10.77 V$VERSION 870](#_Toc444156050)

[10.78 PRODUCT\_COMPONENT\_VERSION 871](#_Toc444156051)

[11 System Catalog Tables 872](#_Toc444156052)

[11.1 dual 872](#_Toc444156053)

[11.2 edb\_dir 872](#_Toc444156054)

[11.3 edb\_all\_resource\_groups 873](#_Toc444156055)

[11.4 edb\_partdef 873](#_Toc444156056)

[11.5 edb\_partition 874](#_Toc444156057)

[11.6 edb\_password\_history 874](#_Toc444156058)

[11.7 edb\_policy 875](#_Toc444156059)

[11.8 edb\_profile 876](#_Toc444156060)

[11.9 edb\_resource\_group 877](#_Toc444156061)

[11.10 edb\_variable 877](#_Toc444156062)

[11.11 pg\_synonym 878](#_Toc444156063)

[11.12 product\_component\_version 878](#_Toc444156064)

[12 Appendix 879](#_Toc444156065)

[12.1 Advanced Server Database Limits 879](#_Toc444156066)

[12.2 Advanced Server Keywords 879](#_Toc444156067)

# Introduction

*Notice: The names for EDB™’s products have changed. The product formerly referred to as Postgres Plus Advanced Server is now referred to as EDB Postgres™ Advanced Server. Until a new version of this documentation is published, wherever you see Postgres Plus Advanced Server you may substitute it with EDB Postgres Advanced Server. Name changes in software and software outputs will be phased in over time.*

This guide describes the features of EnterpriseDB’s EDB Postgres Enterprise, formerly known as EDB Postgres Plus Enterprise Edition. The core of EDB Postgres Enterprise is EnterpriseDB’s database server, EDB Postgres Advanced Server.

Enterprise provides a wide range of additional functionality in various areas including database administration, enhanced SQL capabilities, database and application security, performance monitoring and analysis, and application development utilities.

This guide is arranged as follows:

* Database Administration. Chapter 2 contains the features related to database administration.

Configuration parameters described in Section 2.1 control the basic characteristics and performance of an Advanced Server instance.  
  
Audit logging described in Section 2.2 provides enhanced database auditing capabilities.

Unicode Collation Algorithm described in Section 2.3 provides the capability to create a collation specific to your particular needs on a UTF-8 encoded database.

* Enhanced SQL Features. Chapter 3 contains the SQL enhancements provided for an Advanced Server database.

Synonyms described in Section 3.1 provide for easy-to-use abbreviations for the fully qualified path names of tables and views.

Hierarchical queries described in Section 3.2 provide for a logical display of tables related by foreign key constraints.

Extended functions and operators described in Section 3.3 provides for additional functionality of SQL.  
  
Partitioned tables described in Section 3.4 provide for the implementation of table partitioning using the SQL CREATE TABLE statement.

* Security. Chapter 4 contains various security features.

SQL/Protect described in Section 4.1 provides protection against SQL injection attacks.  
  
EDB\*Wrap described in Section 4.2 provides obfuscation of program source code to prevent unwanted scrutiny.  
  
Virtual Private Database described in Section 4.3 provides fine-grained, row level access.

* EDB Resource Manager. Chapter 5 contains information on the EDB Resource Manager feature, which provides the capability to control system resource usage by Advanced Server processes.

Resource Groups described in Section 5.1 shows how to create and maintain the groups on which resource limits can be defined and to which Advanced Server processes can be assigned.

CPU Usage Throttling described in Section 5.2 provides a method to control CPU usage by Advanced Server processes.

Dirty Buffer Throttling described in Section 5.3 provides a method to control the dirty rate of shared buffers by Advanced Server processes.

* Database Utilities. Chapter 6 contains database utility programs and interfaces.

EDB\*Loader described in Section 6.1 provides a quick and easy method for loading Advanced Server tables.  
  
EDB\*Plus described in Section 6.2 is a command line utility program for running SQL statements.

The libpq C library described in Section 6.3 is the C application programming interface (API) language for Advanced Server.

ECPGPlus described in Section 6.4 is a C precompiler for Advanced Server.

* Open Client Library. Chapter 7 provides information about the Open Client Library, an application programming interface for Advanced Server.  
    
  The PL Debugger described in Section 7.5 is a graphically oriented debugging tool for PL/pgSQL.
* Performance Analysis and Tuning. Chapter 8 contains the various tools for analyzing and improving application and database server performance.

Dynatune described in Section 8.1 provides a quick and easy means for configuring Advanced Server depending upon the type of application usage.  
  
Infinite Cache described in Section 8.2 provides for performance improvement using memory caching. Note: Infinite Cache has been deprecated and may be removed in a future release. Please contact your EnterpriseDB Account Manager or <mailto:sales@enterprisedb.com> for more information.  
  
Index Advisor described in Section 8.3 helps to determine the additional indexes needed on tables to improve application performance.  
  
SQL Profiler described in Section 8.4 locates and diagnoses poorly running SQL queries in applications.  
  
Query optimization hints described in Section 8.5 allows you to influence the manner in which the query planner processes SQL statements.  
  
DBMS\_PROFILER described in Section 8.6 is a built-in package that can be used to gather performance statistics for PL/pgSQL programs.  
  
Dynamic Runtime Instrumentation Tools Architecture (DRITA) described in Section 8.7 provides the capability to capture and view statistics pertaining to wait events that affect system performance.

* Built-In Utility Packages. Chapter 9 contains an extensive set of built-in packages that provide functions to quicken and ease development of PL/pgSQL applications.
* Expanded Catalog Views. Chapter 10 contains additional catalog views added to Advanced Server to simplify the querying of database object information.
* System Catalog Tables. Chapter 11 contains additional system catalog tables added for Advanced Server specific database objects.
* Appendix. Chapter 12 contains various miscellaneous topics such as Advanced Server database limits and keywords.

## What’s New

The following features have been added to EDB Postgres Advanced Server 9.4 to create Advanced Server 9.5:

* Advanced Server now provides support for Profile Management. For more information, see Section 2.4.
* Advanced Server now includes support for DBMS\_SESSION.SET\_ROLE. For more information, see Section 9.13.1.
* Advanced Server now includes support for the UTL\_RAW package. For more information, see Section 9.20.
* Advanced Server now includes the edb\_audit\_tag parameter; the parameter can be used to add a tag to an audit log. For more information, see sections 2.1.3.7.10 and 2.2.1.
* Advanced Server supports the use of EDBLDR\_ENV\_STYLE to specify the style of environment variables recognized by EDB\*Loader. For more information, see Section 6.1.3.
* EDB\*Loader now accepts the ZONED [(precision[,scale])] field type specification. For more information, see Section 6.1.3.
* Advanced Server now supports the UTL\_HTTP.WRITE\_LINE and UTL\_HTTP.WRITE\_TEXT procedures. For more information, see sections 9.18.25 and 9.18.27, respectively.
* Advanced Server now supports the DBA\_PROFILES view. For more information, see Section 10.36.
* Advanced Server now supports the FREEZE keyword in the EDB\*Loader control file and on the command line. For more information, see sections 6.1.3 and 6.1.4, respectively.
* Advanced Server now supports XA functions xaoEnv and xaoSvcCtx in the Open Client Library. For more information, See Section 7.4.6.
* Advanced Server now supports the EDB\_ATTR\_EMPTY\_STRINGS environment attribute in the Open Client Library. For more information, See Section 7.4.2.1.

## Typographical Conventions Used in this Guide

Certain typographical conventions are used in this manual to clarify the meaning and usage of various commands, statements, programs, examples, etc. This section provides a summary of these conventions.

In the following descriptions a term refers to any word or group of words that may be language keywords, user-supplied values, literals, etc. A term’s exact meaning depends upon the context in which it is used.

* Italic font introduces a new term, typically, in the sentence that defines it for the first time.
* Fixed-width (mono-spaced) font is used for terms that must be given literally such as SQL commands, specific table and column names used in the examples, programming language keywords, directory paths and file names, parameter values, etc. For example postgresql.conf, SELECT \* FROM emp;
* Italic fixed-width font is used for terms for which the user must substitute values in actual usage. For example, DELETE FROM table\_name;
* A vertical pipe | denotes a choice between the terms on either side of the pipe. A vertical pipe is used to separate two or more alternative terms within square brackets (optional choices) or braces (one mandatory choice).
* Square brackets [ ] denote that one or none of the enclosed term(s) may be substituted. For example, [ a | b ], means choose one of “a” or “b” or neither of the two.
* Braces {} denote that exactly one of the enclosed alternatives must be specified. For example, { a | b }, means exactly one of “a” or “b” must be specified.
* Ellipses ... denote that the proceeding term may be repeated. For example, [ a | b ] ... means that you may have the sequence, “b a a b a”.

## Other Conventions Used in this Guide

This guide applies to both Linux and Windows systems. Directory paths are presented in the Linux format with forward slashes. When working on Windows systems, start the directory path with the drive letter followed by a colon and substitute back slashes for forward slashes.

Throughout this guide, the directory path of the Advanced Server is referred to as POSTGRES\_PLUS\_HOME.

For Linux installations, the default directory path is

/opt/PostgresPlus/version\_no

For Windows installations, the default directory path is

C:\Program Files\PostgresPlus\version\_no

The product version number is represented by version\_no.

## About the Examples Used in this Guide

The examples in this guide are shown in the type and background illustrated below.

Examples and output from examples are shown in fixed-width, blue font on a light blue background.

The examples use the sample tables, dept, emp, and jobhist, created and loaded when Advanced Server is installed.

The tables and programs in the sample database can be re-created at any time by executing the following script:

POSTGRES\_PLUS\_HOME/installer/server/pg-sample.sql.

The script:

* Creates the sample tables and programs in the currently connected database.
* Grants all permissions on the tables to the PUBLIC group.

The tables and programs will be created in the first schema of the search path in which the current user has permission to create tables and procedures. You can display the search path by issuing the command:

SHOW SEARCH\_PATH;

You can use PSQL commands to modify the search path.

#### Sample Database Description

The sample database represents employees in an organization. It contains three types of records: employees, departments, and historical records of employees.

Each employee has an identification number, name, hire date, salary, and manager. Some employees earn a commission in addition to their salary. All employee-related information is stored in the emp table.

The sample company is regionally diverse, so it tracks the locations of its departments. Each company employee is assigned to a department. Each department is identified by a unique department number and a short name. Each department is associated with one location. All department-related information is stored in the dept table.

The company also tracks information about jobs held by the employees. Some employees have been with the company for a long time and have held different positions, received raises, switched departments, etc. When a change in employee status occurs, the company records the end date of the former position. A new job record is added with the start date and the new job title, department, salary, and the reason for the status change. All employee history is maintained in the jobhist table.

The following is the pg-sample.sql script:

SET datestyle TO 'iso, dmy';

--

-- Script that creates the 'sample' tables, views

-- functions, triggers, etc.

--

-- Start new transaction - commit all or nothing

--

BEGIN;

--

-- Create and load tables used in the documentation examples.

--

-- Create the 'dept' table

--

CREATE TABLE dept (

deptno NUMERIC(2) NOT NULL CONSTRAINT dept\_pk PRIMARY KEY,

dname VARCHAR(14) CONSTRAINT dept\_dname\_uq UNIQUE,

loc VARCHAR(13)

);

--

-- Create the 'emp' table

--

CREATE TABLE emp (

empno NUMERIC(4) NOT NULL CONSTRAINT emp\_pk PRIMARY KEY,

ename VARCHAR(10),

job VARCHAR(9),

mgr NUMERIC(4),

hiredate DATE,

sal NUMERIC(7,2) CONSTRAINT emp\_sal\_ck CHECK (sal > 0),

comm NUMERIC(7,2),

deptno NUMERIC(2) CONSTRAINT emp\_ref\_dept\_fk

REFERENCES dept(deptno)

);

--

-- Create the 'jobhist' table

--

CREATE TABLE jobhist (

empno NUMERIC(4) NOT NULL,

startdate TIMESTAMP(0) NOT NULL,

enddate TIMESTAMP(0),

job VARCHAR(9),

sal NUMERIC(7,2),

comm NUMERIC(7,2),

deptno NUMERIC(2),

chgdesc VARCHAR(80),

CONSTRAINT jobhist\_pk PRIMARY KEY (empno, startdate),

CONSTRAINT jobhist\_ref\_emp\_fk FOREIGN KEY (empno)

REFERENCES emp(empno) ON DELETE CASCADE,

CONSTRAINT jobhist\_ref\_dept\_fk FOREIGN KEY (deptno)

REFERENCES dept (deptno) ON DELETE SET NULL,

CONSTRAINT jobhist\_date\_chk CHECK (startdate <= enddate)

);

--

-- Create the 'salesemp' view

--

CREATE OR REPLACE VIEW salesemp AS

SELECT empno, ename, hiredate, sal, comm FROM emp WHERE job = 'SALESMAN';

--

-- Sequence to generate values for function 'new\_empno'.

--

CREATE SEQUENCE next\_empno START WITH 8000 INCREMENT BY 1;

--

-- Issue PUBLIC grants

--

--GRANT ALL ON emp TO PUBLIC;

--GRANT ALL ON dept TO PUBLIC;

--GRANT ALL ON jobhist TO PUBLIC;

--GRANT ALL ON salesemp TO PUBLIC;

--GRANT ALL ON next\_empno TO PUBLIC;

--

-- Load the 'dept' table

--

INSERT INTO dept VALUES (10,'ACCOUNTING','NEW YORK');

INSERT INTO dept VALUES (20,'RESEARCH','DALLAS');

INSERT INTO dept VALUES (30,'SALES','CHICAGO');

INSERT INTO dept VALUES (40,'OPERATIONS','BOSTON');

--

-- Load the 'emp' table

--

INSERT INTO emp VALUES (7369,'SMITH','CLERK',7902,'17-DEC-80',800,NULL,20);

INSERT INTO emp VALUES (7499,'ALLEN','SALESMAN',7698,'20-FEB-81',1600,300,30);

INSERT INTO emp VALUES (7521,'WARD','SALESMAN',7698,'22-FEB-81',1250,500,30);

INSERT INTO emp VALUES (7566,'JONES','MANAGER',7839,'02-APR-81',2975,NULL,20);

INSERT INTO emp VALUES (7654,'MARTIN','SALESMAN',7698,'28-SEP-81',1250,1400,30);

INSERT INTO emp VALUES (7698,'BLAKE','MANAGER',7839,'01-MAY-81',2850,NULL,30);

INSERT INTO emp VALUES (7782,'CLARK','MANAGER',7839,'09-JUN-81',2450,NULL,10);

INSERT INTO emp VALUES (7788,'SCOTT','ANALYST',7566,'19-APR-87',3000,NULL,20);

INSERT INTO emp VALUES (7839,'KING','PRESIDENT',NULL,'17-NOV-81',5000,NULL,10);

INSERT INTO emp VALUES (7844,'TURNER','SALESMAN',7698,'08-SEP-81',1500,0,30);

INSERT INTO emp VALUES (7876,'ADAMS','CLERK',7788,'23-MAY-87',1100,NULL,20);

INSERT INTO emp VALUES (7900,'JAMES','CLERK',7698,'03-DEC-81',950,NULL,30);

INSERT INTO emp VALUES (7902,'FORD','ANALYST',7566,'03-DEC-81',3000,NULL,20);

INSERT INTO emp VALUES (7934,'MILLER','CLERK',7782,'23-JAN-82',1300,NULL,10);

--

-- Load the 'jobhist' table

--

INSERT INTO jobhist VALUES (7369,'17-DEC-80',NULL,'CLERK',800,NULL,20,'New Hire');

INSERT INTO jobhist VALUES (7499,'20-FEB-81',NULL,'SALESMAN',1600,300,30,'New Hire');

INSERT INTO jobhist VALUES (7521,'22-FEB-81',NULL,'SALESMAN',1250,500,30,'New Hire');

INSERT INTO jobhist VALUES (7566,'02-APR-81',NULL,'MANAGER',2975,NULL,20,'New Hire');

INSERT INTO jobhist VALUES (7654,'28-SEP-81',NULL,'SALESMAN',1250,1400,30,'New Hire');

INSERT INTO jobhist VALUES (7698,'01-MAY-81',NULL,'MANAGER',2850,NULL,30,'New Hire');

INSERT INTO jobhist VALUES (7782,'09-JUN-81',NULL,'MANAGER',2450,NULL,10,'New Hire');

INSERT INTO jobhist VALUES (7788,'19-APR-87','12-APR-88','CLERK',1000,NULL,20,'New Hire');

INSERT INTO jobhist VALUES (7788,'13-APR-88','04-MAY-89','CLERK',1040,NULL,20,'Raise');

INSERT INTO jobhist VALUES (7788,'05-MAY-90',NULL,'ANALYST',3000,NULL,20,'Promoted to Analyst');

INSERT INTO jobhist VALUES (7839,'17-NOV-81',NULL,'PRESIDENT',5000,NULL,10,'New Hire');

INSERT INTO jobhist VALUES (7844,'08-SEP-81',NULL,'SALESMAN',1500,0,30,'New Hire');

INSERT INTO jobhist VALUES (7876,'23-MAY-87',NULL,'CLERK',1100,NULL,20,'New Hire');

INSERT INTO jobhist VALUES (7900,'03-DEC-81','14-JAN-83','CLERK',950,NULL,10,'New Hire');

INSERT INTO jobhist VALUES (7900,'15-JAN-83',NULL,'CLERK',950,NULL,30,'Changed to Dept 30');

INSERT INTO jobhist VALUES (7902,'03-DEC-81',NULL,'ANALYST',3000,NULL,20,'New Hire');

INSERT INTO jobhist VALUES (7934,'23-JAN-82',NULL,'CLERK',1300,NULL,10,'New Hire');

--

-- Populate statistics table and view (pg\_statistic/pg\_stats)

--

ANALYZE dept;

ANALYZE emp;

ANALYZE jobhist;

--

-- Function that lists all employees' numbers and names

-- from the 'emp' table using a cursor.

--

CREATE OR REPLACE FUNCTION list\_emp() RETURNS VOID

AS $$

DECLARE

v\_empno NUMERIC(4);

v\_ename VARCHAR(10);

emp\_cur CURSOR FOR

SELECT empno, ename FROM emp ORDER BY empno;

BEGIN

OPEN emp\_cur;

RAISE INFO 'EMPNO ENAME';

RAISE INFO '----- -------';

LOOP

FETCH emp\_cur INTO v\_empno, v\_ename;

EXIT WHEN NOT FOUND;

RAISE INFO '% %', v\_empno, v\_ename;

END LOOP;

CLOSE emp\_cur;

RETURN;

END;

$$ LANGUAGE 'plpgsql';

--

-- Function that selects an employee row given the employee

-- number and displays certain columns.

--

CREATE OR REPLACE FUNCTION select\_emp (

p\_empno NUMERIC

) RETURNS VOID

AS $$

DECLARE

v\_ename emp.ename%TYPE;

v\_hiredate emp.hiredate%TYPE;

v\_sal emp.sal%TYPE;

v\_comm emp.comm%TYPE;

v\_dname dept.dname%TYPE;

v\_disp\_date VARCHAR(10);

BEGIN

SELECT INTO

v\_ename, v\_hiredate, v\_sal, v\_comm, v\_dname

ename, hiredate, sal, COALESCE(comm, 0), dname

FROM emp e, dept d

WHERE empno = p\_empno

AND e.deptno = d.deptno;

IF NOT FOUND THEN

RAISE INFO 'Employee % not found', p\_empno;

RETURN;

END IF;

v\_disp\_date := TO\_CHAR(v\_hiredate, 'MM/DD/YYYY');

RAISE INFO 'Number : %', p\_empno;

RAISE INFO 'Name : %', v\_ename;

RAISE INFO 'Hire Date : %', v\_disp\_date;

RAISE INFO 'Salary : %', v\_sal;

RAISE INFO 'Commission: %', v\_comm;

RAISE INFO 'Department: %', v\_dname;

RETURN;

EXCEPTION

WHEN OTHERS THEN

RAISE INFO 'The following is SQLERRM : %', SQLERRM;

RAISE INFO 'The following is SQLSTATE: %', SQLSTATE;

RETURN;

END;

$$ LANGUAGE 'plpgsql';

--

-- A RECORD type used to format the return value of

-- function, 'emp\_query'.

--

CREATE TYPE emp\_query\_type AS (

empno NUMERIC,

ename VARCHAR(10),

job VARCHAR(9),

hiredate DATE,

sal NUMERIC

);

--

-- Function that queries the 'emp' table based on

-- department number and employee number or name. Returns

-- employee number and name as INOUT parameters and job,

-- hire date, and salary as OUT parameters. These are

-- returned in the form of a record defined by

-- RECORD type, 'emp\_query\_type'.

--

CREATE OR REPLACE FUNCTION emp\_query (

IN p\_deptno NUMERIC,

INOUT p\_empno NUMERIC,

INOUT p\_ename VARCHAR,

OUT p\_job VARCHAR,

OUT p\_hiredate DATE,

OUT p\_sal NUMERIC

)

AS $$

BEGIN

SELECT INTO

p\_empno, p\_ename, p\_job, p\_hiredate, p\_sal

empno, ename, job, hiredate, sal

FROM emp

WHERE deptno = p\_deptno

AND (empno = p\_empno

OR ename = UPPER(p\_ename));

END;

$$ LANGUAGE 'plpgsql';

--

-- Function to call 'emp\_query\_caller' with IN and INOUT

-- parameters. Displays the results received from INOUT and

-- OUT parameters.

--

CREATE OR REPLACE FUNCTION emp\_query\_caller() RETURNS VOID

AS $$

DECLARE

v\_deptno NUMERIC;

v\_empno NUMERIC;

v\_ename VARCHAR;

v\_rows INTEGER;

r\_emp\_query EMP\_QUERY\_TYPE;

BEGIN

v\_deptno := 30;

v\_empno := 0;

v\_ename := 'Martin';

r\_emp\_query := emp\_query(v\_deptno, v\_empno, v\_ename);

RAISE INFO 'Department : %', v\_deptno;

RAISE INFO 'Employee No: %', (r\_emp\_query).empno;

RAISE INFO 'Name : %', (r\_emp\_query).ename;

RAISE INFO 'Job : %', (r\_emp\_query).job;

RAISE INFO 'Hire Date : %', (r\_emp\_query).hiredate;

RAISE INFO 'Salary : %', (r\_emp\_query).sal;

RETURN;

EXCEPTION

WHEN OTHERS THEN

RAISE INFO 'The following is SQLERRM : %', SQLERRM;

RAISE INFO 'The following is SQLSTATE: %', SQLSTATE;

RETURN;

END;

$$ LANGUAGE 'plpgsql';

--

-- Function to compute yearly compensation based on semimonthly

-- salary.

--

CREATE OR REPLACE FUNCTION emp\_comp (

p\_sal NUMERIC,

p\_comm NUMERIC

) RETURNS NUMERIC

AS $$

BEGIN

RETURN (p\_sal + COALESCE(p\_comm, 0)) \* 24;

END;

$$ LANGUAGE 'plpgsql';

--

-- Function that gets the next number from sequence, 'next\_empno',

-- and ensures it is not already in use as an employee number.

--

CREATE OR REPLACE FUNCTION new\_empno() RETURNS INTEGER

AS $$

DECLARE

v\_cnt INTEGER := 1;

v\_new\_empno INTEGER;

BEGIN

WHILE v\_cnt > 0 LOOP

SELECT INTO v\_new\_empno nextval('next\_empno');

SELECT INTO v\_cnt COUNT(\*) FROM emp WHERE empno = v\_new\_empno;

END LOOP;

RETURN v\_new\_empno;

END;

$$ LANGUAGE 'plpgsql';

--

-- Function that adds a new clerk to table 'emp'.

--

CREATE OR REPLACE FUNCTION hire\_clerk (

p\_ename VARCHAR,

p\_deptno NUMERIC

) RETURNS NUMERIC

AS $$

DECLARE

v\_empno NUMERIC(4);

v\_ename VARCHAR(10);

v\_job VARCHAR(9);

v\_mgr NUMERIC(4);

v\_hiredate DATE;

v\_sal NUMERIC(7,2);

v\_comm NUMERIC(7,2);

v\_deptno NUMERIC(2);

BEGIN

v\_empno := new\_empno();

INSERT INTO emp VALUES (v\_empno, p\_ename, 'CLERK', 7782,

CURRENT\_DATE, 950.00, NULL, p\_deptno);

SELECT INTO

v\_empno, v\_ename, v\_job, v\_mgr, v\_hiredate, v\_sal, v\_comm, v\_deptno

empno, ename, job, mgr, hiredate, sal, comm, deptno

FROM emp WHERE empno = v\_empno;

RAISE INFO 'Department : %', v\_deptno;

RAISE INFO 'Employee No: %', v\_empno;

RAISE INFO 'Name : %', v\_ename;

RAISE INFO 'Job : %', v\_job;

RAISE INFO 'Manager : %', v\_mgr;

RAISE INFO 'Hire Date : %', v\_hiredate;

RAISE INFO 'Salary : %', v\_sal;

RAISE INFO 'Commission : %', v\_comm;

RETURN v\_empno;

EXCEPTION

WHEN OTHERS THEN

RAISE INFO 'The following is SQLERRM : %', SQLERRM;

RAISE INFO 'The following is SQLSTATE: %', SQLSTATE;

RETURN -1;

END;

$$ LANGUAGE 'plpgsql';

--

-- Function that adds a new salesman to table 'emp'.

--

CREATE OR REPLACE FUNCTION hire\_salesman (

p\_ename VARCHAR,

p\_sal NUMERIC,

p\_comm NUMERIC

) RETURNS NUMERIC

AS $$

DECLARE

v\_empno NUMERIC(4);

v\_ename VARCHAR(10);

v\_job VARCHAR(9);

v\_mgr NUMERIC(4);

v\_hiredate DATE;

v\_sal NUMERIC(7,2);

v\_comm NUMERIC(7,2);

v\_deptno NUMERIC(2);

BEGIN

v\_empno := new\_empno();

INSERT INTO emp VALUES (v\_empno, p\_ename, 'SALESMAN', 7698,

CURRENT\_DATE, p\_sal, p\_comm, 30);

SELECT INTO

v\_empno, v\_ename, v\_job, v\_mgr, v\_hiredate, v\_sal, v\_comm, v\_deptno

empno, ename, job, mgr, hiredate, sal, comm, deptno

FROM emp WHERE empno = v\_empno;

RAISE INFO 'Department : %', v\_deptno;

RAISE INFO 'Employee No: %', v\_empno;

RAISE INFO 'Name : %', v\_ename;

RAISE INFO 'Job : %', v\_job;

RAISE INFO 'Manager : %', v\_mgr;

RAISE INFO 'Hire Date : %', v\_hiredate;

RAISE INFO 'Salary : %', v\_sal;

RAISE INFO 'Commission : %', v\_comm;

RETURN v\_empno;

EXCEPTION

WHEN OTHERS THEN

RAISE INFO 'The following is SQLERRM : %', SQLERRM;

RAISE INFO 'The following is SQLSTATE: %', SQLSTATE;

RETURN -1;

END;

$$ LANGUAGE 'plpgsql';

--

-- Rule to INSERT into view 'salesemp'

--

CREATE OR REPLACE RULE salesemp\_i AS ON INSERT TO salesemp

DO INSTEAD

INSERT INTO emp VALUES (NEW.empno, NEW.ename, 'SALESMAN', 7698,

NEW.hiredate, NEW.sal, NEW.comm, 30);

--

-- Rule to UPDATE view 'salesemp'

--

CREATE OR REPLACE RULE salesemp\_u AS ON UPDATE TO salesemp

DO INSTEAD

UPDATE emp SET empno = NEW.empno,

ename = NEW.ename,

hiredate = NEW.hiredate,

sal = NEW.sal,

comm = NEW.comm

WHERE empno = OLD.empno;

--

-- Rule to DELETE from view 'salesemp'

--

CREATE OR REPLACE RULE salesemp\_d AS ON DELETE TO salesemp

DO INSTEAD

DELETE FROM emp WHERE empno = OLD.empno;

--

-- After statement-level trigger that displays a message after

-- an insert, update, or deletion to the 'emp' table. One message

-- per SQL command is displayed.

--

CREATE OR REPLACE FUNCTION user\_audit\_trig() RETURNS TRIGGER

AS $$

DECLARE

v\_action VARCHAR(24);

v\_text TEXT;

BEGIN

IF TG\_OP = 'INSERT' THEN

v\_action := ' added employee(s) on ';

ELSIF TG\_OP = 'UPDATE' THEN

v\_action := ' updated employee(s) on ';

ELSIF TG\_OP = 'DELETE' THEN

v\_action := ' deleted employee(s) on ';

END IF;

v\_text := 'User ' || USER || v\_action || CURRENT\_DATE;

RAISE INFO ' %', v\_text;

RETURN NULL;

END;

$$ LANGUAGE 'plpgsql';

CREATE TRIGGER user\_audit\_trig

AFTER INSERT OR UPDATE OR DELETE ON emp

FOR EACH STATEMENT EXECUTE PROCEDURE user\_audit\_trig();

--

-- Before row-level trigger that displays employee number and

-- salary of an employee that is about to be added, updated,

-- or deleted in the 'emp' table.

--

CREATE OR REPLACE FUNCTION emp\_sal\_trig() RETURNS TRIGGER

AS $$

DECLARE

sal\_diff NUMERIC(7,2);

BEGIN

IF TG\_OP = 'INSERT' THEN

RAISE INFO 'Inserting employee %', NEW.empno;

RAISE INFO '..New salary: %', NEW.sal;

RETURN NEW;

END IF;

IF TG\_OP = 'UPDATE' THEN

sal\_diff := NEW.sal - OLD.sal;

RAISE INFO 'Updating employee %', OLD.empno;

RAISE INFO '..Old salary: %', OLD.sal;

RAISE INFO '..New salary: %', NEW.sal;

RAISE INFO '..Raise : %', sal\_diff;

RETURN NEW;

END IF;

IF TG\_OP = 'DELETE' THEN

RAISE INFO 'Deleting employee %', OLD.empno;

RAISE INFO '..Old salary: %', OLD.sal;

RETURN OLD;

END IF;

END;

$$ LANGUAGE 'plpgsql';

CREATE TRIGGER emp\_sal\_trig

BEFORE DELETE OR INSERT OR UPDATE ON emp

FOR EACH ROW EXECUTE PROCEDURE emp\_sal\_trig();

COMMIT;

# Database Administration

This chapter describes the features that aid in the management and administration of Advanced Server databases.

## Configuration Parameters

This section describes the database server configuration parameters of Advanced Server. These parameters control various aspects of the database server’s behavior and environment such as data file and log file locations, connection, authentication, and security settings, resource allocation and consumption, archiving and replication settings, error logging and statistics gathering, optimization and performance tuning, locale and formatting settings, and so on.

Most of these configuration parameters apply to PostgreSQL as well. Configuration parameters that apply only to Advanced Server are noted in Section 2.1.2.

Additional information about configuration parameters can be found in the PostgreSQL Core Documentation available at:

<http://www.postgresql.org/docs/9.5/static/index.html>

### Setting Configuration Parameters

This section provides an overview of how configuration parameters are specified and set.

Each configuration parameter is set using a name/value pair. Parameter names are case-insensitive. The parameter name is typically separated from its value by an optional equals sign (=).

The following is an example of some configuration parameter settings in the postgresql.conf file:

# This is a comment

log\_connections = yes

log\_destination = 'syslog'

search\_path = '"$user", public'

shared\_buffers = 128MB

Parameter values are specified as one of five types:

* Boolean. Acceptable values can be written as on, off, true, false, yes, no, 1, 0, or any unambiguous prefix of these.
* Integer. Number without a fractional part.
* Floating Point. Number with an optional fractional part separated by a decimal point.
* String. Text value. Enclose in single quotes if the value is not a simple identifier or number (that is, the value contains special characters such as spaces or other punctuation marks).
* Enum. Specific set of string values. The allowed values can be found in the system view pg\_settings.enumvals. Enum values are case-insensitive.

Some settings specify a memory or time value. Each of these has an implicit unit, which is kilobytes, blocks (typically 8 kilobytes), milliseconds, seconds, or minutes. Default units can be found by referencing the system view pg\_settings.unit. A different unit can be specified explicitly.

Valid memory units are kB (kilobytes), MB (megabytes), and GB (gigabytes). Valid time units are ms (milliseconds), s (seconds), min (minutes), h (hours), and d (days). The multiplier for memory units is 1024.

The configuration parameter settings can be established in a number of different ways:

* There is a number of parameter settings that are established when the Advanced Server database product is built. These are read-only parameters, and their values cannot be changed. There are also a couple of parameters that are permanently set for each database when the database is created. These parameters are read-only as well and cannot be subsequently changed for the database.
* The initial settings for almost all configurable parameters across the entire database cluster are listed in the configuration file, postgresql.conf. These settings are put into effect upon database server start or restart. Some of these initial parameter settings can be overridden as discussed in the following bullet points. All configuration parameters have built-in default settings that are in effect if not explicitly overridden.
* Parameter settings can be modified in the configuration file while the database server is running. If the configuration file is then reloaded (meaning a SIGHUP signal is issued), for certain parameter types, the changed parameters settings immediately take effect. For some of these parameter types, the new settings are available in a currently running session immediately after the reload. For other of these parameter types, a new session must be started to use the new settings. And yet for other parameter types, modified settings do not take effect until the database server is stopped and restarted. See Section 18.1, “Setting Parameters” in the PostgreSQL Core Documentation for information on how to reload the configuration file.
* The SQL commands ALTER DATABASE, ALTER ROLE, or ALTER ROLE IN DATABASE can be used to modify certain parameter settings. The modified parameter settings take effect for new sessions after the command is executed. ALTER DATABASE affects new sessions connecting to the specified database. ALTER ROLE affects new sessions started by the specified role. ALTER ROLE IN DATABASE affects new sessions started by the specified role connecting to the specified database. Parameter settings established by these SQL commands remain in effect indefinitely, across database server restarts, overriding settings established by the methods discussed in the second and third bullet points. Parameter settings established using the ALTER DATABASE, ALTER ROLE, or ALTER ROLE IN DATABASE commands can only be changed by: a) re-issuing these commands with a different parameter value, or b) issuing these commands using either of the SET parameter TO DEFAULT clause or the RESET parameter clause. These clauses change the parameter back to using the setting established by the methods set forth in the prior bullet points. See Section I, “SQL Commands” of Chapter VI “Reference” in the PostgreSQL Core Documentation for the exact syntax of these SQL commands.
* Changes can be made for certain parameter settings for the duration of individual sessions using the PGOPTIONS environment variable or by using the SET command within the EDB-PSQL or PSQL command line terminal programs. Parameter settings made in this manner override settings established using any of the methods described by the second, third, and fourth bullet points, but only for the duration of the session.

### Summary of Configuration Parameters

This section contains a summary table listing all Advanced Server configuration parameters along with a number of key attributes of the parameters.

These attributes are described by the following columns of the summary table:

* Parameter. Configuration parameter name.
* Scope of Effect. Scope of effect of the configuration parameter setting. ‘Cluster’ – Setting affects the entire database cluster (that is, all databases managed by the database server instance). ‘Database’ – Setting can vary by database and is established when the database is created. Applies to a small number of parameters related to locale settings. ‘Session’ – Setting can vary down to the granularity of individual sessions. In other words, different settings can be made for the following entities whereby the latter settings in this list override prior ones: a) the entire database cluster, b) specific databases in the database cluster, c) specific roles, d) specific roles when connected to specific databases, e) a specific session.
* When Takes Effect. When a changed parameter setting takes effect. ‘Preset’ – Established when the Advanced Server product is built or a particular database is created. This is a read-only parameter and cannot be changed. ‘Restart’ – Database server must be restarted. ‘Reload’ – Configuration file must be reloaded (or the database server can be restarted). ‘Immediate’ – Immediately effective in a session if the PGOPTIONS environment variable or the SET command is used to change the setting in the current session. Effective in new sessions if ALTER DATABASE, ALTER ROLE, or ALTER ROLE IN DATABASE commands are used to change the setting.
* Authorized User. Type of operating system account or database role that must be used to put the parameter setting into effect. ‘PPAS service account’ – EDB Postgres Advanced Server service account (enterprisedb for an installation compatible with Oracle databases, postgres for a PostgreSQL compatible mode installation). ‘Superuser’ – Database role with superuser privileges. ‘User’ – Any database role with permissions on the affected database object (the database or role to be altered with the ALTER command). ‘n/a’ – Parameter setting cannot be changed by any user.
* Description. Brief description of the configuration parameter.
* PPAS Only. ‘X’ – Configuration parameter is applicable to EDB Postgres Advanced Server only. No entry in this column indicates the configuration parameter applies to PostgreSQL as well.

Note: There are a number of parameters that should never be altered. These are designated as “Note: For internal use only” in the Description column.

Table ‑ - Summary of Configuration Parameters

| Parameter | Scope of Effect | When Takes Effect | Authorized  User | Description | PPAS Only |
| --- | --- | --- | --- | --- | --- |
| allow\_system\_table\_mods | Cluster | Restart | PPAS service account | Allows modifications of the structure of system tables. |  |
| application\_name | Session | Immediate | User | Sets the application name to be reported in statistics and logs. |  |
| archive\_command | Cluster | Reload | PPAS service account | Sets the shell command that will be called to archive a WAL file. |  |
| archive\_mode | Cluster | Restart | PPAS service account | Allows archiving of WAL files using archive\_command. |  |
| archive\_timeout | Cluster | Reload | PPAS service account | Forces a switch to the next xlog file if a new file has not been started within N seconds. |  |
| array\_nulls | Session | Immediate | User | Enable input of NULL elements in arrays. |  |
| authentication\_timeout | Cluster | Reload | PPAS service account | Sets the maximum allowed time to complete client authentication. |  |
| autovacuum | Cluster | Reload | PPAS service account | Starts the autovacuum subprocess. |  |
| autovacuum\_analyze\_scale\_factor | Cluster | Reload | PPAS service account | Number of tuple inserts, updates or deletes prior to analyze as a fraction of reltuples. |  |
| autovacuum\_analyze\_threshold | Cluster | Reload | PPAS service account | Minimum number of tuple inserts, updates or deletes prior to analyze. |  |
| autovacuum\_freeze\_max\_age | Cluster | Restart | PPAS service account | Age at which to autovacuum a table to prevent transaction ID wraparound. |  |
| autovacuum\_max\_workers | Cluster | Restart | PPAS service account | Sets the maximum number of simultaneously running autovacuum worker processes. |  |
| autovacuum\_multixact\_freeze\_max\_age | Cluster | Restart | PPAS service account | Multixact age at which to autovacuum a table to prevent multixact wraparound. |  |
| autovacuum\_naptime | Cluster | Reload | PPAS service account | Time to sleep between autovacuum runs. |  |
| autovacuum\_vacuum\_cost\_delay | Cluster | Reload | PPAS service account | Vacuum cost delay in milliseconds, for autovacuum. |  |
| autovacuum\_vacuum\_cost\_limit | Cluster | Reload | PPAS service account | Vacuum cost amount available before napping, for autovacuum. |  |
| autovacuum\_vacuum\_scale\_factor | Cluster | Reload | PPAS service account | Number of tuple updates or deletes prior to vacuum as a fraction of reltuples. |  |
| autovacuum\_vacuum\_threshold | Cluster | Reload | PPAS service account | Minimum number of tuple updates or deletes prior to vacuum. |  |
| autovacuum\_work\_mem | Cluster | Reload | PPAS service account | Sets the maximum memory to be used by each autovacuum worker process. |  |
| backslash\_quote | Session | Immediate | User | Sets whether "\'" is allowed in string literals. |  |
| bgwriter\_delay | Cluster | Reload | PPAS service account | Background writer sleep time between rounds. |  |
| bgwriter\_lru\_maxpages | Cluster | Reload | PPAS service account | Background writer maximum number of LRU pages to flush per round. |  |
| bgwriter\_lru\_multiplier | Cluster | Reload | PPAS service account | Multiple of the average buffer usage to free per round. |  |
| block\_size | Cluster | Preset | n/a | Shows the size of a disk block. |  |
| bonjour | Cluster | Restart | PPAS service account | Enables advertising the server via Bonjour. |  |
| bonjour\_name | Cluster | Restart | PPAS service account | Sets the Bonjour service name. |  |
| bytea\_output | Session | Immediate | User | Sets the output format for bytea. |  |
| check\_function\_bodies | Session | Immediate | User | Check function bodies during CREATE FUNCTION. |  |
| checkpoint\_completion\_target | Cluster | Reload | PPAS service account | Time spent flushing dirty buffers during checkpoint, as fraction of checkpoint interval. |  |
| checkpoint\_segments | Deprecated in 9.5 | Deprecated in 9.5 | Deprecated in 9.5 | This parameter is not supported by server version 9.5 or later. Specifying a value for the parameter will prevent the server from starting. |  |
| checkpoint\_timeout | Cluster | Reload | PPAS service account | Sets the maximum time between automatic WAL checkpoints. |  |
| checkpoint\_warning | Cluster | Reload | PPAS service account | Enables warnings if checkpoint segments are filled more frequently than this. |  |
| client\_encoding | Session | Immediate | User | Sets the client's character set encoding. |  |
| client\_min\_messages | Session | Immediate | User | Sets the message levels that are sent to the client. |  |
| commit\_delay | Session | Immediate | Superuser | Sets the delay in microseconds between transaction commit and flushing WAL to disk. |  |
| commit\_siblings | Session | Immediate | User | Sets the minimum concurrent open transactions before performing commit\_delay. |  |
| config\_file | Cluster | Restart | PPAS service account | Sets the server's main configuration file. |  |
| constraint\_exclusion | Session | Immediate | User | Enables the planner to use constraints to optimize queries. |  |
| cpu\_index\_tuple\_cost | Session | Immediate | User | Sets the planner's estimate of the cost of processing each index entry during an index scan. |  |
| cpu\_operator\_cost | Session | Immediate | User | Sets the planner's estimate of the cost of processing each operator or function call. |  |
| cpu\_tuple\_cost | Session | Immediate | User | Sets the planner's estimate of the cost of processing each tuple (row). |  |
| cursor\_tuple\_fraction | Session | Immediate | User | Sets the planner's estimate of the fraction of a cursor's rows that will be retrieved. |  |
| custom\_variable\_classes | Cluster | Reload | PPAS service account | Deprecated in Advanced Server 9.2. | X |
| data\_checksums | Cluster | Preset | n/a | Shows whether data checksums are turned on for this cluster. |  |
| data\_directory | Cluster | Restart | PPAS service account | Sets the server's data directory. |  |
| DateStyle | Session | Immediate | User | Sets the display format for date and time values. |  |
| db\_dialect | Session | Immediate | User | Sets the precedence of built-in namespaces. | X |
| dbms\_alert.max\_alerts | Cluster | Restart | PPAS service account | Sets maximum number of alerts. | X |
| dbms\_pipe.total\_message\_buffer | Cluster | Restart | PPAS service account | Specifies the total size of the buffer used for the DBMS\_PIPE package. | X |
| db\_user\_namespace | Cluster | Reload | PPAS service account | Enables per-database user names. |  |
| deadlock\_timeout | Session | Immediate | Superuser | Sets the time to wait on a lock before checking for deadlock. |  |
| debug\_assertions | Cluster | Preset | n/a | Turns on various assertion checks. (Not supported in PPAS builds.) |  |
| debug\_pretty\_print | Session | Immediate | User | Indents parse and plan tree displays. |  |
| debug\_print\_parse | Session | Immediate | User | Logs each query's parse tree. |  |
| debug\_print\_plan | Session | Immediate | User | Logs each query's execution plan. |  |
| debug\_print\_rewritten | Session | Immediate | User | Logs each query's rewritten parse tree. |  |
| default\_heap\_fillfactor | Session | Immediate | User | Create new tables with this heap fillfactor by default. | X |
| default\_statistics\_target | Session | Immediate | User | Sets the default statistics target. |  |
| default\_tablespace | Session | Immediate | User | Sets the default tablespace to create tables and indexes in. |  |
| default\_text\_search\_config | Session | Immediate | User | Sets default text search configuration. |  |
| default\_transaction\_deferrable | Session | Immediate | User | Sets the default deferrable status of new transactions. |  |
| default\_transaction\_isolation | Session | Immediate | User | Sets the transaction isolation level of each new transaction. |  |
| default\_transaction\_read\_only | Session | Immediate | User | Sets the default read-only status of new transactions. |  |
| default\_with\_oids | Session | Immediate | User | Create new tables with OIDs by default. |  |
| default\_with\_rowids | Session | Immediate | User | Create new tables with ROWID support (OIDs with indexes) by default. | X |
| dynamic\_library\_path | Session | Immediate | Superuser | Sets the path for dynamically loadable modules. |  |
| dynamic\_shared\_memory\_type | Cluster | Restart | PPAS service account | Selects the dynamic shared memory implementation used. |  |
| edb\_audit | Cluster | Reload | PPAS service account | Enable EDB Auditing to create audit reports in XML or CSV format. | X |
| edb\_audit\_connect | Cluster | Reload | PPAS service account | Audits each successful connection. | X |
| edb\_audit\_directory | Cluster | Reload | PPAS service account | Sets the destination directory for audit files. | X |
| edb\_audit\_disconnect | Cluster | Reload | PPAS service account | Audits end of a session. | X |
| edb\_audit\_filename | Cluster | Reload | PPAS service account | Sets the file name pattern for audit files. | X |
| edb\_audit\_rotation\_day | Cluster | Reload | PPAS service account | Automatic rotation of logfiles based on day of week. | X |
| edb\_audit\_rotation\_seconds | Cluster | Reload | PPAS service account | Automatic log file rotation will occur after N seconds. | X |
| edb\_audit\_rotation\_size | Cluster | Reload | PPAS service account | Automatic log file rotation will occur after N Megabytes. | X |
| edb\_audit\_statement | Cluster | Reload | PPAS service account | Sets the type of statements to audit. | X |
| edb\_audit\_tag | Session | Immediate | User | Specify a tag to be included in the audit log. | X |
| edb\_connectby\_order | Session | Immediate | User | Sort results of CONNECT BY queries with no ORDER BY to depth-first order. Note: For internal use only. | X |
| edb\_custom\_plan\_tries | Session | Immediate | User | Specifies the number of custom execution plans considered by the planner before the planner selects a generic execution plan. | X |
| edb\_dynatune | Cluster | Restart | PPAS service account | Sets the edb utilization percentage. | X |
| edb\_dynatune\_profile | Cluster | Restart | PPAS service account | Sets the workload profile for dynatune. | X |
| edb\_enable\_icache | Cluster | Restart | PPAS service account | Enable external shared buffer infinitecache mechanism. | X |
| edb\_enable\_pruning | Session | Immediate | User | Enables the planner to early-prune partitioned tables. | X |
| edb\_icache\_compression\_level | Session | Immediate | Superuser | Sets compression level of infinitecache buffers. | X |
| edb\_icache\_servers | Cluster | Reload | PPAS service account | A list of comma separated hostname:portnumber icache servers. | X |
| edb\_max\_resource\_groups | Cluster | Restart | PPAS service account | Specifies the maximum number of resource groups for simultaneous use. | X |
| edb\_max\_spins\_per\_delay | Cluster | Restart | PPAS service account | Specifies the number of times a session will spin while waiting for a lock. | X |
| edb\_redwood\_date | Session | Immediate | User | Determines whether DATE should behave like a TIMESTAMP or not. | X |
| edb\_redwood\_greatest\_least | Session | Immediate | User | Determines how GREATEST and LEAST functions should handle NULL parameters. | X |
| edb\_redwood\_raw\_names | Session | Immediate | User | Return the unmodified name stored in the PostgreSQL system catalogs from Redwood interfaces. | X |
| edb\_redwood\_strings | Session | Immediate | User | Treat NULL as an empty string when concatenated with a text value. | X |
| edb\_resource\_group | Session | Immediate | User | Specifies the resource group to be used by the current process. | X |
| edb\_sql\_protect.enabled | Cluster | Reload | PPAS service account | Defines whether SQL/Protect should track queries or not. | X |
| edb\_sql\_protect.level | Cluster | Reload | PPAS service account | Defines the behavior of SQL/Protect when an event is found. | X |
| edb\_sql\_protect.max\_protected\_relations | Cluster | Restart | PPAS service account | Sets the maximum number of relations protected by SQL/Protect per role. | X |
| edb\_sql\_protect.max\_protected\_roles | Cluster | Restart | PPAS service account | Sets the maximum number of roles protected by SQL/Protect. | X |
| edb\_sql\_protect.max\_queries\_to\_save | Cluster | Restart | PPAS service account | Sets the maximum number of offending queries to save by SQL/Protect. | X |
| edb\_stmt\_level\_tx | Session | Immediate | User | Allows continuing on errors instead of requiring a transaction abort. | X |
| edbldr.empty\_csv\_field | Session | Immediate | Superuser | Specifies how EDB\*Loader handles empty strings. | X |
| effective\_cache\_size | Session | Immediate | User | Sets the planner's assumption about the size of the disk cache. |  |
| effective\_io\_concurrency | Session | Immediate | User | Number of simultaneous requests that can be handled efficiently by the disk subsystem. |  |
| enable\_bitmapscan | Session | Immediate | User | Enables the planner's use of bitmap-scan plans. |  |
| enable\_hashagg | Session | Immediate | User | Enables the planner's use of hashed aggregation plans. |  |
| enable\_hashjoin | Session | Immediate | User | Enables the planner's use of hash join plans. |  |
| enable\_hints | Session | Immediate | User | Enable optimizer hints in SQL statements. | X |
| enable\_indexonlyscan | Session | Immediate | User | Enables the planner’s use of index-only-scan plans. |  |
| enable\_indexscan | Session | Immediate | User | Enables the planner's use of index-scan plans. |  |
| enable\_material | Session | Immediate | User | Enables the planner's use of materialization. |  |
| enable\_mergejoin | Session | Immediate | User | Enables the planner's use of merge join plans. |  |
| enable\_nestloop | Session | Immediate | User | Enables the planner's use of nested-loop join plans. |  |
| enable\_seqscan | Session | Immediate | User | Enables the planner's use of sequential-scan plans. |  |
| enable\_sort | Session | Immediate | User | Enables the planner's use of explicit sort steps. |  |
| enable\_tidscan | Session | Immediate | User | Enables the planner's use of TID scan plans. |  |
| escape\_string\_warning | Session | Immediate | User | Warn about backslash escapes in ordinary string literals. |  |
| event\_source | Cluster | Restart | PPAS service account | Sets the application name used to identify PostgreSQL messages in the event log. |  |
| exit\_on\_error | Session | Immediate | User | Terminate session on any error. |  |
| external\_pid\_file | Cluster | Restart | PPAS service account | Writes the postmaster PID to the specified file. |  |
| extra\_float\_digits | Session | Immediate | User | Sets the number of digits displayed for floating-point values. |  |
| from\_collapse\_limit | Session | Immediate | User | Sets the FROM-list size beyond which subqueries are not collapsed. |  |
| fsync | Cluster | Reload | PPAS service account | Forces synchronization of updates to disk. |  |
| full\_page\_writes | Cluster | Reload | PPAS service account | Writes full pages to WAL when first modified after a checkpoint. |  |
| geqo | Session | Immediate | User | Enables genetic query optimization. |  |
| geqo\_effort | Session | Immediate | User | GEQO: effort is used to set the default for other GEQO parameters. |  |
| geqo\_generations | Session | Immediate | User | GEQO: number of iterations of the algorithm. |  |
| geqo\_pool\_size | Session | Immediate | User | GEQO: number of individuals in the population. |  |
| geqo\_seed | Session | Immediate | User | GEQO: seed for random path selection. |  |
| geqo\_selection\_bias | Session | Immediate | User | GEQO: selective pressure within the population. |  |
| geqo\_threshold | Session | Immediate | User | Sets the threshold of FROM items beyond which GEQO is used. |  |
| gin\_fuzzy\_search\_limit | Session | Immediate | User | Sets the maximum allowed result for exact search by GIN. |  |
| hba\_file | Cluster | Restart | PPAS service account | Sets the server's "hba" configuration file. |  |
| hot\_standby | Cluster | Restart | PPAS service account | Allows connections and queries during recovery. |  |
| hot\_standby\_feedback | Cluster | Reload | PPAS service account | Allows feedback from a hot standby to the primary that will avoid query conflicts. |  |
| huge\_pages | Cluster | Restart | PPAS service account | Use of huge pages on Linux. |  |
| icu\_short\_form | Database | Preset | n/a | Shows the ICU collation order configuration. | X |
| ident\_file | Cluster | Restart | PPAS service account | Sets the server's "ident" configuration file. |  |
| ignore\_checksum\_failure | Session | Immediate | Superuser | Continues processing after a checksum failure. |  |
| ignore\_system\_indexes | Cluster/  Session | Reload/  Immediate | PPAS service account/  User | Disables reading from system indexes. (Can also be set with PGOPTIONS at session start.) |  |
| index\_advisor.enabled | Session | Immediate | User | Enable Index Advisor plugin. | X |
| integer\_datetimes | Cluster | Preset | n/a | Datetimes are integer based. |  |
| IntervalStyle | Session | Immediate | User | Sets the display format for interval values. |  |
| join\_collapse\_limit | Session | Immediate | User | Sets the FROM-list size beyond which JOIN constructs are not flattened. |  |
| krb\_caseins\_users | Cluster | Reload | PPAS service account | Sets whether Kerberos and GSSAPI user names should be treated as case-insensitive. |  |
| krb\_server\_keyfile | Cluster | Reload | PPAS service account | Sets the location of the Kerberos server key file. |  |
| lc\_collate | Database | Preset | n/a | Shows the collation order locale. |  |
| lc\_ctype | Database | Preset | n/a | Shows the character classification and case conversion locale. |  |
| lc\_messages | Session | Immediate | Superuser | Sets the language in which messages are displayed. |  |
| lc\_monetary | Session | Immediate | User | Sets the locale for formatting monetary amounts. |  |
| lc\_numeric | Session | Immediate | User | Sets the locale for formatting numbers. |  |
| lc\_time | Session | Immediate | User | Sets the locale for formatting date and time values. |  |
| listen\_addresses | Cluster | Restart | PPAS service account | Sets the host name or IP address(es) to listen to. |  |
| local\_preload\_libraries | Cluster/  Session | Reload/  Immediate | PPAS service account/  User | Lists shared libraries to preload into each backend. (Can also be set with PGOPTIONS at session start.) |  |
| lock\_timeout | Session | Immediate | User | Sets the maximum time allowed that a statement may wait for a lock. |  |
| lo\_compat\_privileges | Session | Immediate | Superuser | Enables backward compatibility mode for privilege checks on large objects. |  |
| log\_autovacuum\_min\_duration | Cluster | Reload | PPAS service account | Sets the minimum execution time above which autovacuum actions will be logged. |  |
| log\_checkpoints | Cluster | Reload | PPAS service account | Logs each checkpoint. |  |
| log\_connections | Cluster/  Session | Reload/  Immediate | PPAS service account/  User | Logs each successful connection. (Can also be set with PGOPTIONS at session start.) |  |
| log\_destination | Cluster | Reload | PPAS service account | Sets the destination for server log output. |  |
| log\_directory | Cluster | Reload | PPAS service account | Sets the destination directory for log files. |  |
| log\_disconnections | Cluster/  Session | Reload/  Immediate | PPAS service account/  User | Logs end of a session, including duration. (Can also be set with PGOPTIONS at session start.) |  |
| log\_duration | Session | Immediate | Superuser | Logs the duration of each completed SQL statement. |  |
| log\_error\_verbosity | Session | Immediate | Superuser | Sets the verbosity of logged messages. |  |
| log\_executor\_stats | Session | Immediate | Superuser | Writes executor performance statistics to the server log. |  |
| log\_file\_mode | Cluster | Reload | PPAS service account | Sets the file permissions for log files. |  |
| log\_filename | Cluster | Reload | PPAS service account | Sets the file name pattern for log files. |  |
| log\_hostname | Cluster | Reload | PPAS service account | Logs the host name in the connection logs. |  |
| log\_line\_prefix | Cluster | Reload | PPAS service account | Controls information prefixed to each log line. |  |
| log\_lock\_waits | Session | Immediate | Superuser | Logs long lock waits. |  |
| log\_min\_duration\_statement | Session | Immediate | Superuser | Sets the minimum execution time above which statements will be logged. |  |
| log\_min\_error\_statement | Session | Immediate | Superuser | Causes all statements generating error at or above this level to be logged. |  |
| log\_min\_messages | Session | Immediate | Superuser | Sets the message levels that are logged. |  |
| log\_parser\_stats | Session | Immediate | Superuser | Writes parser performance statistics to the server log. |  |
| log\_planner\_stats | Session | Immediate | Superuser | Writes planner performance statistics to the server log. |  |
| log\_rotation\_age | Cluster | Reload | PPAS service account | Automatic log file rotation will occur after N minutes. |  |
| log\_rotation\_size | Cluster | Reload | PPAS service account | Automatic log file rotation will occur after N kilobytes. |  |
| log\_statement | Session | Immediate | Superuser | Sets the type of statements logged. |  |
| log\_statement\_stats | Session | Immediate | Superuser | Writes cumulative performance statistics to the server log. |  |
| log\_temp\_files | Session | Immediate | Superuser | Log the use of temporary files larger than this number of kilobytes. |  |
| log\_timezone | Cluster | Reload | PPAS service account | Sets the time zone to use in log messages. |  |
| log\_truncate\_on\_rotation | Cluster | Reload | PPAS service account | Truncate existing log files of same name during log rotation. |  |
| logging\_collector | Cluster | Restart | PPAS service account | Start a subprocess to capture stderr output and/or csvlogs into log files. |  |
| maintenance\_work\_mem | Session | Immediate | User | Sets the maximum memory to be used for maintenance operations. |  |
| max\_connections | Cluster | Restart | PPAS service account | Sets the maximum number of concurrent connections. |  |
| max\_files\_per\_process | Cluster | Restart | PPAS service account | Sets the maximum number of simultaneously open files for each server process. |  |
| max\_function\_args | Cluster | Preset | n/a | Shows the maximum number of function arguments. |  |
| max\_identifier\_length | Cluster | Preset | n/a | Shows the maximum identifier length. |  |
| max\_index\_keys | Cluster | Preset | n/a | Shows the maximum number of index keys. |  |
| max\_locks\_per\_transaction | Cluster | Restart | PPAS service account | Sets the maximum number of locks per transaction. |  |
| max\_pred\_locks\_per\_transaction | Cluster | Restart | PPAS service account | Sets the maximum number of predicate locks per transaction. |  |
| max\_prepared\_transactions | Cluster | Restart | PPAS service account | Sets the maximum number of simultaneously prepared transactions. |  |
| max\_replication\_slots | Cluster | Restart | PPAS service account | Sets the maximum number of simultaneously defined replication slots. |  |
| max\_stack\_depth | Session | Immediate | Superuser | Sets the maximum stack depth, in kilobytes. |  |
| max\_standby\_archive\_delay | Cluster | Reload | PPAS service account | Sets the maximum delay before canceling queries when a hot standby server is processing archived WAL data. |  |
| max\_standby\_streaming\_delay | Cluster | Reload | PPAS service account | Sets the maximum delay before canceling queries when a hot standby server is processing streamed WAL data. |  |
| max\_wal\_senders | Cluster | Restart | PPAS service account | Sets the maximum number of simultaneously running WAL sender processes. |  |
| max\_wal\_size | Cluster | Reload | PPAS service account | Sets the maximum size to which the WAL will grow between automatic WAL checkpoints. The default is 1GB. |  |
| max\_worker\_processes | Cluster | Restart | PPAS service account | Maximum number of concurrent worker processes. |  |
| min\_wal\_size | Cluster | Reload | PPAS service account | Sets thethreshold at which WAL logs will be recycled rather than removed. The default is 80 MB. |  |
| nls\_length\_semantics | Session | Immediate | Superuser | Sets the semantics to use for char, varchar, varchar2 and long columns. | X |
| odbc\_lib\_path | Cluster | Restart | PPAS service account | Sets the path for ODBC library. | X |
| optimizer\_mode | Session | Immediate | User | Default optimizer mode. | X |
| oracle\_home | Cluster | Restart | PPAS service account | Sets the path for the Oracle home directory. | X |
| password\_encryption | Session | Immediate | User | Encrypt passwords. |  |
| port | Cluster | Restart | PPAS service account | Sets the TCP port on which the server listens. |  |
| post\_auth\_delay | Cluster/  Session | Reload/  Immediate | PPAS service account/  User | Waits N seconds on connection startup after authentication. (Can also be set with PGOPTIONS at session start.) |  |
| pre\_auth\_delay | Cluster | Reload | PPAS service account | Waits N seconds on connection startup before authentication. |  |
| qreplace\_function | Session | Immediate | Superuser | The function to be used by Query Replace feature. Note: For internal use only. | X |
| query\_rewrite\_enabled | Session | Immediate | User | Child table scans will be skipped if their constraints guarantee that no rows match the query. | X |
| query\_rewrite\_integrity | Session | Immediate | Superuser | Sets the degree to which query rewriting must be enforced. | X |
| quote\_all\_identifiers | Session | Immediate | User | When generating SQL fragments, quote all identifiers. |  |
| random\_page\_cost | Session | Immediate | User | Sets the planner's estimate of the cost of a nonsequentially fetched disk page. |  |
| restart\_after\_crash | Cluster | Reload | PPAS service account | Reinitialize server after backend crash. |  |
| search\_path | Session | Immediate | User | Sets the schema search order for names that are not schema-qualified. |  |
| segment\_size | Cluster | Preset | n/a | Shows the number of pages per disk file. |  |
| seq\_page\_cost | Session | Immediate | User | Sets the planner's estimate of the cost of a sequentially fetched disk page. |  |
| server\_encoding | Database | Preset | n/a | Sets the server (database) character set encoding. |  |
| server\_version | Cluster | Preset | n/a | Shows the server version. |  |
| server\_version\_num | Cluster | Preset | n/a | Shows the server version as an integer. |  |
| session\_preload\_libraries | Session | Immediate but only at connection start | Superuser | Lists shared libraries to preload into each backend. |  |
| session\_replication\_role | Session | Immediate | Superuser | Sets the session's behavior for triggers and rewrite rules. |  |
| shared\_buffers | Cluster | Restart | PPAS service account | Sets the number of shared memory buffers used by the server. |  |
| shared\_preload\_libraries | Cluster | Restart | PPAS service account | Lists shared libraries to preload into server. |  |
| sql\_inheritance | Session | Immediate | User | Causes subtables to be included by default in various commands. |  |
| ssl | Cluster | Restart | PPAS service account | Enables SSL connections. |  |
| ssl\_ca\_file | Cluster | Restart | PPAS service account | Location of the SSL certificate authority file. |  |
| ssl\_cert\_file | Cluster | Restart | PPAS service account | Location of the SSL server certificate file. |  |
| ssl\_ciphers | Cluster | Restart | PPAS service account | Sets the list of allowed SSL ciphers. |  |
| ssl\_crl\_file | Cluster | Restart | PPAS service account | Location of the SSL certificate revocation list file. |  |
| ssl\_ecdh\_curve | Cluster | Restart | PPAS service account | Sets the curve to use for ECDH. |  |
| ssl\_key\_file | Cluster | Restart | PPAS service account | Location of the SSL server private key file. |  |
| ssl\_prefer\_server\_ciphers | Cluster | Restart | PPAS service account | Give priority to server ciphersuite order. |  |
| ssl\_renegotiation\_limit | Session | Immediate | User | Set the amount of traffic to send and receive before renegotiating the encryption keys. |  |
| standard\_conforming\_strings | Session | Immediate | User | Causes '...' strings to treat backslashes literally. |  |
| statement\_timeout | Session | Immediate | User | Sets the maximum allowed duration of any statement. |  |
| stats\_temp\_directory | Cluster | Reload | PPAS service account | Writes temporary statistics files to the specified directory. |  |
| superuser\_reserved\_connections | Cluster | Restart | PPAS service account | Sets the number of connection slots reserved for superusers. |  |
| synchronize\_seqscans | Session | Immediate | User | Enable synchronized sequential scans. |  |
| synchronous\_commit | Session | Immediate | User | Sets immediate fsync at commit. |  |
| synchronous\_standby\_names | Cluster | Reload | PPAS service account | List of names of potential synchronous standbys. |  |
| syslog\_facility | Cluster | Reload | PPAS service account | Sets the syslog "facility" to be used when syslog enabled. |  |
| syslog\_ident | Cluster | Reload | PPAS service account | Sets the program name used to identify PostgreSQL messages in syslog. |  |
| tcp\_keepalives\_count | Session | Immediate | User | Maximum number of TCP keepalive retransmits. |  |
| tcp\_keepalives\_idle | Session | Immediate | User | Time between issuing TCP keepalives. |  |
| tcp\_keepalives\_interval | Session | Immediate | User | Time between TCP keepalive retransmits. |  |
| temp\_buffers | Session | Immediate | User | Sets the maximum number of temporary buffers used by each session. |  |
| temp\_file\_limit | Session | Immediate | Superuser | Limits the total size of all temporary files used by each session. |  |
| temp\_tablespaces | Session | Immediate | User | Sets the tablespace(s) to use for temporary tables and sort files. |  |
| timed\_statistics | Session | Immediate | User | Enables the recording of timed statistics. | X |
| timezone | Session | Immediate | User | Sets the time zone for displaying and interpreting time stamps. |  |
| timezone\_abbreviations | Session | Immediate | User | Selects a file of time zone abbreviations. |  |
| trace\_hints | Session | Immediate | User | Emit debug info about hints being honored. | X |
| trace\_notify | Session | Immediate | User | Generates debugging output for LISTEN and NOTIFY. |  |
| trace\_recovery\_messages | Cluster | Reload | PPAS service account | Enables logging of recovery-related debugging information. |  |
| trace\_sort | Session | Immediate | User | Emit information about resource usage in sorting. |  |
| track\_activities | Session | Immediate | Superuser | Collects information about executing commands. |  |
| track\_activity\_query\_size | Cluster | Restart | PPAS service account | Sets the size reserved for pg\_stat\_activity.current\_query, in bytes. |  |
| track\_counts | Session | Immediate | Superuser | Collects statistics on database activity. |  |
| track\_functions | Session | Immediate | Superuser | Collects function-level statistics on database activity. |  |
| track\_io\_timing | Session | Immediate | Superuser | Collects timing statistics for database I/O activity. |  |
| transaction\_deferrable | Session | Immediate | User | Whether to defer a read-only serializable transaction until it can be executed with no possible serialization failures. |  |
| transaction\_isolation | Session | Immediate | User | Sets the current transaction's isolation level. |  |
| transaction\_read\_only | Session | Immediate | User | Sets the current transaction's read-only status. |  |
| transform\_null\_equals | Session | Immediate | User | Treats "expr=NULL" as "expr IS NULL". |  |
| unix\_socket\_directories | Cluster | Restart | PPAS service account | Sets the directory where the Unix-domain socket will be created. |  |
| unix\_socket\_group | Cluster | Restart | PPAS service account | Sets the owning group of the Unix-domain socket. |  |
| unix\_socket\_permissions | Cluster | Restart | PPAS service account | Sets the access permissions of the Unix-domain socket. |  |
| update\_process\_title | Session | Immediate | Superuser | Updates the process title to show the active SQL command. |  |
| utl\_encode.uudecode\_redwood | Session | Immediate | User | Allows decoding of Oracle-created uuencoded data. | X |
| utl\_file.umask | Session | Immediate | User | Umask used for files created through the UTL\_FILE package. | X |
| vacuum\_cost\_delay | Session | Immediate | User | Vacuum cost delay in milliseconds. |  |
| vacuum\_cost\_limit | Session | Immediate | User | Vacuum cost amount available before napping. |  |
| vacuum\_cost\_page\_dirty | Session | Immediate | User | Vacuum cost for a page dirtied by vacuum. |  |
| vacuum\_cost\_page\_hit | Session | Immediate | User | Vacuum cost for a page found in the buffer cache. |  |
| vacuum\_cost\_page\_miss | Session | Immediate | User | Vacuum cost for a page not found in the buffer cache. |  |
| vacuum\_defer\_cleanup\_age | Cluster | Reload | PPAS service account | Number of transactions by which VACUUM and HOT cleanup should be deferred, if any. |  |
| vacuum\_freeze\_min\_age | Session | Immediate | User | Minimum age at which VACUUM should freeze a table row. |  |
| vacuum\_freeze\_table\_age | Session | Immediate | User | Age at which VACUUM should scan whole table to freeze tuples. |  |
| vacuum\_multixact\_freeze\_min\_age | Session | Immediate | User | Minimum age at which VACUUM should freeze a MultiXactId in a table row. |  |
| vacuum\_multixact\_freeze\_table\_age | Session | Immediate | User | Multixact age at which VACUUM should scan whole table to freeze tuples. |  |
| wal\_block\_size | Cluster | Preset | n/a | Shows the block size in the write ahead log. |  |
| wal\_buffers | Cluster | Restart | PPAS service account | Sets the number of disk-page buffers in shared memory for WAL. |  |
| wal\_keep\_segments | Cluster | Reload | PPAS service account | Sets the number of WAL files held for standby servers. |  |
| wal\_level | Cluster | Restart | PPAS service account | Set the level of information written to the WAL. |  |
| wal\_log\_hints | Cluster | Restart | PPAS service account | Writes full pages to WAL when first modified after a checkpoint, even for non-critical modifications. |  |
| wal\_receiver\_status\_interval | Cluster | Reload | PPAS service account | Sets the maximum interval between WAL receiver status reports to the primary. |  |
| wal\_receiver\_timeout | Cluster | Reload | PPAS service account | Sets the maximum wait time to receive data from the primary. |  |
| wal\_segment\_size | Cluster | Preset | n/a | Shows the number of pages per write ahead log segment. |  |
| wal\_sender\_timeout | Cluster | Reload | PPAS service account | Sets the maximum time to wait for WAL replication. |  |
| wal\_sync\_method | Cluster | Reload | PPAS service account | Selects the method used for forcing WAL updates to disk. |  |
| wal\_writer\_delay | Cluster | Reload | PPAS service account | WAL writer sleep time between WAL flushes. |  |
| work\_mem | Session | Immediate | User | Sets the maximum memory to be used for query workspaces. |  |
| xloginsert\_locks | Cluster | Restart | PPAS service account | Sets the number of locks used for concurrent xlog insertions. |  |
| xmlbinary | Session | Immediate | User | Sets how binary values are to be encoded in XML. |  |
| Xmloption | Session | Immediate | User | Sets whether XML data in implicit parsing and serialization operations is to be considered as documents or content fragments. |  |
| zero\_damaged\_pages | Session | Immediate | Superuser | Continues processing past damaged page headers. |  |

### Configuration Parameters by Functionality

This section provides more detail for certain groups of configuration parameters.

The section heading for each parameter is followed by a list of attributes:

* Parameter Type. Type of values the parameter can accept. See Section 2.1.1 for a discussion of parameter type values.
* Default Value. Default setting if a value is not explicitly set in the configuration file.
* Range. Permitted range of values.
* Minimum Scope of Effect. Smallest scope for which a distinct setting can be made. Generally, the minimal scope of a distinct setting is either the entire cluster (the setting is the same for all databases and sessions thereof, in the cluster), or per session (the setting may vary by role, database, or individual session). (This attribute has the same meaning as the “Scope of Effect” column in the table of Section 2.1.2.)
* When Value Changes Take Effect. Least invasive action required to activate a change to a parameter’s value. All parameter setting changes made in the configuration file can be put into effect with a restart of the database server; however certain parameters require a database server restart. Some parameter setting changes can be put into effect with a reload of the configuration file without stopping the database server. Finally, other parameter setting changes can be put into effect with some client side action whose result is immediate. (This attribute has the same meaning as the “When Takes Effect” column in the table of Section 2.1.2.)
* Required Authorization to Activate. The type of user authorization to activate a change to a parameter’s setting. If a database server restart or a configuration file reload is required, then the user must be a PPAS service account (enterprisedb or postgres depending upon the Advanced Server compatibility installation mode). This attribute has the same meaning as the “Authorized User” column in the table of Section 2.1.2.

#### Top Performance Related Parameters

This section discusses the configuration parameters that have the most immediate impact on performance.

##### shared\_buffers

Parameter Type: Integer

Default Value: 32MB

Range: 128kB to system dependent

Minimum Scope of Effect: Cluster

When Value Changes Take Effect: Restart

Required Authorization to Activate: PPAS service account

Sets the amount of memory the database server uses for shared memory buffers. The default is typically 32 megabytes (32MB), but might be less if your kernel settings will not support it (as determined during initdb). This setting must be at least 128 kilobytes. (Non-default values of BLCKSZ change the minimum.) However, settings significantly higher than the minimum are usually needed for good performance.

If you have a dedicated database server with 1GB or more of RAM, a reasonable starting value for shared\_buffers is 25% of the memory in your system. There are some workloads where even large settings for shared\_buffers are effective, but because Advanced Server also relies on the operating system cache, it is unlikely that an allocation of more than 40% of RAM to shared\_buffers will work better than a smaller amount.

On systems with less than 1GB of RAM, a smaller percentage of RAM is appropriate, so as to leave adequate space for the operating system (15% of memory is more typical in these situations). Also, on Windows, large values for shared\_buffers aren't as effective. You may find better results keeping the setting relatively low and using the operating system cache more instead. The useful range for shared\_buffers on Windows systems is generally from 64MB to 512MB.

Increasing this parameter might cause Advanced Server to request more System V shared memory than your operating system's default configuration allows. See Section 17.4.1, “Shared Memory and Semaphores” in the PostgreSQL Core Documentation for information on how to adjust those parameters, if necessary.

##### work\_mem

Parameter Type: Integer

Default Value: 1MB

Range: 64kB to 2097151kB

Minimum Scope of Effect: Per session

When Value Changes Take Effect: Immediate

Required Authorization to Activate: Session user

Specifies the amount of memory to be used by internal sort operations and hash tables before writing to temporary disk files. The value defaults to one megabyte (1MB). Note that for a complex query, several sort or hash operations might be running in parallel; each operation will be allowed to use as much memory as this value specifies before it starts to write data into temporary files. Also, several running sessions could be doing such operations concurrently. Therefore, the total memory used could be many times the value of work\_mem; 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.

Reasonable values are typically between 4MB and 64MB, depending on the size of your machine, how many concurrent connections you expect (determined by max\_connections), and the complexity of your queries.

##### maintenance\_work\_mem

Parameter Type: Integer

Default Value: 16MB

Range: 1024kB to 2097151kB

Minimum Scope of Effect: Per session

When Value Changes Take Effect: Immediate

Required Authorization to Activate: Session user

Specifies the maximum amount of memory to be used by maintenance operations, such as VACUUM, CREATE INDEX, and ALTER TABLE ADD FOREIGN KEY. It defaults to 16 megabytes (16MB). Since only one of these operations can be executed at a time by a database session, and an installation normally doesn't have many of them running concurrently, it's safe to set this value significantly larger than work\_mem. Larger settings might improve performance for vacuuming and for restoring database dumps.

Note that when autovacuum runs, up to autovacuum\_max\_workers times this memory may be allocated, so be careful not to set the default value too high.

A good rule of thumb is to set this to about 5% of system memory, but not more than about 512MB. Larger values won't necessarily improve performance.

##### wal\_buffers

Parameter Type: Integer

Default Value: 64kB

Range: 32kB to system dependent

Minimum Scope of Effect: Cluster

When Value Changes Take Effect: Restart

Required Authorization to Activate: PPAS service account

The amount of memory used in shared memory for WAL data. The default is 64 kilobytes (64kB). 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.

Increasing this parameter might cause Advanced Server to request more System V shared memory than your operating system's default configuration allows. See Section 17.4.1, “Shared Memory and Semaphores” in the PostgreSQL Core Documentation for information on how to adjust those parameters, if necessary.

Although even this very small setting does not always cause a problem, there are situations where it can result in extra fsync calls, and degrade overall system throughput. Increasing this value to 1MB or so can alleviate this problem. On very busy systems, an even higher value may be needed, up to a maximum of about 16MB. Like shared\_buffers, this parameter increases Advanced Server’s initial shared memory allocation, so if increasing it causes an Advanced Server start failure, you will need to increase the operating system limit.

##### checkpoint\_segments

Now deprecated; this parameter is not supported by server versions 9.5 or later.

##### checkpoint\_completion\_target

Parameter Type: Floating point

Default Value: 0.5

Range: 0 to 1

Minimum Scope of Effect: Cluster

When Value Changes Take Effect: Reload

Required Authorization to Activate: PPAS service account

Specifies the target of checkpoint completion as a fraction of total time between checkpoints. This spreads out the checkpoint writes while the system starts working towards the next checkpoint.

The default of 0.5 means aim to finish the checkpoint writes when 50% of the next checkpoint is ready. A value of 0.9 means aim to finish the checkpoint writes when 90% of the next checkpoint is done, thus throttling the checkpoint writes over a larger amount of time and avoiding spikes of performance bottlenecking.

##### checkpoint\_timeout

Parameter Type: Integer

Default Value: 5min

Range: 30s to 3600s

Minimum Scope of Effect: Cluster

When Value Changes Take Effect: Reload

Required Authorization to Activate: PPAS service account

Maximum time between automatic WAL checkpoints, in seconds. The default is five minutes (5min). Increasing this parameter can increase the amount of time needed for crash recovery.

Increasing checkpoint\_timeout to a larger value, such as 15 minutes, can reduce the I/O load on your system, especially when using large values for shared\_buffers.

The downside of making the aforementioned adjustments to the checkpoint parameters is that your system will use a modest amount of additional disk space, and will take longer to recover in the event of a crash. However, for most users, this is a small price to pay for a significant performance improvement.

##### max\_wal\_size

Parameter Type: Integer

Default Value: 1 GB

Range: 2 to 2147483647

Minimum Scope of Effect: Cluster

When Value Changes Take Effect: Reload

Required Authorization to Activate: PPAS service account

max\_wal\_size specifies the maximum size that the WAL will reach between automatic WAL checkpoints. This is a soft limit; WAL size can exceed max\_wal\_size under special circumstances (when under a heavy load, a failing archive\_command, or a high wal\_keep\_segments setting).

Increasing this parameter can increase the amount of time needed for crash recovery. This parameter can only be set in the postgresql.conf file or on the server command line.

##### min\_wal\_size

Parameter Type: Integer

Default Value: 80 MB

Range: 2 to 2147483647

Minimum Scope of Effect: Cluster

When Value Changes Take Effect: Reload

Required Authorization to Activate: PPAS service account

If WAL disk usage stays below the value specified by min\_wal\_size, old WAL files are recycled for future use at a checkpoint, rather than removed. This ensures that enough WAL space is reserved to handle spikes in WAL usage (like when running large batch jobs). This parameter can only be set in the postgresql.conf file or on the server command line.

##### bgwriter\_delay

Parameter Type: Integer

Default Value: 200ms

Range: 10ms to 10000ms

Minimum Scope of Effect: Cluster

When Value Changes Take Effect: Reload

Required Authorization to Activate: PPAS service account

Specifies the delay between activity rounds for the background writer. In each round the writer issues writes for some number of dirty buffers (controllable by the bgwriter\_lru\_maxpages and bgwriter\_lru\_multiplier parameters). It then sleeps for bgwriter\_delay milliseconds, and repeats.

The default value is 200 milliseconds (200ms). Note that on many systems, the effective resolution of sleep delays is 10 milliseconds; setting bgwriter\_delay to a value that is not a multiple of 10 might have the same results as setting it to the next higher multiple of 10.

Typically, when tuning bgwriter\_delay, it should be reduced from its default value. This parameter is rarely increased, except perhaps to save on power consumption on a system with very low utilization.

##### seq\_page\_cost

Parameter Type: Floating point

Default Value: 1

Range: 0 to 1.79769e+308

Minimum Scope of Effect: Per session

When Value Changes Take Effect: Immediate

Required Authorization to Activate: Session user

Sets the planner's estimate of the cost of a disk page fetch that is part of a series of sequential fetches. The default is 1.0. This value can be overridden for a particular tablespace by setting the tablespace parameter of the same name. (Refer to the ALTER TABLESPACE command in the PostgreSQL Core Documentation.)

The default value assumes very little caching, so it's frequently a good idea to reduce it. Even if your database is significantly larger than physical memory, you might want to try setting this parameter to less than 1 (rather than its default value of 1) to see whether you get better query plans that way. If your database fits entirely within memory, you can lower this value much more, perhaps to 0.1.

##### random\_page\_cost

Parameter Type: Floating point

Default Value: 4

Range: 0 to 1.79769e+308

Minimum Scope of Effect: Per session

When Value Changes Take Effect: Immediate

Required Authorization to Activate: Session user

Sets the planner's estimate of the cost of a non-sequentially-fetched disk page. The default is 4.0. This value can be overridden for a particular tablespace by setting the tablespace parameter of the same name. (Refer to the ALTER TABLESPACE command in the PostgreSQL Core Documentation.)

Reducing this value relative to seq\_page\_cost will cause the system to prefer index scans; raising it will make index scans look relatively more expensive. You can raise or lower both values together to change the importance of disk I/O costs relative to CPU costs, which are described by the cpu\_tuple\_cost and cpu\_index\_tuple\_cost parameters.

The default value assumes very little caching, so it's frequently a good idea to reduce it. Even if your database is significantly larger than physical memory, you might want to try setting this parameter to 2 (rather than its default of 4) to see whether you get better query plans that way. If your database fits entirely within memory, you can lower this value much more, perhaps to 0.1.

Although the system will let you do so, never set random\_page\_cost less than seq\_page\_cost. However, setting them equal (or very close to equal) makes sense if the database fits mostly or entirely within memory, since in that case there is no penalty for touching pages out of sequence. Also, in a heavily-cached database you should lower both values relative to the CPU parameters, since the cost of fetching a page already in RAM is much smaller than it would normally be.

##### effective\_cache\_size

Parameter Type: Integer

Default Value: 128MB

Range: 8kB to 17179869176kB

Minimum Scope of Effect: Per session

When Value Changes Take Effect: Immediate

Required Authorization to Activate: Session user

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. When setting this parameter you should consider both Advanced Server’s shared buffers and the portion of the kernel's disk cache that will be used for Advanced Server data files. Also, take into account the expected number of concurrent queries on different tables, since they will have to share the available space. This parameter has no effect on the size of shared memory allocated by Advanced Server, nor does it reserve kernel disk cache; it is used only for estimation purposes. The default is 128 megabytes (128MB).

If this parameter is set too low, the planner may decide not to use an index even when it would be beneficial to do so. Setting effective\_cache\_size to 50% of physical memory is a normal, conservative setting. A more aggressive setting would be approximately 75% of physical memory.

##### synchronous\_commit

Parameter Type: Boolean

Default Value: true

Range: {true | false}

Minimum Scope of Effect: Per session

When Value Changes Take Effect: Immediate

Required Authorization to Activate: Session user

Specifies whether transaction commit will wait for WAL records to be written to disk before the command returns a "success" indication to the client. The default, and safe, setting is on. When off, there can be a delay between when success is reported to the client and when the transaction is really guaranteed to be safe against a server crash. (The maximum delay is three times wal\_writer\_delay.)

Unlike fsync, setting this parameter to off does not create any risk of database inconsistency: an operating system or database crash might result in some recent allegedly-committed transactions being lost, but the database state will be just the same as if those transactions had been aborted cleanly.

So, turning synchronous\_commit off can be a useful alternative when performance is more important than exact certainty about the durability of a transaction. See Section 29.3, *Asynchronous* *Commit* in the PostgreSQL Core Documentation for information.

This parameter can be changed at any time; the behavior for any one transaction is determined by the setting in effect when it commits. It is therefore possible, and useful, to have some transactions commit synchronously and others asynchronously. For example, to make a single multistatement transaction commit asynchronously when the default is the opposite, issue SET LOCAL synchronous\_commit TO OFF within the transaction.

##### edb\_max\_spins\_per\_delay

Parameter Type: Integer

Default Value: 1000

Range: {10 | 1000}

Minimum Scope of Effect: Per cluster

When Value Changes Take Effect: Restart

Required Authorization to Activate: PPAS service account

Use edb\_max\_spins\_per\_delay to specify the maximum number of times that a session will 'spin' while waiting for a spin-lock. If a lock is not acquired, the session will sleep. If you do not specify an alternative value for edb\_max\_spins\_per\_delay, the server will enforce the default value of 1000.

This may be useful for sytems that use NUMA (non-uniform memory access) architecture.

#### Resource Usage / Memory

The configuration parameters in this section control resource usage pertaining to memory.

##### edb\_dynatune

Parameter Type: Integer

Default Value: 0

Range: 0 to 100

Minimum Scope of Effect: Cluster

When Value Changes Take Effect: Restart

Required Authorization to Activate: PPAS service account

Determines how much of the host system’s resources are to be used by the database server based upon the host machine’s total available resources and the intended usage of the host machine.

When Advanced Server is initially installed, the edb\_dynatune parameter is set in accordance with the selected usage of the host machine on which it was installed (i.e., development machine, mixed use machine, or dedicated server). For most purposes, there is no need for the database administrator to adjust the various configuration parameters in the postgresql.conf file in order to improve performance.

The edb\_dynatune parameter can be set to any integer value between 0 and 100, inclusive. A value of 0, turns off the dynamic tuning feature thereby leaving the database server resource usage totally under the control of the other configuration parameters in the postgresql.conf file.

A low non-zero, value (e.g., 1 - 33) dedicates the least amount of the host machine’s resources to the database server. This setting would be used for a development machine where many other applications are being used.

A value in the range of 34 - 66 dedicates a moderate amount of resources to the database server. This setting might be used for a dedicated application server that may have a fixed number of other applications running on the same machine as Advanced Server.

The highest values (e.g., 67 - 100) dedicate most of the server’s resources to the database server. This setting would be used for a host machine that is totally dedicated to running Advanced Server.

Once a value of edb\_dynatune is selected, database server performance can be further fine-tuned by adjusting the other configuration parameters in the postgresql.conf file. Any adjusted setting overrides the corresponding value chosen by edb\_dynatune. You can change the value of a parameter by un-commenting the configuration parameter, specifying the desired value, and restarting the database server.

##### edb\_dynatune\_profile

Parameter Type: Enum

Default Value: oltp

Range: {oltp | reporting | mixed}

Minimum Scope of Effect: Cluster

When Value Changes Take Effect: Restart

Required Authorization to Activate: PPAS service account

This parameter is used to control tuning aspects based upon the expected workload profile on the database server.

The following are the possible values:

* oltp. Recommended when the database server is processing heavy online transaction processing workloads.
* reporting. Recommended for database servers used for heavy data reporting.
* mixed. Recommended for servers that provide a mix of transaction processing and data reporting.

##### edb\_enable\_icache

Parameter Type: Boolean

Default Value: false

Range: {true | false}

Minimum Scope of Effect: Cluster

When Value Changes Take Effect: Restart

Required Authorization to Activate: PPAS service account

Enables or disables Infinite Cache. If edb\_enable\_icache is set to on, Infinite Cache is enabled; if the parameter is set to off, Infinite Cache is disabled.

If you set edb\_enable\_icache to on, you must also specify a list of cache servers by setting the edb\_icache\_servers parameter.

##### edb\_icache\_servers

Parameter Type: String

Default Value: none

Range: n/a

Minimum Scope of Effect: Cluster

When Value Changes Take Effect: Reload

Required Authorization to Activate: PPAS service account

The edb\_icache\_servers parameter specifies a list of one or more servers with active edb-icache daemons. edb\_icache\_servers is a string value that takes the form of a comma-separated list of hostname:port pairs. You can specify each pair in any of the following forms:

* hostname
* IP\_address
* hostname:portnumber
* IP\_address:portnumber

If you do not specify a port number, Infinite Cache assumes that the cache server is listening at port 11211. This configuration parameter will take effect only if edb\_enable\_icache is set to on. Use the edb\_icache\_servers parameter to specify a maximum of 128 cache nodes.

You can dynamically modify the Infinite Cache server nodes. To change the Infinite Cache server configuration, use the edb\_icache\_servers parameter in the postgresql.conf file to perform the following:

* Specify additional cache information to add server(s).
* Delete server information to remove server(s).
* Specify additional server information and delete existing server information to both add and delete servers during the same reload operation.

After updating the edb\_icache\_servers parameter in the postgresql.conf file, you must reload the configuration parameters for the changes to take effect.

##### edb\_icache\_compression\_level

Parameter Type: Integer

Default Value: 6

Range: 0 to 9

Minimum Scope of Effect: Per session

When Value Changes Take Effect: Immediate

Required Authorization to Activate: Superuser

The edb\_icache\_compression\_level parameter controls the compression level that is applied to each page before storing it in the distributed Infinite Cache.

When Advanced Server reads data from disk, it typically reads the data in 8kB increments. If edb\_icache\_compression\_level is set to 0, each time Advanced Server sends an 8kB page to the Infinite Cache server that page is stored (uncompressed) in 8kB of cache memory. If the edb\_icache\_compression\_level parameter is set to 9, Advanced Server applies the maximum compression possible before sending it to the Infinite Cache server, so a page that previously took 8kB of cached memory might take 2kB of cached memory. Exact compression numbers are difficult to predict, as they are dependent on the nature of the data on each page.

This parameter must be an integer in the range 0 to 9.

* A compression level of 0 disables compression; it uses no CPU time for compression, but requires more storage space and network resources to process.
* A compression level of 9 invokes the maximum amount of compression; it increases the load on the CPU, but less data flows across the network, so network demand is reduced. Each page takes less room on the Infinite Cache server, so memory requirements are reduced.
* A compression level of 5 or 6 is a reasonable compromise between the amount of compression received and the amount of CPU time invested.

The compression level must be set by the superuser and can be changed for the current session while the server is running. The following command disables the compression mechanism for the currently active session:

SET edb\_icache\_compression\_level TO 0;

#### Resource Usage / EDB Resource Manager

The configuration parameters in this section control resource usage through EDB Resource Manager.

##### edb\_max\_resource\_groups

Parameter Type: Integer

Default Value: 16

Range: 0 to 65536

Minimum Scope of Effect: Cluster

When Value Changes Take Effect: Restart

Required Authorization to Activate: PPAS service account

This parameter controls the maximum number of resource groups that can be used simultaneously by EDB Resource Manager. More resource groups can be created than the value specified by edb\_max\_resource\_groups, however, the number of resource groups in active use by processes in these groups cannot exceed this value.

Parameter edb\_max\_resource\_groups should be set comfortably larger than the number of groups you expect to maintain so as not to run out.

##### edb\_resource\_group

Parameter Type: String

Default Value: none

Range: n/a

Minimum Scope of Effect: Per session

When Value Changes Take Effect: Immediate

Required Authorization to Activate: Session user

Set the edb\_resource\_group parameter to the name of the resource group to which the current session is to be controlled by EDB Resource Manager according to the group’s resource type settings.

If the parameter is not set, then the current session does not utilize EDB Resource Manager.

#### Query Tuning

This section describes the configuration parameters used for optimizer hints.

##### enable\_hints

Parameter Type: Boolean

Default Value: true

Range: {true | false}

Minimum Scope of Effect: Per session

When Value Changes Take Effect: Immediate

Required Authorization to Activate: Session user

Optimizer hints embedded in SQL commands are utilized when enable\_hints is on. Optimizer hints are ignored when this parameter is off.

#### Query Tuning / Planner Method Configuration

This section describes the configuration parameters used for planner method configuration.

##### edb\_custom\_plan\_tries

Parameter Type: Numeric

Default Value: 5

Range: {0 | 100}

Minimum Scope of Effect: Per session

When Value Changes Take Effect: Immediate

Required Authorization to Activate: Session User

This configuration parameter controls the number of custom execution plans considered by the planner before the planner settles on a generic execution plan.

When a client application repeatedly executes a prepared statement, the server may decide to evaluate several execution plans before deciding to choose a *custom* plan or a *generic* plan.

* A custom plan is a plan built for a specific set of parameter values.
* A generic plan is a plan that will work with any set of parameter values supplied by the client application.

By default, the optimizer will generate five custom plans before evaluating a generic plan. That means that if you execute a prepared statement six times, the optimizer will generate five custom plans, then one generic plan, and then decide whether to stick with the generic plan.

In certain workloads, this extra planning can have a negative impact on performance. You can adjust the edb\_custom\_plan\_tries configuration parameter to decrease the number of custom plans considered before evaluating a generic plan. Setting edb\_custom\_plan\_tries to 0 will effectively disable custom plan generation.

Consider the following query:

PREPARE custQuery AS SELECT \* FROM customer WHERE salesman >= $1

The $1 token in this query is a parameter marker - the client application must provide a value for each parameter marker each time the statement executes.

If an index has been defined on customer.salesman, the optimizer may choose to execute this query using a sequential scan, or using an index scan. In some cases, an index is faster than a sequential scan; in other cases, the sequential scan will win. The optimal plan will depend on the distribution of salesman values in the table and on the search value (the value provided for the $1 parameter).

When the client application repeatedly executes the custQuery prepared statement, the optimizer will generate some number of parameter-value-specific execution plans (custom plans), followed by a generic plan (a plan that ignores the parameter values), and then decide whether to stick with the generic plan or to continue to generate custom plans for each execution. The decision process takes into account not only the cost of executing the plans, but the cost of generating custom plans as well.

##### edb\_enable\_pruning

Parameter Type: Boolean

Default Value: true

Range: {true | false}

Minimum Scope of Effect: Per session

When Value Changes Take Effect: Immediate

Required Authorization to Activate: Session user

When set to TRUE, edb\_enable\_pruning allows the query planner to early-prune partitioned tables. Early-pruning means that the query planner can “prune” (i.e., ignore) partitions that would not be searched in a query before generating query plans. This helps improve performance time as it eliminates the generation of query plans of partitions that would not be searched.

Conversely, late-pruning means that the query planner prunes partitions after generating query plans for each partition. (The constraint\_exclusion configuration parameter controls late-pruning.)

The ability to early-prune depends upon the nature of the query in the WHERE clause. Early-pruning can be utilized in only simple queries with constraints of the type WHERE column = literal (e.g., WHERE deptno = 10).

Early-pruning is not used for more complex queries such as WHERE column = expression (e.g., WHERE deptno = 10 + 5).

#### Reporting and Logging / What to Log

The configuration parameters in this section control reporting and logging.

##### trace\_hints

Parameter Type: Boolean

Default Value: false

Range: {true | false}

Minimum Scope of Effect: Per session

When Value Changes Take Effect: Immediate

Required Authorization to Activate: Session user

Use with the optimizer hints feature to provide more detailed information regarding whether or not a hint was used by the planner. Set the client\_min\_messages and trace\_hints configuration parameters as follows:

SET client\_min\_messages TO info;

SET trace\_hints TO true;

The SELECT command with the NO\_INDEX hint shown below illustrates the additional information produced when the aforementioned configuration parameters are set.

EXPLAIN SELECT /\*+ NO\_INDEX(accounts accounts\_pkey) \*/ \* FROM accounts WHERE aid = 100;

INFO: [HINTS] Index Scan of [accounts].[accounts\_pkey] rejected because of NO\_INDEX hint.

INFO: [HINTS] Bitmap Heap Scan of [accounts].[accounts\_pkey] rejected because of NO\_INDEX hint.

QUERY PLAN

-------------------------------------------------------------

Seq Scan on accounts (cost=0.00..14461.10 rows=1 width=97)

Filter: (aid = 100)

(2 rows)

#### EnterpriseDB Auditing Settings

This section describes configuration parameters used by the Advanced Server database auditing feature.

##### edb\_audit

Parameter Type: Enum

Default Value: none

Range: {none | csv | xml}

Minimum Scope of Effect: Cluster

When Value Changes Take Effect: Reload

Required Authorization to Activate: PPAS service account

Enables or disables database auditing. The values xml or csv will enable database auditing. These values represent the file format in which auditing information will be captured. none will disable database auditing and is also the default.

##### edb\_audit\_directory

Parameter Type: String

Default Value: edb\_audit

Range: n/a

Minimum Scope of Effect: Cluster

When Value Changes Take Effect: Reload

Required Authorization to Activate: PPAS service account

Specifies the directory where the audit log files will be created. The path of the directory can be absolute or relative to the POSTGRES\_PLUS\_HOME/data directory.

##### edb\_audit\_filename

Parameter Type: String

Default Value: audit-%Y%m%d\_%H%M%S

Range: n/a

Minimum Scope of Effect: Cluster

When Value Changes Take Effect: Reload

Required Authorization to Activate: PPAS service account

Specifies the file name of the audit file where the auditing information will be stored. The default file name will be audit-%Y%m%d\_%H%M%S. The escape sequences, %Y, %m etc., will be replaced by the appropriate current values according to the system date and time.

##### edb\_audit\_rotation\_day

Parameter Type: String

Default Value: every

Range: {none | every | sun | mon | tue | wed | thu | fri | sat} ...

Minimum Scope of Effect: Cluster

When Value Changes Take Effect: Reload

Required Authorization to Activate: PPAS service account

Specifies the day of the week on which to rotate the audit files. Valid values are sun, mon, tue, wed, thu, fri, sat, every, and none. To disable rotation, set the value to none. To rotate the file every day, set the edb\_audit\_rotation\_day value to every. To rotate the file on a specific day of the week, set the value to the desired day of the week.

##### edb\_audit\_rotation\_size

Parameter Type: Integer

Default Value: 0MB

Range: 0MB to 5000MB

Minimum Scope of Effect: Cluster

When Value Changes Take Effect: Reload

Required Authorization to Activate: PPAS service account

Specifies a file size threshold in megabytes when file rotation will be forced to occur. The default value is 0MB. If the parameter is commented out or set to 0, rotation of the file on a size basis will not occur.

##### edb\_audit\_rotation\_seconds

Parameter Type: Integer

Default Value: 0s

Range: 0s to 2147483647s

Minimum Scope of Effect: Cluster

When Value Changes Take Effect: Reload

Required Authorization to Activate: PPAS service account

Specifies the rotation time in seconds when a new log file should be created. To disable this feature, set this parameter to 0.

##### edb\_audit\_connect

Parameter Type: Enum

Default Value: failed

Range: {none | failed | all}

Minimum Scope of Effect: Cluster

When Value Changes Take Effect: Reload

Required Authorization to Activate: PPAS service account

Enables auditing of database connection attempts by users. To disable auditing of all connection attempts, set edb\_audit\_connect to none. To audit all failed connection attempts, set the value to failed. To audit all connection attempts, set the value to all.

##### edb\_audit\_disconnect

Parameter Type: Enum

Default Value: none

Range: {none | all}

Minimum Scope of Effect: Cluster

When Value Changes Take Effect: Reload

Required Authorization to Activate: PPAS service account

Enables auditing of database disconnections by connected users. To enable auditing of disconnections, set the value to all. To disable, set the value to none.

##### edb\_audit\_statement

Parameter Type: String

Default Value: ddl, error

Range: {none | ddl | dml | select | error | rollback | all} ...

Minimum Scope of Effect: Cluster

When Value Changes Take Effect: Reload

Required Authorization to Activate: PPAS service account

This configuration parameter is used to specify auditing of different categories of SQL statements. To audit statements resulting in error, set the parameter value to error. To audit DDL statements such as CREATE TABLE, ALTER TABLE, etc., set the parameter value to ddl. Modification statements such as INSERT, UPDATE, DELETE or TRUNCATE can be audited by setting edb\_audit\_statement to dml. Setting the value to all will audit every statement while none disables this feature.

##### edb\_audit\_tag

Parameter Type: String

Default Value: none

Minimum Scope of Effect: Session

When Value Changes Take Effect: Immediate

Required Authorization to Activate: User

Use edb\_audit\_tag to specify a string value that will be included in the audit log when the edb\_audit parameter is set to csv or xml.

#### Client Connection Defaults / Locale and Formatting

This section describes configuration parameters affecting locale and formatting.

##### icu\_short\_form

Parameter Type: String

Default Value: none

Range: n/a

Minimum Scope of Effect: Database

When Value Changes Take Effect: n/a

Required Authorization to Activate: n/a

The configuration parameter icu\_short\_form is a parameter containing the default ICU short form name assigned to a database or to the Advanced Server instance. See Section 2.3 for general information about the ICU short form and the Unicode Collation Algorithm.

This configuration parameter is set either when the CREATE DATABASE command is used with the ICU\_SHORT\_FORM parameter (see Section 2.3.3.2) in which case the specified short form name is set and appears in the icu\_short\_form configuration parameter when connected to this database, or when an Advanced Server instance is created with the initdb command used with the --icu\_short\_form option (see Section 2.3.3.3) in which case the specified short form name is set and appears in the icu\_short\_form configuration parameter when connected to a database in that Advanced Server instance, and the database does not override it with its own ICU\_SHORT\_FORM parameter with a different short form.

Once established in the manner described, the icu\_short\_form configuration parameter cannot be changed.

#### Client Connection Defaults / Statement Behavior

This section describes configuration parameters affecting statement behavior.

##### default\_heap\_fillfactor

Parameter Type: Integer

Default Value: 100

Range: 10 to 100

Minimum Scope of Effect: Per session

When Value Changes Take Effect: Immediate

Required Authorization to Activate: Session user

Sets the fillfactor for a table when the FILLFACTOR storage parameter is omitted from a CREATE TABLE command.

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.

#### Client Connection Defaults / Other Defaults

The parameters in this section set other miscellaneous client connection defaults.

##### oracle\_home

Parameter Type: String

Default Value: none

Range: n/a

Minimum Scope of Effect: Cluster

When Value Changes Take Effect: Restart

Required Authorization to Activate: PPAS service account

Before creating an Oracle Call Interface (OCI) database link to an Oracle server, you must direct Advanced Server to the correct Oracle home directory. Set the LD\_LIBRARY\_PATH environment variable on Linux (or PATH on Windows) to the lib directory of the Oracle client installation directory.

For Windows only, you can instead set the value of the oracle\_home configuration parameter in the postgresql.conf file. The value specified in the oracle\_home configuration parameter will override the Windows PATH environment variable.

The LD\_LIBRARY\_PATH environment variable on Linux (PATH environment variable or oracle\_home configuration parameter on Windows) must be set properly each time you start Advanced Server.

For Windows only: To set the oracle\_home configuration parameter in the postgresql.conf file, edit the file, adding the following line:

oracle\_home = 'lib\_directory'

Substitute the name of the Windows directory that contains oci.dll for lib\_directory.

After setting the oracle\_home configuration parameter, you must restart the server for the changes to take effect. Restart the server from the Windows Services console.

##### odbc\_lib\_path

Parameter Type: String

Default Value: none

Range: n/a

Minimum Scope of Effect: Cluster

When Value Changes Take Effect: Restart

Required Authorization to Activate: PPAS service account

If you will be using an ODBC driver manager, and if it is installed in a non-standard location, you specify the location by setting the odbc\_lib\_path configuration parameter in the postgresql.conf file:

odbc\_lib\_path = 'complete\_path\_to\_libodbc.so'

The configuration file must include the complete pathname to the driver manager shared library (typically libodbc.so).

#### Compatibility Options

The configuration parameters described in this section control various database compatibility features.

##### edb\_redwood\_date

Parameter Type: Boolean

Default Value: false

Range: {true | false}

Minimum Scope of Effect: Per session

When Value Changes Take Effect: Immediate

Required Authorization to Activate: Session user

When DATE appears as the data type of a column in the commands, it is translated to TIMESTAMP(0) at the time the table definition is stored in the database if the configuration parameter edb\_redwood\_date is set to TRUE. Thus, a time component will also be stored in the column along with the date.

If edb\_redwood\_date is set to FALSE the column’s data type in a CREATE TABLE or ALTER TABLE command remains as a native PostgreSQL DATE data type and is stored as such in the database. The PostgreSQL DATE data type stores only the date without a time component in the column.

Regardless of the setting of edb\_redwood\_date, when DATE appears as a data type in any other context such as the data type of a variable in an SPL declaration section, or the data type of a formal parameter in an SPL procedure or SPL function, or the return type of an SPL function, it is always internally translated to a TIMESTAMP(0) and thus, can handle a time component if present.

##### edb\_redwood\_greatest\_least

Parameter Type: Boolean

Default Value: true

Range: {true | false}

Minimum Scope of Effect: Per session

When Value Changes Take Effect: Immediate

Required Authorization to Activate: Session user

The GREATEST function returns the parameter with the greatest value from its list of parameters. The LEAST function returns the parameter with the least value from its list of parameters.

When edb\_redwood\_greatest\_least is set to TRUE, the GREATEST and LEAST functions return null when at least one of the parameters is null.

SET edb\_redwood\_greatest\_least TO on;

SELECT GREATEST(1, 2, NULL, 3);

greatest

----------

(1 row)

When edb\_redwood\_greatest\_least is set to FALSE, null parameters are ignored except when all parameters are null in which case null is returned by the functions.

SET edb\_redwood\_greatest\_least TO off;

SELECT GREATEST(1, 2, NULL, 3);

greatest

----------

3

(1 row)

SELECT GREATEST(NULL, NULL, NULL);

greatest

----------

(1 row)

##### edb\_redwood\_raw\_names

Parameter Type: Boolean

Default Value: false

Range: {true | false}

Minimum Scope of Effect: Per session

When Value Changes Take Effect: Immediate

Required Authorization to Activate: Session user

When edb\_redwood\_raw\_names is set to its default value of FALSE, database object names such as table names, column names, trigger names, program names, user names, etc. appear in uppercase letters when viewed from Redwood catalogs (that is, system catalogs prefixed by ALL\_, DBA\_, or USER\_. See Chapter 10 for a list of such catalogs). In addition, quotation marks enclose names that were created with enclosing quotation marks.

When edb\_redwood\_raw\_names is set to TRUE, the database object names are displayed exactly as they are stored in the PostgreSQL system catalogs when viewed from the Redwood catalogs. Thus, names created without enclosing quotation marks appear in lowercase as expected in PostgreSQL. Names created with enclosing quotation marks appear exactly as they were created, but without the quotation marks.

For example, the following user name is created, and then a session is started with that user.

CREATE USER reduser IDENTIFIED BY password;

edb=# \c - reduser

Password for user reduser:

You are now connected to database "edb" as user "reduser".

When connected to the database as reduser, the following tables are created.

CREATE TABLE all\_lower (col INTEGER);

CREATE TABLE ALL\_UPPER (COL INTEGER);

CREATE TABLE "Mixed\_Case" ("Col" INTEGER);

When viewed from the Redwood catalog, USER\_TABLES, with edb\_redwood\_raw\_names set to the default value FALSE, the names appear in uppercase except for the Mixed\_Case name, which appears as created and also with enclosing quotation marks.

edb=> SELECT \* FROM USER\_TABLES;

schema\_name | table\_name | tablespace\_name | status | temporary

-------------+--------------+-----------------+--------+-----------

REDUSER | ALL\_LOWER | | VALID | N

REDUSER | ALL\_UPPER | | VALID | N

REDUSER | "Mixed\_Case" | | VALID | N

(3 rows)

When viewed with edb\_redwood\_raw\_names set to TRUE, the names appear in lowercase except for the Mixed\_Case name, which appears as created, but now without the enclosing quotation marks.

edb=> SET edb\_redwood\_raw\_names TO true;

SET

edb=> SELECT \* FROM USER\_TABLES;

schema\_name | table\_name | tablespace\_name | status | temporary

-------------+------------+-----------------+--------+-----------

reduser | all\_lower | | VALID | N

reduser | all\_upper | | VALID | N

reduser | Mixed\_Case | | VALID | N

(3 rows)

These names now match the case when viewed from the PostgreSQL pg\_tables catalog.

edb=> SELECT schemaname, tablename, tableowner FROM pg\_tables WHERE tableowner = 'reduser';

schemaname | tablename | tableowner

------------+------------+------------

reduser | all\_lower | reduser

reduser | all\_upper | reduser

reduser | Mixed\_Case | reduser

(3 rows)

##### edb\_redwood\_strings

Parameter Type: Boolean

Default Value: false

Range: {true | false}

Minimum Scope of Effect: Per session

When Value Changes Take Effect: Immediate

Required Authorization to Activate: Session user

If the edb\_redwood\_strings parameter is set to TRUE, when a string is concatenated with a null variable or null column, the result is the original string. If edb\_redwood\_strings is set to FALSE, the native PostgreSQL behavior is maintained, which is the concatenation of a string with a null variable or null column gives a null result.

The following example illustrates the difference.

The sample application contains a table of employees. This table has a column named comm that is null for most employees. The following query is run with edb\_redwood\_string set to FALSE. The concatenation of a null column with non-empty strings produces a final result of null, so only employees that have a commission appear in the query result. The output line for all other employees is null.

SET edb\_redwood\_strings TO off;

SELECT RPAD(ename,10) || ' ' || TO\_CHAR(sal,'99,999.99') || ' ' || TO\_CHAR(comm,'99,999.99') "EMPLOYEE COMPENSATION" FROM emp;

EMPLOYEE COMPENSATION

----------------------------------

ALLEN 1,600.00 300.00

WARD 1,250.00 500.00

MARTIN 1,250.00 1,400.00

TURNER 1,500.00 .00

(14 rows)

The following is the same query executed when edb\_redwood\_strings is set to TRUE. Here, the value of a null column is treated as an empty string. The concatenation of an empty string with a non-empty string produces the non-empty string.

SET edb\_redwood\_strings TO on;

SELECT RPAD(ename,10) || ' ' || TO\_CHAR(sal,'99,999.99') || ' ' || TO\_CHAR(comm,'99,999.99') "EMPLOYEE COMPENSATION" FROM emp;

EMPLOYEE COMPENSATION

----------------------------------

SMITH 800.00

ALLEN 1,600.00 300.00

WARD 1,250.00 500.00

JONES 2,975.00

MARTIN 1,250.00 1,400.00

BLAKE 2,850.00

CLARK 2,450.00

SCOTT 3,000.00

KING 5,000.00

TURNER 1,500.00 .00

ADAMS 1,100.00

JAMES 950.00

FORD 3,000.00

MILLER 1,300.00

(14 rows)

##### edb\_stmt\_level\_tx

Parameter Type: Boolean

Default Value: false

Range: {true | false}

Minimum Scope of Effect: Per session

When Value Changes Take Effect: Immediate

Required Authorization to Activate: Session user

The term statement level transaction isolation describes the behavior whereby when a runtime error occurs in a SQL command, all the updates on the database caused by that single command are rolled back. For example, if a single UPDATE command successfully updates five rows, but an attempt to update a sixth row results in an exception, the updates to all six rows made by this UPDATE command are rolled back. The effects of prior SQL commands that have not yet been committed or rolled back are pending until a COMMIT or ROLLBACK command is executed.

In Advanced Server, if an exception occurs while executing a SQL command, all the updates on the database since the start of the transaction are rolled back. In addition, the transaction is left in an aborted state and either a COMMIT or ROLLBACK command must be issued before another transaction can be started.

If edb\_stmt\_level\_tx is set to TRUE, then an exception will not automatically roll back prior uncommitted database updates. If edb\_stmt\_level\_tx is set to FALSE, then an exception will roll back uncommitted database updates.

Note: Use edb\_stmt\_level\_tx set to TRUE only when absolutely necessary, as this may cause a negative performance impact.

The following example run in PSQL shows that when edb\_stmt\_level\_tx is FALSE, the abort of the second INSERT command also rolls back the first INSERT command. Note that in PSQL, the command \set AUTOCOMMIT off must be issued, otherwise every statement commits automatically defeating the purpose of this demonstration of the effect of edb\_stmt\_level\_tx.

\set AUTOCOMMIT off

SET edb\_stmt\_level\_tx TO off;

INSERT INTO emp (empno,ename,deptno) VALUES (9001, 'JONES', 40);

INSERT INTO emp (empno,ename,deptno) VALUES (9002, 'JONES', 00);

ERROR: insert or update on table "emp" violates foreign key constraint "emp\_ref\_dept\_fk"

DETAIL: Key (deptno)=(0) is not present in table "dept".

COMMIT;

SELECT empno, ename, deptno FROM emp WHERE empno > 9000;

empno | ename | deptno

-------+-------+--------

(0 rows)

In the following example, with edb\_stmt\_level\_tx set to TRUE, the first INSERT command has not been rolled back after the error on the second INSERT command. At this point, the first INSERT command can either be committed or rolled back.

\set AUTOCOMMIT off

SET edb\_stmt\_level\_tx TO on;

INSERT INTO emp (empno,ename,deptno) VALUES (9001, 'JONES', 40);

INSERT INTO emp (empno,ename,deptno) VALUES (9002, 'JONES', 00);

ERROR: insert or update on table "emp" violates foreign key constraint "emp\_ref\_dept\_fk"

DETAIL: Key (deptno)=(0) is not present in table "dept"

SELECT empno, ename, deptno FROM emp WHERE empno > 9000;

empno | ename | deptno

-------+-------+--------

9001 | JONES | 40

(1 row)

COMMIT;

A ROLLBACK command could have been issued instead of the COMMIT command in which case the insert of employee number 9001 would have been rolled back as well.

##### db\_dialect

Parameter Type: Enum

Default Value: postgres

Range: {postgres | redwood}

Minimum Scope of Effect: Per session

When Value Changes Take Effect: Immediate

Required Authorization to Activate: Session user

In addition to the native PostgreSQL system catalog, pg\_catalog, Advanced Server contains extended catalog views (see Chapter 10) as well as system catalogs compatible with Microsoft® SQL Server®. These are sys for the expanded catalog views and dbo for SQL Server. The db\_dialect parameter controls the order in which these catalogs are searched for name resolution.

When set to postgres, the namespace precedence is pg\_catalog, sys, then dbo, giving the PostgreSQL catalog the highest precedence. When set to redwood, the namespace precedence is sys, dbo, then pg\_catalog, giving the expanded catalog views the highest precedence.

##### default\_with\_rowids

Parameter Type: Boolean

Default Value: false

Range: {true | false}

Minimum Scope of Effect: Per session

When Value Changes Take Effect: Immediate

Required Authorization to Activate: Session user

When set to on, CREATE TABLE includes a ROWID column in newly created tables, which can then be referenced in SQL commands.

##### optimizer\_mode

Parameter Type: Enum

Default Value: choose

Range: {choose | ALL\_ROWS | FIRST\_ROWS | FIRST\_ROWS\_10 | FIRST\_ROWS\_100 | FIRST\_ROWS\_1000}

Minimum Scope of Effect: Per session

When Value Changes Take Effect: Immediate

Required Authorization to Activate: Session user

Sets the default optimization mode for analyzing optimizer hints.

The following table shows the possible values:

Table ‑ - Optimizer Modes

| Hint | Description |
| --- | --- |
| ALL\_ROWS | Optimizes for retrieval of all rows of the result set. |
| CHOOSE | Does no default optimization based on assumed number of rows to be retrieved from the result set. This is the default. |
| FIRST\_ROWS | Optimizes for retrieval of only the first row of the result set. |
| FIRST\_ROWS\_10 | Optimizes for retrieval of the first 10 rows of the results set. |
| FIRST\_ROWS\_100 | Optimizes for retrieval of the first 100 rows of the result set. |
| FIRST\_ROWS\_1000 | Optimizes for retrieval of the first 1000 rows of the result set. |

These optimization modes are based upon the assumption that the client submitting the SQL command is interested in viewing only the first “n” rows of the result set and will then abandon the remainder of the result set. Resources allocated to the query are adjusted as such.

#### Customized Options

In previous releases of Advanced Server, the custom\_variable\_classes was required by those parameters not normally known to be added by add-on modules (such as procedural languages).

##### custom\_variable\_classes

The custom\_variable\_classes parameter is deprecated in Advanced Server 9.2; parameters that previously depended on this parameter no longer require its support.

##### dbms\_alert.max\_alerts

Parameter Type: Integer

Default Value: 100

Range: 0 to 500

Minimum Scope of Effect: Cluster

When Value Changes Take Effect: Restart

Required Authorization to Activate: PPAS service account

Specifies the maximum number of concurrent alerts allowed on a system using the DBMS\_ALERTS package.

##### dbms\_pipe.total\_message\_buffer

Parameter Type: Integer

**Default Value:** 30 Kb

**Range:** 30 Kb to 256 Kb

**Minimum Scope of Effect:** Postmaster

**When Value Changes Take Effect:** Restart

**Required Authorization to Activate:** PPAS service account

Specifies the total size of the buffer used for the DBMS\_PIPE package.

##### index\_advisor.enabled

Parameter Type: Boolean

Default Value: true

Range: {true | false}

Minimum Scope of Effect: Per session

When Value Changes Take Effect: Immediate

Required Authorization to Activate: Session user

Provides the capability to temporarily suspend Index Advisor in an EDB-PSQL or PSQL session. The Index Advisor plugin, index\_advisor, must be loaded in the EDB-PSQL or PSQL session in order to use the index\_advisor.enabled configuration parameter.

The Index Advisor plugin can be loaded as shown by the following example:

$ psql -d edb -U enterprisedb

Password for user enterprisedb:

psql (9.5.0.0)

Type "help" for help.

edb=# LOAD 'index\_advisor';

LOAD

Use the SET command to change the parameter setting to control whether or not Index Advisor generates an alternative query plan as shown by the following example:

edb=# SET index\_advisor.enabled TO off;

SET

edb=# EXPLAIN SELECT \* FROM t WHERE a < 10000;

QUERY PLAN

-------------------------------------------------------

Seq Scan on t (cost=0.00..1693.00 rows=9864 width=8)

Filter: (a < 10000)

(2 rows)

edb=# SET index\_advisor.enabled TO on;

SET

edb=# EXPLAIN SELECT \* FROM t WHERE a < 10000;

QUERY PLAN

-----------------------------------------------------------------------------

Seq Scan on t (cost=0.00..1693.00 rows=9864 width=8)

Filter: (a < 10000)

Result (cost=0.00..327.88 rows=9864 width=8)

One-Time Filter: '===[ HYPOTHETICAL PLAN ]==='::text

-> Index Scan using "<hypothetical-index>:1" on t (cost=0.00..327.88 rows=9864 width=8)

Index Cond: (a < 10000)

(6 rows)

##### edb\_sql\_protect.enabled

Parameter Type: Boolean

Default Value: false

Range: {true | false}

Minimum Scope of Effect: Cluster

When Value Changes Take Effect: Reload

Required Authorization to Activate: PPAS service account

Controls whether or not SQL/Protect is actively monitoring protected roles by analyzing SQL statements issued by those roles and reacting according to the setting of edb\_sql\_protect.level. When you are ready to begin monitoring with SQL/Protect set this parameter to on.

##### edb\_sql\_protect.level

Parameter Type: Enum

Default Value: passive

Range: {learn | passive | active}

Minimum Scope of Effect: Cluster

When Value Changes Take Effect: Reload

Required Authorization to Activate: PPAS service account

Sets the action taken by SQL/Protect when a SQL statement is issued by a protected role.

The edb\_sql\_protect.level configuration parameter can be set to one of the following values to use either learn mode, passive mode, or active mode:

* learn. Tracks the activities of protected roles and records the relations used by the roles. This is used when initially configuring SQL/Protect so the expected behaviors of the protected applications are learned.
* passive. Issues warnings if protected roles are breaking the defined rules, but does not stop any SQL statements from executing. This is the next step after SQL/Protect has learned the expected behavior of the protected roles. This essentially behaves in intrusion detection mode and can be run in production when properly monitored.
* active. Stops all invalid statements for a protected role. This behaves as a SQL firewall preventing dangerous queries from running. This is particularly effective against early penetration testing when the attacker is trying to determine the vulnerability point and the type of database behind the application. Not only does SQL/Protect close those vulnerability points, but it tracks the blocked queries allowing administrators to be alerted before the attacker finds an alternate method of penetrating the system.

If you are using SQL/Protect for the first time, set edb\_sql\_protect.level to learn.

##### edb\_sql\_protect.max\_protected\_relations

Parameter Type: Integer

Default Value: 1024

Range: 1 to 2147483647

Minimum Scope of Effect: Cluster

When Value Changes Take Effect: Restart

Required Authorization to Activate: PPAS service account

Sets the maximum number of relations that can be protected per role. Please note the total number of protected relations for the server will be the number of protected relations times the number of protected roles. Every protected relation consumes space in shared memory. The space for the maximum possible protected relations is reserved during database server startup.

If the server is started when edb\_sql\_protect.max\_protected\_relations is set to a value outside of the valid range (for example, a value of 2,147,483,648), then a warning message is logged in the database server log file:

2014-07-18 16:04:12 EDT WARNING: invalid value for parameter "edb\_sql\_protect.max\_protected\_relations": "2147483648"

2014-07-18 16:04:12 EDT HINT: Value exceeds integer range.

The database server starts successfully, but with edb\_sql\_protect.max\_protected\_relations set to the default value of 1024.

Though the upper range for the parameter is listed as the maximum value for an integer data type, the practical setting depends on how much shared memory is available and the parameter value used during database server startup.

As long as the space required can be reserved in shared memory, the value will be acceptable. If the value is such that the space in shared memory cannot be reserved, the database server startup fails with an error message such as the following:

2014-07-18 15:22:17 EDT FATAL: could not map anonymous shared memory: Cannot allocate memory

2014-07-18 15:22:17 EDT HINT: This error usually means that PostgreSQL's request for a shared memory segment exceeded available memory, swap space or huge pages. To reduce the request size (currently 2070118400 bytes), reduce PostgreSQL's shared memory usage, perhaps by reducing shared\_buffers or max\_connections.

In such cases, reduce the parameter value until the database server can be started successfully.

##### edb\_sql\_protect.max\_protected\_roles

Parameter Type: Integer

Default Value: 64

Range: 1 to 2147483647

Minimum Scope of Effect: Cluster

When Value Changes Take Effect: Restart

Required Authorization to Activate: PPAS service account

Sets the maximum number of roles that can be protected.

Every protected role consumes space in shared memory. Please note that the server will reserve space for the number of protected roles times the number of protected relations (edb\_sql\_protect.max\_protected\_relations). The space for the maximum possible protected roles is reserved during database server startup.

If the database server is started when edb\_sql\_protect.max\_protected\_roles is set to a value outside of the valid range (for example, a value of 2,147,483,648), then a warning message is logged in the database server log file:

2014-07-18 16:04:12 EDT WARNING: invalid value for parameter "edb\_sql\_protect.max\_protected\_roles": "2147483648"

2014-07-18 16:04:12 EDT HINT: Value exceeds integer range.

The database server starts successfully, but with edb\_sql\_protect.max\_protected\_roles set to the default value of 64.

Though the upper range for the parameter is listed as the maximum value for an integer data type, the practical setting depends on how much shared memory is available and the parameter value used during database server startup.

As long as the space required can be reserved in shared memory, the value will be acceptable. If the value is such that the space in shared memory cannot be reserved, the database server startup fails with an error message such as the following:

2014-07-18 15:22:17 EDT FATAL: could not map anonymous shared memory: Cannot allocate memory

2014-07-18 15:22:17 EDT HINT: This error usually means that PostgreSQL's request for a shared memory segment exceeded available memory, swap space or huge pages. To reduce the request size (currently 2070118400 bytes), reduce PostgreSQL's shared memory usage, perhaps by reducing shared\_buffers or max\_connections.

In such cases, reduce the parameter value until the database server can be started successfully.

##### edb\_sql\_protect.max\_queries\_to\_save

Parameter Type: Integer

Default Value: 5000

Range: 100 to 2147483647

Minimum Scope of Effect: Cluster

When Value Changes Take Effect: Restart

Required Authorization to Activate: PPAS service account

Sets the maximum number of offending queries to save in view edb\_sql\_protect\_queries.

Every query that is saved consumes space in shared memory. The space for the maximum possible queries that can be saved is reserved during database server startup.

If the database server is started when edb\_sql\_protect.max\_queries\_to\_save is set to a value outside of the valid range (for example, a value of 10), then a warning message is logged in the database server log file:

2014-07-18 13:05:31 EDT WARNING: 10 is outside the valid range for parameter "edb\_sql\_protect.max\_queries\_to\_save" (100 .. 2147483647)

The database server starts successfully, but with edb\_sql\_protect.max\_queries\_to\_save set to the default value of 5000.

Though the upper range for the parameter is listed as the maximum value for an integer data type, the practical setting depends on how much shared memory is available and the parameter value used during database server startup.

As long as the space required can be reserved in shared memory, the value will be acceptable. If the value is such that the space in shared memory cannot be reserved, the database server startup fails with an error message such as the following:

2014-07-18 15:22:17 EDT FATAL: could not map anonymous shared memory: Cannot allocate memory

2014-07-18 15:22:17 EDT HINT: This error usually means that PostgreSQL's request for a shared memory segment exceeded available memory, swap space or huge pages. To reduce the request size (currently 2070118400 bytes), reduce PostgreSQL's shared memory usage, perhaps by reducing shared\_buffers or max\_connections.

In such cases, reduce the parameter value until the database server can be started successfully.

##### edbldr.empty\_csv\_field

Parameter Type: Enum

Default Value: NULL

Range: {NULL | empty\_string | pgsql}

Minimum Scope of Effect: Per session

When Value Changes Take Effect: Immediate

Required Authorization to Activate: Session user

Use the edbldr.empty\_csv\_field parameter to specify how EDB\*Loader will treat an empty string. The valid values for the edbldr.empty\_csv\_field parameter are:

|  |  |
| --- | --- |
| Parameter Setting | EDB\*Loader Behavior |
| NULL | An empty field is treated as NULL. |
| empty\_string | An empty field is treated as a string of length zero. |
| pgsql | An empty field is treated as a NULL if it does not contain quotes and as an empty string if it contains quotes. |

For more information about the edbldr.empty\_csv\_field parameter in EDB\*Loader, see Section 6.1.9.

##### utl\_encode.uudecode\_redwood

Parameter Type: Boolean

Default Value: false

Range: {true | false}

Minimum Scope of Effect: Per session

When Value Changes Take Effect: Immediate

Required Authorization to Activate: Session user

When set to TRUE, Advanced Server’s UTL\_ENCODE.UUDECODE function can decode uuencoded data that was created by the Oracle implementation of the UTL\_ENCODE.UUENCODE function.

When set to the default setting of FALSE, Advanced Server’s UTL\_ENCODE.UUDECODE function can decode uuencoded data created by Advanced Server’s UTL\_ENCODE.UUENCODE function.

##### utl\_file.umask

Parameter Type: String

Default Value: 0077

Range: Octal digits for umask settings

Minimum Scope of Effect: Per session

When Value Changes Take Effect: Immediate

Required Authorization to Activate: Session user

The utl\_file.umask parameter sets the file mode creation mask or simply, the mask, in a manner similar to the Linux umask command. This is for usage only within the Advanced Server UTL\_FILE package.

Note: The utl\_file.umask parameter is not supported on Windows systems.

The value specified for utl\_file.umask is a 3 or 4-character octal string that would be valid for the Linux umask command. The setting determines the permissions on files created by the UTL\_FILE functions and procedures. (Refer to any information source regarding Linux or Unix systems for information on file permissions and the usage of the umask command.)

The following shows the results of the default utl\_file.umask setting of 0077. Note that all permissions are denied on users belonging to the enterprisedb group as well as all other users. Only user enterprisedb has read and write permissions on the file.

-rw------- 1 enterprisedb enterprisedb 21 Jul 24 16:08 utlfile

For an example of using utl\_file.umask, see Section 9.17.1.

#### Ungrouped

Configuration parameters in this section apply to Advanced Server only and are for a specific, limited purpose.

##### nls\_length\_semantics

Parameter Type: Enum

Default Value: byte

Range: {byte | char}

Minimum Scope of Effect: Per session

When Value Changes Take Effect: Immediate

Required Authorization to Activate: Superuser

This parameter has no effect in Advanced Server.

For example, the form of the ALTER SESSION command is accepted in Advanced Server without throwing a syntax error, but does not alter the session environment:

ALTER SESSION SET nls\_length\_semantics = char;

Note: Since the setting of this parameter has no effect on the server environment, it does not appear in the system view pg\_settings.

##### query\_rewrite\_enabled

Parameter Type: Enum

Default Value: false

Range: {true | false | force}

Minimum Scope of Effect: Per session

When Value Changes Take Effect: Immediate

Required Authorization to Activate: Session user

This parameter has no effect in Advanced Server.

For example, the following form of the ALTER SESSION command is accepted in Advanced Server without throwing a syntax error, but does not alter the session environment:

ALTER SESSION SET query\_rewrite\_enabled = force;

Note: Since the setting of this parameter has no effect on the server environment, it does not appear in the system view pg\_settings.

##### query\_rewrite\_integrity

Parameter Type: Enum

Default Value: enforced

Range: {enforced | trusted | stale\_tolerated}

Minimum Scope of Effect: Per session

When Value Changes Take Effect: Immediate

Required Authorization to Activate: Superuser

This parameter has no effect in Advanced Server.

For example, the following form of the ALTER SESSION command is accepted in Advanced Server without throwing a syntax error, but does not alter the session environment:

ALTER SESSION SET query\_rewrite\_integrity = stale\_tolerated;

Note: Since the setting of this parameter has no effect on the server environment, it does not appear in the system view pg\_settings.

##### timed\_statistics

Parameter Type: Boolean

Default Value: true

Range: {true | false}

Minimum Scope of Effect: Per session

When Value Changes Take Effect: Immediate

Required Authorization to Activate: Session user

Controls the collection of timing data for the Dynamic Runtime Instrumentation Tools Architecture (DRITA) feature. When set to on, timing data is collected.

Note: When Advanced Server is installed, the postgresql.conf file contains an explicit entry setting timed\_statistics to off. If this entry is commented out letting timed\_statistics to default, and the configuration file is reloaded, timed statistics collection would be turned on.

## Controlling the Audit Logs

Advanced Server allows database and security administrators, auditors, and operators to track and analyze database activities using the audit logs. The audit logs can be configured to information such as:

* When a role establishes a connection to an Advanced Server database
* What database objects a role creates, modifies, or deletes when connected to Advanced Server
* When any failed authentication attempts occur

You can use parameters specified in the postgresql.conf file to control the information included in the audit logs.

### Auditing Configuration Parameters

Use the following postgresql.conf configuration parameters to control database auditing:

edb\_audit

Enables or disables database auditing. The values xml or csv will enable database auditing. These values represent the file format in which auditing information will be captured. none will disable database auditing and is also the default. This option can only be set at server start or in the postgresql.conf file.

edb\_audit\_directory

Specifies the directory where the log files will be created. The path of the directory can be relative or absolute to the data folder. This option can only be set at server start or in the postgresql.conf configuration file.

edb\_audit\_filename

Specifies the file name of the audit file where the auditing information will be stored. The default file name will be audit-%Y%m%d\_%H%M%S. The escape sequences, %Y, %m etc., will be replaced by the appropriate current values according to the system date and time. This option can only be set at server start or in the postgresql.conf configuration file.

edb\_audit\_rotation\_day

Specifies the day of the week on which to rotate the audit files. Valid values are sun, mon, tue, wed, thu, fri, sat, every, and none. To disable rotation, set the value to none. To rotate the file every day, set the edb\_audit\_rotation\_day value to every. To rotate the file on a specific day of the week, set the value to the desired day of the week. every is the default value. This option can only be set at server start or in the postgresql.conf configuration file.

edb\_audit\_rotation\_size

Specifies a file size threshold in megabytes when file rotation will be forced to occur. The default value is 0 MB. If the parameter is commented out or set to 0, rotation of the file on a size basis will not occur. This option can only be set at server start or in the postgresql.conf configuration file.

edb\_audit\_rotation\_seconds

Specifies the rotation time in seconds when a new log file should be created. To disable this feature, set this parameter to 0. This option can only be set at server start or in the postgresql.conf configuration file.

edb\_audit\_connect

Enables auditing of database connection attempts by users. To disable auditing of all connection attempts, set edb\_audit\_connect to none. To audit all failed connection attempts, set the value to failed. To audit all connection attempts, set the value to all. This option can only be set at server start or in the postgresql.conf configuration file.

edb\_audit\_disconnect

Enables auditing of database disconnections by connected users. To enable auditing of disconnections, set the value to all. To disable, set the value to none. This option can only be set at server start or in the postgresql.conf configuration file.

edb\_audit\_statement

This configuration parameter is used to specify auditing of different categories of SQL statements. To audit statements resulting in error, set the parameter value to error. To audit DDL statements such as CREATE TABLE, ALTER TABLE, etc., set the parameter value to ddl. Modification statements such as INSERT, UPDATE, DELETE or TRUNCATE can be audited by setting edb\_audit\_statement to dml. To audit ROLLBACK statements, set the parameter value to rollback. Setting the value to all will audit every statement while none disables this feature. This option can only be set at server start or in the postgresql.conf configuration file.

edb\_audit\_tag

Use this configuration parameter is used to specify a string value that will be included in the audit log when the edb\_audit parameter is set to csv or xml.

The following steps describe how to configure Advanced Server to log all connections, disconnections, DDL statements, and any statements resulting in an error. The resulting audit file will rotate every Sunday.

1. Enable auditing by the setting the edb\_audit parameter to xml or csv.
2. Set the file rotation day when the new file will be created by setting the parameter edb\_audit\_rotation\_day to sun.
3. To audit all connections, set the parameter, edb\_audit\_connect, to all.
4. To audit all disconnections, set the parameter, edb\_audit\_disconnect, to all.
5. To audit all DDL statements and error statements, set the parameter, edb\_audit\_statement, to ddl, error.

The following is the CSV and XML output when auditing is enabled:

CSV Audit Log File

,,,1452,,,2008-03-13 12:40:02 ,startup,"AUDIT: database system is ready"

enterprisedb,mgmtsvr,127.0.0.1(1266),1620,47d9595b.654,0,2008-03-13 12:42:03 ,connect,"AUDIT: connection authorized: user=enterprisedb database=mgmtsvr"

enterprisedb,mgmtsvr,127.0.0.1(1266),1620,47d9595b.654,1588,2008-03-13 12:42:08 ,ddl,"AUDIT: statement: drop table HILOSEQUENCES

"

enterprisedb,mgmtsvr,127.0.0.1(1266),1620,47d9595b.654,1590,2008-03-13 12:42:09 ,ddl,"AUDIT: statement: create table HILOSEQUENCES (

SEQUENCENAME varchar(50) not null,

HIGHVALUES integer not null,

constraint hilo\_pk primary key (SEQUENCENAME)

)

"

enterprisedb,edb,127.0.0.1(1269),904,47d9598d.388,0,2008-03-13 12:42:53 ,connect,"AUDIT: connection authorized: user=enterprisedb database=edb"

enterprisedb,edb,127.0.0.1(1269),904,47d9598d.388,1618,2008-03-13 12:43:02 ,ddl,"AUDIT: statement: CREATE TABLE test (f1 INTEGER);"

enterprisedb,edb,127.0.0.1(1269),904,47d9598d.388,1620,2008-03-13 12:43:02 ,sql statement,"AUDIT: statement: SELECT \* FROM testx;"

enterprisedb,edb,127.0.0.1(1269),904,47d9598d.388,1620,2008-03-13 12:43:02 ,error,"ERROR: relation "testx" does not exist"

enterprisedb,edb,127.0.0.1(1269),904,47d9598d.388,1621,2008-03-13 12:43:04 ,ddl,"AUDIT: statement: DROP TABLE test;"

enterprisedb,edb,127.0.0.1(1269),904,47d9598d.388,0,2008-03-13 12:43:20 ,disconnect,"AUDIT: disconnection: session time: 0:00:26.953 user=enterprisedb database=edb host=127.0.0.1 port=1269"

enterprisedb,mgmtsvr,127.0.0.1(1266),1620,47d9595b.654,0,2008-03-13 12:43:29 ,disconnect,"AUDIT: disconnection: session time: 0:01:26.594 user=enterprisedb database=mgmtsvr host=127.0.0.1 port=1266"

,,,3148,,,2008-03-13 12:43:35 ,shutdown,"AUDIT: database system is shut down"

XML Audit Log File

<event process\_id="2516" time="2008-03-13 13:22:42 " type="startup">

<message>AUDIT: database system is ready</message>

</event>

<event user="enterprisedb" database="mgmtsvr" remote\_host\_and\_port="127.0.0.1(1281)"

process\_id="352" session\_id="47d96338.160" transaction="0"

time="2008-03-13 13:24:08 " type="connect">

<message>AUDIT: connection authorized: user=enterprisedb

database=mgmtsvr</message>

</event>

<event user="enterprisedb" database="mgmtsvr" remote\_host\_and\_port="127.0.0.1(1281)"

process\_id="352" session\_id="47d96338.160" transaction="1635"

time="2008-03-13 13:24:10 " type="ddl">

<command>AUDIT: statement: drop table HILOSEQUENCES</command>

</event>

<event user="enterprisedb" database="mgmtsvr" remote\_host\_and\_port="127.0.0.1(1281)"

process\_id="352" session\_id="47d96338.160" transaction="1637"

time="2008-03-13 13:24:10 " type="ddl">

<command>AUDIT: statement: create table HILOSEQUENCES (

SEQUENCENAME varchar(50) not null,

HIGHVALUES integer not null,

constraint hilo\_pk primary key (SEQUENCENAME)

)</command>

</event>

<event user="enterprisedb" database="edb" remote\_host\_and\_port="127.0.0.1(1283)"

process\_id="3776" session\_id="47d96378.ec0" transaction="0"

time="2008-03-13 13:25:12 " type="connect">

<message>AUDIT: connection authorized: user=enterprisedb database=edb</message>

</event>

<event user="enterprisedb" database="edb" remote\_host\_and\_port="127.0.0.1(1283)"

process\_id="3776" session\_id="47d96378.ec0" transaction="1667"

time="2008-03-13 13:25:17 " type="ddl">

<command>AUDIT: statement: CREATE TABLE test (f1 INTEGER);</command>

</event>

<event user="enterprisedb" database="edb" remote\_host\_and\_port="127.0.0.1(1283)"

process\_id="3776" session\_id="47d96378.ec0" transaction="1669"

time="2008-03-13 13:25:17 " type="sql statement">

<command>AUDIT: statement: SELECT \* FROM testx;</command>

</event>

<event user="enterprisedb" database="edb" remote\_host\_and\_port="127.0.0.1(1283)"

process\_id="3776" session\_id="47d96378.ec0" transaction="1669"

time="2008-03-13 13:25:17 " type="error">

<message>ERROR: relation &quot;testx&quot; does not exist</message>

</event>

<event user="enterprisedb" database="edb" remote\_host\_and\_port="127.0.0.1(1283)"

process\_id="3776" session\_id="47d96378.ec0" transaction="1670"

time="2008-03-13 13:25:18 " type="ddl">

<command>AUDIT: statement: DROP TABLE test;</command>

</event>

<event user="enterprisedb" database="edb" remote\_host\_and\_port="127.0.0.1(1283)"

process\_id="3776" session\_id="47d96378.ec0" transaction="0"

time="2008-03-13 13:25:22 " type="disconnect">

<message>AUDIT: disconnection: session time: 0:00:10.094 user=enterprisedb

database=edb host=127.0.0.1 port=1283</message>

</event>

<event user="enterprisedb" database="mgmtsvr" remote\_host\_and\_port="127.0.0.1(1281)"

process\_id="352" session\_id="47d96338.160" transaction="0"

time="2008-03-13 13:25:31 " type="disconnect">

<message>AUDIT: disconnection: session time: 0:01:23.046 user=enterprisedb

database=mgmtsvr host=127.0.0.1 port=1281</message>

</event>

<event process\_id="2768" time="2008-03-13 13:25:36 " type="shutdown">

<message>AUDIT: database system is shut down</message>

</event>

## Unicode Collation Algorithm

The Unicode Collation Algorithm (UCA) is a specification (Unicode Technical Report #10) that defines a customizable method of collating and comparing Unicode data. Collation means how data is sorted as with a SELECT … ORDER BY clause. Comparison is relevant for searches that use ranges with less than, greater than, or equal to operators.

Customizability is an important factor for various reasons such as the following.

* Unicode supports a vast number of languages. Letters that may be common to several languages may be expected to collate in different orders depending upon the language.
* Characters that appear with letters in certain languages such as accents or umlauts have an impact on the expected collation depending upon the language.
* In some languages, combinations of several consecutive characters should be treated as a single character with regards to its collation sequence.
* There may be certain preferences as to the collation of letters according to case. For example, should the lowercase form of a letter collate before the uppercase form of the same letter or vice versa.
* There may be preferences as to whether punctuation marks such as underscore characters, hyphens, or space characters should be considered in the collating sequence or should they simply be ignored as if they did not exist in the string.

Given all of these variations with the vast number of languages supported by Unicode, there is a necessity for a method to select the specific criteria for determining a collating sequence. This is what the Unicode Collation Algorithm defines.

Note: In addition, another advantage for using ICU collations (the implementation of the Unicode Collation Algorithm) is for performance. Sorting tasks, including B-tree index creation, can complete in less than half the time it takes with a non-ICU collation. The exact performance gain depends on your operating system version, the language of your text data, and other factors.

The following sections provide a brief, simplified explanation of the Unified Collation Algorithm concepts. As the algorithm and its usage are quite complex with numerous variations, refer to the official documents cited in these sections for complete details.

### Basic Unicode Collation Algorithm Concepts

The official information for the Unicode Collation Algorithm is specified in Unicode Technical Report #10, which can be found on The Unicode Consortium website at:

<http://www.unicode.org/reports/tr10/>

The ICU – International Components for Unicode also provides much useful information. An explanation of the collation concepts can be found on their website located at:

<http://userguide.icu-project.org/collation/concepts>

The basic concept behind the Unicode Collation Algorithm is the use of multilevel comparison. This means that a number of levels are defined, which are listed as level 1 through level 5 in the following bullet points. Each level defines a type of comparison. Strings are first compared using the primary level, also called level 1.

If the order can be determined based on the primary level, then the algorithm is done. If the order cannot be determined based on the primary level, then the secondary level, level 2, is applied. If the order can be determined based on the secondary level, then the algorithm is done, otherwise the tertiary level is applied, and so on. There is typically, a final tie-breaking level to determine the order if it cannot be resolved by the prior levels.

* Level 1 – Primary Level for Base Characters. The order of basic characters such as letters and digits determines the difference such as A < B.
* Level 2 – Secondary Level for Accents. If there are no primary level differences, then the presence or absence of accents and other such characters determine the order such as a < á.
* Level 3 – Tertiary Level for Case. If there are no primary level or secondary level differences, then a difference in case determines the order such as a < A.
* Level 4 – Quaternary Level for Punctuation. If there are no primary, secondary, or tertiary level differences, then the presence or absence of white space characters, control characters, and punctuation determine the order such as -A < A.
* Level 5 – Identical Level for Tie-Breaking. If there are no primary, secondary, tertiary, or quaternary level differences, then some other difference such as the code point values determines the order.

### International Components for Unicode

The Unicode Collation Algorithm is implemented by open source software provided by the International Components for Unicode (ICU). The software is a set of C/C++ and Java libraries.

When Advanced Server is used to create a collation that invokes the ICU components to produce the collation, the result is referred to as an ICU collation.

#### Locale Collations

When creating a collation for a locale, a predefined ICU short form name for the given locale is typically provided.

An ICU short form is a method of specifying collation attributes, which are the properties of a collation. Section 2.3.2.2 provides additional information on collation attributes.

There are predefined ICU short forms for locales. The ICU short form for a locale incorporates the collation attribute settings typically used for the given locale. This simplifies the collation creation process by eliminating the need to specify the entire list of collation attributes for that locale.

The system catalog pg\_catalog.pg\_icu\_collate\_names contains a list of the names of the ICU short forms for locales. The ICU short form name is listed in column icu\_short\_form.

edb=# SELECT icu\_short\_form, valid\_locale FROM pg\_icu\_collate\_names ORDER BY valid\_locale;

icu\_short\_form | valid\_locale

----------------+--------------

LAF | af

LAR | ar

LAS | as

LAZ | az

LBE | be

LBG | bg

LBN | bn

LBS | bs

LBS\_ZCYRL | bs\_Cyrl

LROOT | ca

LROOT | chr

LCS | cs

LCY | cy

LDA | da

LROOT | de

LROOT | dz

LEE | ee

LEL | el

LROOT | en

LROOT | en\_US

LEN\_RUS\_VPOSIX | en\_US\_POSIX

LEO | eo

LES | es

LET | et

LFA | fa

LFA\_RAF | fa\_AF

.

.

.

If needed, the default characteristics of an ICU short form for a given locale can be overridden by specifying the collation attributes to override that property. This is discussed in the next section.

#### Collation Attributes

When creating an ICU collation, the desired characteristics of the collation must be specified. As discussed in Section 2.3.2.1, this can typically be done with an ICU short form for the desired locale. However, if more specific information is required, the specification of the collation properties can be done by using collation attributes.

Collation attributes define the rules of how characters are to be compared for determining the collation sequence of text strings. As Unicode covers a vast set of languages in numerous variations according to country, territory and culture, these collation attributes are quite complex.

For the complete, precise meaning and usage of collation attributes, see Section 13 “Collator Naming Scheme” on the ICU – International Components for Unicode website at:

<http://userguide.icu-project.org/collation/concepts>

The following is a brief summary of the collation attributes and how they are specified using the ICU short form method

Each collation attribute is represented by an uppercase letter, which are listed in the following bullet points. The possible valid values for each attribute are given by codes shown within the parentheses. Some codes have general meanings for all attributes. X means to set the attribute off. O means to set the attribute on. D means to set the attribute to its default value.

* A – Alternate (N, S, D). Handles treatment of variable characters such as white spaces, punctuation marks, and symbols. When set to non-ignorable (N), differences in variable characters are treated with the same importance as differences in letters. When set to shifted (S), then differences in variable characters are of minor importance (that is, the variable character is ignored when comparing base characters).
* C – Case First (X, L, U, D). Controls whether a lowercase letter sorts before the same uppercase letter (L), or the uppercase letter sorts before the same lowercase letter (U). Off (X) is typically specified when lowercase first (L) is desired.
* E – Case Level (X, O, D). Set in combination with the Strength attribute, the Case Level attribute is used when accents are to be ignored, but not case.
* F – French Collation (X, O, D). When set to on, secondary differences (presence of accents) are sorted from the back of the string as done in the French Canadian locale.
* H – Hiragana Quaternary (X, O, D). Introduces an additional level to distinguish between the Hiragana and Katakana characters for compatibility with the JIS X 4061 collation of Japanese character strings.
* N – Normalization Checking (X, O, D). Controls whether or not text is thoroughly normalized for comparison. Normalization deals with the issue of canonical equivalence of text whereby different code point sequences represent the same character, which then present issues when sorting or comparing such characters. Languages such as Arabic, ancient Greek, Hebrew, Hindi, Thai, or Vietnamese should be used with Normalization Checking set to on.
* S – Strength (1, 2, 3, 4, I, D). Maximum collation level used for comparison. Influences whether accents or case are taken into account when collating or comparing strings. Each number represents a level. A setting of I represents identical strength (that is, level 5).
* T – Variable Top (hexadecimal digits). Applicable only when the Alternate attribute is not set to non-ignorable (N). The hexadecimal digits specify the highest character sequence that is to be considered ignorable. For example, if white space is to be ignorable, but visible variable characters are not to be ignorable, then Variable Top set to 0020 would be specified along with the Alternate attribute set to S and the Strength attribute set to 3. (The space character is hexadecimal 0020. Other non-visible variable characters such as backspace, tab, line feed, carriage return, etc. have values less than 0020. All visible punctuation marks have values greater than 0020.)

A set of collation attributes and their values is represented by a text string consisting of the collation attribute letter concatenated with the desired attribute value. Each attribute/value pair is joined to the next pair with an underscore character as shown by the following example.

AN\_CX\_EX\_FX\_HX\_NO\_S3

Collation attributes can be specified along with a locale’s ICU short form name to override those default attribute settings of the locale.

The following is an example where the ICU short form named LROOT is modified with a number of other collation attribute/value pairs.

AN\_CX\_EX\_LROOT\_NO\_S3

In the preceding example, the Alternate attribute (A) is set to non-ignorable (N). The Case First attribute (C) is set to off (X). The Case Level attribute (E) is set to off (X). The Normalization attribute (N) is set to on (O). The Strength attribute (S) is set to the tertiary level 3. LROOT is the ICU short form to which these other attributes are applying modifications.

### Creating an ICU Collation

Creating an ICU collation can be done a number of different ways.

* When creating a new database cluster with the initdb command, the --icu-short-form option can be specified to define the ICU collation to be used by default by all databases in the cluster.
* When creating a new database with the CREATE DATABASE command, the ICU\_SHORT\_FORM parameter can be specified to define the ICU collation to be used by default in that database.
* In an existing database, the CREATE COLLATION command can be used with the ICU\_SHORT\_FORM parameter to define an ICU collation to be used under specific circumstances such as when assigned with the COLLATE clause onto selected columns of certain tables or when appended with the COLLATE clause onto an expression such as ORDER BY expr COLLATE "collation\_name".

The following describes the various ways of creating an ICU collation.

#### CREATE COLLATION

Use the ICU\_SHORT\_FORM parameter with the CREATE COLLATION command to create an ICU collation:

CREATE COLLATION *collation\_name* (

[ LOCALE = locale, ]

[ LC\_COLLATE = lc\_collate, ]

[ LC\_CTYPE = lc\_ctype, ]

[ ICU\_SHORT\_FORM = icu\_short\_form ]

);

To be able to create a collation, you must have CREATE privilege on the destination schema where the collation is to reside.

For information about the general usage of the CREATE COLLATION command, please refer to the PostgreSQL core documentation available at:

<http://www.postgresql.org/docs/9.5/static/sql-createcollation.html>

UTF-8 character encoding of the database is required. Any LOCALE, or LC\_COLLATE and LC\_CTYPE settings that are accepted with UTF-8 encoding can be used.

Parameters

collation\_name

The name of the collation to be added. The collation\_name may be schema-qualified.

locale

The locale to be used. Short cut for setting LC\_COLLATE and LC\_TYPE. If LOCALE is specified, then LC\_COLLATE and LC\_TYPE must be omitted.

lc\_collate

The collation to be used. If LC\_CTYPE is specified, then LC\_COLLATE must also be specified and LOCALE must be omitted.

lc\_ctype

The character classification to be used. If LC\_COLLATE is specified, then LC\_CTYPE must also be specified and LOCALE must be omitted.

icu\_short\_form

The text string specifying the collation attributes and their settings. This typically consists of an ICU short form name, possibly appended with additional collation attribute/value pairs. A list of ICU short form names is available from column icu\_short\_form in system catalog pg\_catalog.pg\_icu\_collate\_names.

Example

The following creates a collation using the LROOT ICU short form.

edb=# CREATE COLLATION icu\_collate\_a (LOCALE = 'en\_US.UTF8', ICU\_SHORT\_FORM = 'LROOT');

CREATE COLLATION

The definition of the new collation can be seen with the following psql command.

edb=# \dO

List of collations

Schema | Name | Collate | Ctype | ICU

--------------+---------------+------------+------------+-------

enterprisedb | icu\_collate\_a | en\_US.UTF8 | en\_US.UTF8 | LROOT

(1 row)

#### CREATE DATABASE

The following is the syntax for creating a database with an ICU collation:

CREATE DATABASE *database\_name*

[ [ WITH ] [ OWNER [=] user\_name ]

[ TEMPLATE [=] template ]

[ ENCODING [=] encoding ]

[ LC\_COLLATE [=] lc\_collate ]

[ LC\_CTYPE [=] lc\_ctype ]

[ TABLESPACE [=] tablespace\_name ]

[ CONNECTION LIMIT [=] connlimit ]

[ ICU\_SHORT\_FORM [=] icu\_short\_form ]];

For complete information about the general usage, syntax, and parameters of the CREATE DATABASE command, please refer to the PostgreSQL core documentation available at:

<http://www.postgresql.org/docs/9.5/static/sql-createdatabase.html>

When using the CREATE DATABASE command to create a database using an ICU collation, the TEMPLATE template0 clause must be specified and the database encoding must be UTF-8.

The following is an example of creating a database using the LROOT ICU short form collation, but sorts an uppercase form of a letter before its lowercase counterpart (CU) and treats variable characters as non-ignorable (AN).

CREATE DATABASE collation\_db

TEMPLATE template0

ENCODING 'UTF8'

ICU\_SHORT\_FORM = 'AN\_CU\_EX\_NX\_LROOT';

The following psql command shows the newly created database.

edb=# \l collation\_db

List of databases

Name | Owner | Encoding | Collate | Ctype | ICU | Access privileges

--------------+--------------+----------+-------------+-------------+-------------------+-------------------

collation\_db | enterprisedb | UTF8 | en\_US.UTF-8 | en\_US.UTF-8 | AN\_CU\_EX\_NX\_LROOT |

(1 row)

The following table is created and populated with rows in the database.

CREATE TABLE collate\_tbl (

id INTEGER,

c2 VARCHAR(2)

);

INSERT INTO collate\_tbl VALUES (1, 'A');

INSERT INTO collate\_tbl VALUES (2, 'B');

INSERT INTO collate\_tbl VALUES (3, 'C');

INSERT INTO collate\_tbl VALUES (4, 'a');

INSERT INTO collate\_tbl VALUES (5, 'b');

INSERT INTO collate\_tbl VALUES (6, 'c');

INSERT INTO collate\_tbl VALUES (7, '1');

INSERT INTO collate\_tbl VALUES (8, '2');

INSERT INTO collate\_tbl VALUES (9, '.B');

INSERT INTO collate\_tbl VALUES (10, '-B');

INSERT INTO collate\_tbl VALUES (11, ' B');

The following query shows that the uppercase form of a letter sorts before the lowercase form of the same base letter, and in addition, variable characters are taken into account when sorted as they appear at the beginning of the sort list. (The default behavior for en\_US.UTF-8 is to sort the lowercase form of a letter before the uppercase form of the same base letter, and to ignore variable characters.)

collation\_db=# SELECT id, c2 FROM collate\_tbl ORDER BY c2;

id | c2

----+----

11 | B

10 | -B

9 | .B

7 | 1

8 | 2

1 | A

4 | a

2 | B

5 | b

3 | C

6 | c

(11 rows)

#### initdb

A database cluster can be created with a default ICU collation for all databases in the cluster by using the --icu-short-form option with the initdb command.

For complete information about the general usage, syntax, and parameters of the initdb command, please refer to the PostgreSQL core documentation available at:

<http://www.postgresql.org/docs/9.5/static/app-initdb.html>

The following illustrates this process.

$ su enterprisedb

Password:

$ /opt/PostgresPlus/9.5AS/bin/initdb -U enterprisedb -D /tmp/collation\_data --encoding UTF8 --icu-short-form 'AN\_CU\_EX\_NX\_LROOT'

The files belonging to this database system will be owned by user "enterprisedb".

This user must also own the server process.

The database cluster will be initialized with locale "en\_US.UTF-8".

The default text search configuration will be set to "english".

Data page checksums are disabled.

creating directory /tmp/collation\_data ... ok

creating subdirectories ... ok

.

.

.

Success. You can now start the database server using:

/opt/PostgresPlus/9.5AS/bin/edb-postgres -D /tmp/collation\_data

or

/opt/PostgresPlus/9.5AS/bin/pg\_ctl -D /tmp/collation\_data -l logfile start

The following shows the databases created in the cluster which all have an ICU collation of AN\_CU\_EX\_NX\_LROOT.

edb=# \l

List of databases

Name | Owner | Encoding | Collate | Ctype | ICU | Access privileges

-----------+--------------+----------+-------------+-------------+-------------------+-------------------------------

edb | enterprisedb | UTF8 | en\_US.UTF-8 | en\_US.UTF-8 | AN\_CU\_EX\_NX\_LROOT |

postgres | enterprisedb | UTF8 | en\_US.UTF-8 | en\_US.UTF-8 | AN\_CU\_EX\_NX\_LROOT |

template0 | enterprisedb | UTF8 | en\_US.UTF-8 | en\_US.UTF-8 | AN\_CU\_EX\_NX\_LROOT | =c/enterprisedb +

| | | | | | enterprisedb=CTc/enterprisedb

template1 | enterprisedb | UTF8 | en\_US.UTF-8 | en\_US.UTF-8 | AN\_CU\_EX\_NX\_LROOT | =c/enterprisedb +

| | | | | | enterprisedb=CTc/enterprisedb

(4 rows)

### Using a Collation

A newly defined ICU collation can be used anywhere the COLLATION "collation\_name" clause can be used in a SQL command such as in the column specifications of the CREATE TABLE command or appended to an expression in the ORDER BY clause of a SELECT command.

The following are some examples of the creation and usage of ICU collations based on the English language in the United States (en\_US.UTF8).

In these examples, ICU collations are created with the following characteristics.

Collation icu\_collate\_lowercase forces the lowercase form of a letter to sort before its uppercase counterpart (CL).

Collation icu\_collate\_uppercase forces the uppercase form of a letter to sort before its lowercase counterpart (CU).

Collation icu\_collate\_ignore\_punct causes variable characters (white space and punctuation marks) to be ignored during sorting (AS).

Collation icu\_collate\_ignore\_white\_sp causes white space and other non-visible variable characters to be ignored during sorting, but visible variable characters (punctuation marks) are not ignored (AS, T0020).

CREATE COLLATION icu\_collate\_lowercase (

LOCALE = 'en\_US.UTF8',

ICU\_SHORT\_FORM = 'AN\_CL\_EX\_NX\_LROOT'

);

CREATE COLLATION icu\_collate\_uppercase (

LOCALE = 'en\_US.UTF8',

ICU\_SHORT\_FORM = 'AN\_CU\_EX\_NX\_LROOT'

);

CREATE COLLATION icu\_collate\_ignore\_punct (

LOCALE = 'en\_US.UTF8',

ICU\_SHORT\_FORM = 'AS\_CX\_EX\_NX\_LROOT\_L3'

);

CREATE COLLATION icu\_collate\_ignore\_white\_sp (

LOCALE = 'en\_US.UTF8',

ICU\_SHORT\_FORM = 'AS\_CX\_EX\_NX\_LROOT\_L3\_T0020'

);

Note: When creating collations, ICU may generate notice and warning messages when attributes are given to modify the LROOT collation.

The following psql command lists the collations.

edb=# \dO

List of collations

Schema | Name | Collate | Ctype | ICU

--------------+-----------------------------+------------+------------+----------------------------

enterprisedb | icu\_collate\_ignore\_punct | en\_US.UTF8 | en\_US.UTF8 | AS\_CX\_EX\_NX\_LROOT\_L3

enterprisedb | icu\_collate\_ignore\_white\_sp | en\_US.UTF8 | en\_US.UTF8 | AS\_CX\_EX\_NX\_LROOT\_L3\_T0020

enterprisedb | icu\_collate\_lowercase | en\_US.UTF8 | en\_US.UTF8 | AN\_CL\_EX\_NX\_LROOT

enterprisedb | icu\_collate\_uppercase | en\_US.UTF8 | en\_US.UTF8 | AN\_CU\_EX\_NX\_LROOT

(4 rows)

The following table is created and populated.

CREATE TABLE collate\_tbl (

id INTEGER,

c2 VARCHAR(2)

);

INSERT INTO collate\_tbl VALUES (1, 'A');

INSERT INTO collate\_tbl VALUES (2, 'B');

INSERT INTO collate\_tbl VALUES (3, 'C');

INSERT INTO collate\_tbl VALUES (4, 'a');

INSERT INTO collate\_tbl VALUES (5, 'b');

INSERT INTO collate\_tbl VALUES (6, 'c');

INSERT INTO collate\_tbl VALUES (7, '1');

INSERT INTO collate\_tbl VALUES (8, '2');

INSERT INTO collate\_tbl VALUES (9, '.B');

INSERT INTO collate\_tbl VALUES (10, '-B');

INSERT INTO collate\_tbl VALUES (11, ' B');

The following query sorts on column c2 using the default collation. Note that variable characters (white space and punctuation marks) with id column values of 9, 10, and 11 are ignored and sort with the letter B.

edb=# SELECT \* FROM collate\_tbl ORDER BY c2;

id | c2

----+----

7 | 1

8 | 2

4 | a

1 | A

5 | b

2 | B

11 | B

10 | -B

9 | .B

6 | c

3 | C

(11 rows)

The following query sorts on column c2 using collation icu\_collate\_lowercase, which forces the lowercase form of a letter to sort before the uppercase form of the same base letter. Also note that the AN attribute forces variable characters to be included in the sort order at the same level when comparing base characters so rows with id values of 9, 10, and 11 appear at the beginning of the sort list before all letters and numbers.

edb=# SELECT \* FROM collate\_tbl ORDER BY c2 COLLATE "icu\_collate\_lowercase";

id | c2

----+----

11 | B

10 | -B

9 | .B

7 | 1

8 | 2

4 | a

1 | A

5 | b

2 | B

6 | c

3 | C

(11 rows)

The following query sorts on column c2 using collation icu\_collate\_uppercase, which forces the uppercase form of a letter to sort before the lowercase form of the same base letter.

edb=# SELECT \* FROM collate\_tbl ORDER BY c2 COLLATE "icu\_collate\_uppercase";

id | c2

----+----

11 | B

10 | -B

9 | .B

7 | 1

8 | 2

1 | A

4 | a

2 | B

5 | b

3 | C

6 | c

(11 rows)

The following query sorts on column c2 using collation icu\_collate\_ignore\_punct, which causes variable characters to be ignored so rows with id values of 9, 10, and 11 sort with the letter B as that is the character immediately following the ignored variable character.

edb=# SELECT \* FROM collate\_tbl ORDER BY c2 COLLATE "icu\_collate\_ignore\_punct";

id | c2

----+----

7 | 1

8 | 2

4 | a

1 | A

5 | b

11 | B

2 | B

9 | .B

10 | -B

6 | c

3 | C

(11 rows)

The following query sorts on column c2 using collation icu\_collate\_ignore\_white\_sp. The AS and T0020 attributes of the collation cause variable characters with code points less than or equal to hexadecimal 0020 to be ignored while variable characters with code points greater than hexadecimal 0020 are included in the sort.

The row with id value of 11, which starts with a space character (hexadecimal 0020) sorts with the letter B. The rows with id values of 9 and 10, which start with visible punctuation marks greater than hexadecimal 0020, appear at the beginning of the sort list as these particular variable characters are included in the sort order at the same level when comparing base characters.

edb=# SELECT \* FROM collate\_tbl ORDER BY c2 COLLATE "icu\_collate\_ignore\_white\_sp";

id | c2

----+----

10 | -B

9 | .B

7 | 1

8 | 2

4 | a

1 | A

5 | b

11 | B

2 | B

6 | c

3 | C

(11 rows)

## Profile Management

Advanced Server 9.5 allows a database superuser to create named *profiles*. Each profile defines rules for password management that augment password and md5 authentication. The rules in a profile can:

* count failed login attempts
* lock an account due to excessive failed login attempts
* mark a password for expiration
* define a grace period after a password expiration
* define rules for password complexity
* define rules that limit password re-use

A profile is a named set of password attributes that allow you to easily manage a group of roles that share comparable authentication requirements. If the password requirements change, you can modify the profile to have the new requirements applied to each user that is associated with that profile.

After creating the profile, you can associate the profile with one or more users. When a user connects to the server, the server enforces the profile that is associated with their login role. Profiles are shared by all databases within a cluster, but each cluster may have multiple profiles. A single user with access to multiple databases will use the same profile when connecting to each database within the cluster.

Advanced Server 9.5 creates a profile named default that is associated with a new role when the role is created unless an alternate profile is specified. If you upgrade to Advanced Server 9.5 from a previous server version, existing roles will automatically be assigned to the default profile. You cannot delete the default profile.

The default profile specifies the following attributes:

FAILED\_LOGIN\_ATTEMPTS UNLIMITED

PASSWORD\_LOCK\_TIME UNLIMITED

PASSWORD\_LIFE\_TIME UNLIMITED

PASSWORD\_GRACE\_TIME UNLIMITED

PASSWORD\_REUSE\_TIME UNLIMITED

PASSWORD\_REUSE\_MAX UNLIMITED

PASSWORD\_VERIFY\_FUNCTION NULL

A database superuser can use the ALTER PROFILE command to modify the values specified by the default profile. For more information about modifying a profile, see Section 2.4.2.

### Creating a New Profile

Use the CREATE PROFILE command to create a new profile. The syntax is:

CREATE PROFILE *profile*\_*name*   
 [LIMIT {*parameter* *value*} ... ];

Include the LIMIT clause and one or more space-delimited *parameter*/*value* pairs to specify the rules enforced by Advanced Server.

**Parameters:**

profile\_name specifies the name of the profile.

*parameter* specifies the attribute limited by the profile.

*value* specifies the parameter limit.

Advanced Server supports the *value* shown below for each *parameter*:

FAILED\_LOGIN\_ATTEMPTS specifies the number of failed login attempts that a user may make before the server locks the user out of their account for the length of time specified by PASSWORD\_LOCK\_TIME. Supported values are:

* An INTEGER value greater than 0.
* DEFAULT - the value of FAILED\_LOGIN\_ATTEMPTS specified in the DEFAULT profile.
* UNLIMITED – the connecting user may make an unlimited number of failed login attempts.

PASSWORD\_LOCK\_TIME specifies the length of time that must pass before the server unlocks an account that has been locked because of FAILED\_LOGIN\_ATTEMPTS. Supported values are:

* A NUMERIC value greater than or equal to 0. To specify a fractional portion of a day, specify a decimal value. For example, use the value 4.5 to specify 4 days, 12 hours.
* DEFAULT - the value of PASSWORD\_LOCK\_TIME specified in the DEFAULT profile.
* UNLIMITED – the account is locked until it is manually unlocked by a database superuser.

PASSWORD\_LIFE\_TIME specifies the number of days that the current password may be used before the user is prompted to provide a new password. Include the PASSWORD\_GRACE\_TIME clause when using the PASSWORD\_LIFE\_TIME clause to specify the number of days that will pass after the password expires before connections by the role are rejected. If PASSWORD\_GRACE\_TIME is not specified, the password will expire on the day specified by the default value of PASSWORD\_GRACE\_TIME, and the user will not be allowed to execute any command until a new password is provided. Supported values are:

* A NUMERIC value greater than or equal to 0. To specify a fractional portion of a day, specify a decimal value. For example, use the value 4.5 to specify 4 days, 12 hours.
* DEFAULT - the value of PASSWORD\_LIFE\_TIME specified in the DEFAULT profile.
* UNLIMITED – The password does not have an expiration date.

PASSWORD\_GRACE\_TIME specifies the length of the grace period after a password expires until the user is forced to change their password. When the grace period expires, a user will be allowed to connect, but will not be allowed to execute any command until they update their expired password. Supported values are:

* A NUMERIC value greater than or equal to 0. To specify a fractional portion of a day, specify a decimal value. For example, use the value 4.5 to specify 4 days, 12 hours.
  + DEFAULT - the value of PASSWORD\_GRACE\_TIME specified in the DEFAULT profile.
  + UNLIMITED – The grace period is infinite.

PASSWORD\_REUSE\_TIME specifies the number of days a user must wait before re-using a password. The PASSWORD\_REUSE\_TIME and PASSWORD\_REUSE\_MAX parameters are intended to be used together. If you specify a finite value for one of these parameters while the other is UNLIMITED, old passwords can never be reused. If both parameters are set to UNLIMITED there are no restrictions on password reuse. Supported values are:

* A NUMERIC value greater than or equal to 0. To specify a fractional portion of a day, specify a decimal value. For example, use the value 4.5 to specify 4 days, 12 hours.
* DEFAULT - the value of PASSWORD\_REUSE\_TIME specified in the DEFAULT profile.
* UNLIMITED – The password can be re-used without restrictions.

PASSWORD\_REUSE\_MAX specifies the number of password changes that must occur before a password can be reused. The PASSWORD\_REUSE\_TIME and PASSWORD\_REUSE\_MAX parameters are intended to be used together. If you specify a finite value for one of these parameters while the other is UNLIMITED, old passwords can never be reused. If both parameters are set to UNLIMITED there are no restrictions on password reuse. Supported values are:

* An INTEGER value greater than or equal to 0.
* DEFAULT - the value of PASSWORD\_REUSE\_MAX specified in the DEFAULT profile.
* UNLIMITED – The password can be re-used without restrictions.

PASSWORD\_VERIFY\_FUNCTION specifies password complexity. Supported values are:

* The name of a PL/SQL function.
* DEFAULT - the value of PASSWORD\_VERIFY\_FUNCTION specified in the DEFAULT profile.
* NULL

**Notes**

Use DROP PROFILE command to remove the profile.

**Examples**

The following command creates a profile named acctg. The profile specifies that if a user has not authenticated with the correct password in five attempts, the account will be locked for one day:

CREATE PROFILE acctg LIMIT  
 FAILED\_LOGIN\_ATTEMPTS 5   
 PASSWORD\_LOCK\_TIME 1;

The following command creates a profile named sales. The profile specifies that a user must change their password every 90 days:

CREATE PROFILE sales LIMIT  
 PASSWORD\_LIFE\_TIME 90  
 PASSWORD\_GRACE\_TIME 3;

If the user has not changed their password before the 90 days specified in the profile has passed, they will be issued a warning at login. After a grace period of 3 days, their account will not be allowed to invoke any commands until they change their password.

The following command creates a profile named accts. The profile specifies that a user cannot re-use a password within 180 days of the last use of the password, and must change their password at least 5 times before re-using the password:

CREATE PROFILE accts LIMIT  
 PASSWORD\_REUSE\_TIME 180  
 PASSWORD\_REUSE\_MAX 5;

The following command creates a profile named resources; the profile calls a user-defined function named password\_rules that will verify that the password provided meets their standards for complexity:

CREATE PROFILE resources LIMIT  
 PASSWORD\_VERIFY\_FUNCTION password\_rules;

#### Creating a Password Function

When specifying PASSWORD\_VERIFY\_FUNCTION, you can provide a customized function that specifies the security rules that will be applied when your users change their password. For example, you can specify rules that stipulate that the new password must be at least *n* characters long, and may not contain a specific value.

The password function has the following signature:

*function*\_*name* (*user*\_*name* VARCHAR2,   
 *new*\_*password* VARCHAR2,  
 *old*\_*password* VARCHAR2) RETURN boolean

**Where:**

*user*\_*name* is the name of the user.

*new*\_*password* is the new password.

*old*\_*password* is the user's previous password. If you reference this parameter within your function:

When a database superuser changes their password, the third parameter will always be NULL.

When a user with the CREATEROLE attribute changes their password, the parameter will pass the previous password if the statement includes the REPLACE clause. Note that the REPLACE clause is optional syntax for a user with the CREATEROLE privilege.

When a user that is not a database superuser and does not have the CREATEROLE attribute changes their password, the third parameter will contain the previous password for the role.

The function returns a Boolean value. If the function returns true and does not raise an exception, the password is accepted; if the function returns false or raises an exception, the password is rejected. If the function raises an exception, the specified error message is displayed to the user. If the function does not raise an exception, but returns false, the following error message is displayed:

ERROR: password verification for the specified password failed

The function must be owned by a database superuser, and reside in the sys schema.

**Example:**

The following example creates a profile and a custom function; then, the function is associated with the profile. The following CREATE PROFILE command creates a profile named acctg\_pwd\_profile:

CREATE PROFILE acctg\_pwd\_profile;

The following commands create a (schema-qualified) function named verify\_password:

CREATE OR REPLACE FUNCTION sys.verify\_password(user\_name varchar2, new\_password varchar2, old\_password varchar2)   
RETURN boolean IMMUTABLE   
IS   
BEGIN   
 IF (length(new\_password) < 5)   
 THEN   
 raise\_application\_error(-20001, 'too short');   
 END IF;  
  
 IF substring(new\_password FROM old\_password) IS NOT NULL  
 THEN   
 raise\_application\_error(-20002, 'includes old password');   
 END IF;  
  
 RETURN true;   
END;

The function first ensures that the password is at least 5 characters long, and then compares the new password to the old password. If the new password contains fewer than 5 characters, or contains the old password, the function raises an error.

The following statement sets the ownership of the verify\_password function to the enterprisedb database superuser:

ALTER FUNCTION verify\_password(varchar2, varchar2, varchar2) OWNER TO enterprisedb;

Then, the verify\_password function is associated with the profile:

ALTER PROFILE acctg\_pwd\_profile LIMIT PASSWORD\_VERIFY\_FUNCTION verify\_password;

The following statements confirm that the function is working by first creating a test user (alice), and then attempting to associate invalid and valid passwords with her role:

CREATE ROLE alice WITH LOGIN PASSWORD 'temp\_password' PROFILE acctg\_pwd\_profile;

Then, when alice connects to the database and attempts to change her password, she must adhere to the rules established by the profile function. A non-superuser without CREATEROLE must include the REPLACE clause when changing a password:

edb=> ALTER ROLE alice PASSWORD 'hey';

ERROR: missing REPLACE clause

The new password must be at least 5 characters long:

edb=> ALTER USER alice PASSWORD 'hey' REPLACE 'temp\_password';

ERROR: EDB-20001: too short

CONTEXT: edb-spl function verify\_password(character varying,character varying,character varying) line 5 at procedure/function invocation statement

If the new password is acceptable, the command completes without error:

edb=> ALTER USER alice PASSWORD 'hello' REPLACE 'temp\_password';

ALTER ROLE

If alice decides to change her password, the new password must not contain the old password:

edb=> ALTER USER alice PASSWORD 'helloworld' REPLACE 'hello';

ERROR: EDB-20002: includes old password

CONTEXT: edb-spl function verify\_password(character varying,character varying,character varying) line 10 at procedure/function invocation statement

To remove the verify function, set password\_verify\_function to NULL:

ALTER PROFILE acctg\_pwd\_profile LIMIT password\_verify\_function NULL;

Then, all password constraints will be lifted:

edb=# ALTER ROLE alice PASSWORD 'hey';

ALTER ROLE

### Altering a Profile

Use the ALTER PROFILE command to modify a user-defined profile; Advanced Server supports two forms of the command:

ALTER PROFILE *profile*\_*name* RENAME TO *new*\_*name*;

ALTER PROFILE *profile*\_*name*   
 LIMIT {*parameter* *value*}[...];

Include the LIMIT clause and one or more space-delimited *parameter*/*value* pairs to specify the rules enforced by Advanced Server, or use ALTER PROFILE…RENAME TO to change the name of a profile.

**Parameters:**

profile\_name specifies the name of the profile.

*new\_name* specifies the new name of the profile.

*parameter* specifies the attribute limited by the profile.

*value* specifies the parameter limit.

See the table in Section 2.4.1 for a complete list of accepted parameter/value pairs.

**Examples**

The following example modifies a profile named acctg\_profile:

ALTER PROFILE acctg\_profile   
 LIMIT FAILED\_LOGIN\_ATTEMPTS 3 PASSWORD\_LOCK\_TIME 1;

acctg\_profile will count failed connection attempts when a login role attempts to connect to the server. The profile specifies that if a user has not authenticated with the correct password in three attempts, the account will be locked for one day.

The following example changes the name of acctg\_profile to payables\_profile:

ALTER PROFILE acctg\_profile RENAME TO payables\_profile;

### Dropping a Profile

Use the DROP PROFILE command to drop a profile. The syntax is:

DROP PROFILE [IF EXISTS] *profile*\_*name* [CASCADE|RESTRICT];

Include the IF EXISTS clause to instruct the server to not throw an error if the specified profile does not exist. The server will issue a notice if the profile does not exist.

Include the optional CASCADE clause to reassign any users that are currently associated with the profile to the default profile, and then drop the profile. Include the optional RESTRICT clause to instruct the server to not drop any profile that is associated with a role. This is the default behavior.

**Parameters**

profile\_name

The name of the profile being dropped.

**Examples**

The following example drops a profile named acctg\_profile:

DROP PROFILE acctg\_profile CASCADE;

The command first re-associates any roles associated with the acctg\_profile profile with the default profile, and then drops the acctg\_profile profile.

The following example drops a profile named acctg\_profile:

DROP PROFILE acctg\_profile RESTRICT;

The RESTRICT clause in the command instructs the server to not drop acctg\_profile if there are any roles associated with the profile.

### Associating a Profile with an Existing Role

After creating a profile, you can use the ALTER USER… PROFILE or ALTER ROLE… PROFILE command to associate the profile with a role. The command syntax related to profile management functionality is:

ALTER USER|ROLE name [[WITH] option[…]

where option can be the following clauses, compatible with Oracle usage:

PROFILE *profile*\_*name*

| ACCOUNT {LOCK|UNLOCK}

| PASSWORD EXPIRE [AT 'timestamp']

or option can be the following non-compatible clauses:

| PASSWORD SET AT 'timestamp'

| LOCK TIME 'timestamp'  
 | STORE PRIOR PASSWORD {'password' 'timestamp} [, ...]

For information about the administrative clauses of the ALTER USER or ALTER ROLE command that are supported by Advanced Server, please refer to the PostgreSQL core documentation available at:

<http://www.postgresql.org/docs/9.5/static/sql-commands.html>

Only a database superuser can use the ALTER USER|ROLE clauses that enforce profile management. The clauses enforce the following behaviors:

Include the PROFILE clause and a *profile\_name* to associate a pre-defined profile with a role, or to change which pre-defined profile is associated with a user.

Include the ACCOUNT clause and the LOCK or UNLOCK keyword to specify that the user account should be placed in a locked or unlocked state.

Include the LOCK TIME 'timestamp' clause and a date/time value to lock the role at the specified time, and unlock the role at the time indicated by the PASSWORD\_LOCK\_TIME parameter of the profile assigned to this role. If LOCK TIME is used with the ACCOUNT LOCK clause, the role can only be unlocked by a database superuser with the ACCOUNT UNLOCK clause.

Include the PASSWORD EXPIRE clause with the AT 'timestamp' keywords to specify a date/time when the password associated with the role will expire. If you omit the AT 'timestamp' keywords, the password will expire immediately.

Include the PASSWORD SET AT 'timestamp' keywords to set the password modification date to the time specified.

Include the STORE PRIOR PASSWORD {'password' 'timestamp} [, ...] clause to modify the password history, adding the new password and the time the password was set.

Each login role may only have one profile. To discover the profile that is currently associated with a login role, query the profile column of the DBA\_USERS view.

**Parameters**

name

The name of the role with which the specified profile will be associated.

password

The password associated with the role.

profile\_name

The name of the profile that will be associated with the role.

timestamp

The date and time at which the clause will be enforced. When specifying a value for timestamp, enclose the value in single-quotes.

**Examples**

The following command uses the ALTER USER… PROFILE command to associate a profile named acctg with a user named john:

ALTER USER john PROFILE acctg\_profile;

The following command uses the ALTER ROLE… PROFILE command to associate a profile named acctg with a user named john:

ALTER ROLE john PROFILE acctg\_profile;

### Unlocking a Locked Account

A database superuser can use clauses of the ALTER USER|ROLE… command to lock or unlock a role. The syntax is:

ALTER USER|ROLE name   
 ACCOUNT {LOCK|UNLOCK}  
 LOCK TIME 'timestamp'

Include the ACCOUNT LOCK clause to lock a role immediately; when locked, a role’s LOGIN functionality is disabled. When you specify the ACCOUNT LOCK clause without the LOCK TIME clause, the state of the role will not change until a superuser uses the ACCOUNT UNLOCK clause to unlock the role.

Use the ACCOUNT UNLOCK clause to unlock a role.

Use the LOCK TIME 'timestamp' clause to instruct the server to lock the account at the time specified by the given timestamp for the length of time specified by the PASSWORD\_LOCK\_TIME parameter of the profile associated with this role.

Combine the LOCK TIME 'timestamp' clause and the ACCOUNT LOCK clause to lock an account at a specified time until the account is unlocked by a superuser invoking the ACCOUNT UNLOCK clause.

**Parameters**

name

The name of the role that is being locked or unlocked.

timestamp

The date and time at which the role will be locked. When specifying a value for timestamp, enclose the value in single-quotes.

**Note**

This command (available only in Advanced Server) is implemented to support Oracle-styled profile management.

**Examples**

The following example uses the ACCOUNT LOCK clause to lock the role named john. The account will remain locked until the account is unlocked with the ACCOUNT UNLOCK clause:

ALTER ROLE john ACCOUNT LOCK;

The following example uses the ACCOUNT UNLOCK clause to unlock the role named john:

ALTER USER john ACCOUNT UNLOCK;

The following example uses the LOCK TIME 'timestamp' clause to lock the role named john on September 4, 2015:

ALTER ROLE john LOCK TIME ‘September 4 12:00:00 2015’;

The role will remain locked for the length of time specified by the PASSWORD\_LOCK\_TIME parameter.

The following example combines the LOCK TIME 'timestamp' clause and the ACCOUNT LOCK clause to lock the role named john on September 4, 2015:

ALTER ROLE john LOCK TIME ‘September 4 12:00:00 2015’ ACCOUNT LOCK;

The role will remain locked until a database superuser uses the ACCOUNT UNLOCK command to unlock the role.

### Creating a New Role Associated with a Profile

A database superuser can use clauses of the CREATE USER|ROLE command to assign a named profile to a role when creating the role, or to specify profile management details for a role. The command syntax related to profile management functionality is:

CREATE USER|ROLE *name* [[WITH] *option* […]]

where option can be the following clauses compatible with Oracle databases:

PROFILE *profile*\_*name*   
 | ACCOUNT {LOCK|UNLOCK}  
 | PASSWORD EXPIRE [AT 'timestamp']

or option can be the following non-compatible clauses:

| LOCK TIME 'timestamp'

For information about the administrative clauses of the CREATE USER or CREATE ROLE command that are supported by Advanced Server, please refer to the PostgreSQL core documentation available at:

<http://www.postgresql.org/docs/9.5/static/sql-commands.html>

CREATE ROLE|USER… PROFILE adds a new role with an associated profile to an Advanced Server database cluster.

Roles created with the CREATE USER command are (by default) login roles. Roles created with the CREATE ROLE command are (by default) not login roles. To create a login account with the CREATE ROLE command, you must include the LOGIN keyword.

Only a database superuser can use the CREATE USER|ROLE clauses that enforce profile management; these clauses enforce the following behaviors:

Include the PROFILE clause and a *profile\_name* to associate a pre-defined profile with a role, or to change which pre-defined profile is associated with a user.

Include the ACCOUNT clause and the LOCK or UNLOCK keyword to specify that the user account should be placed in a locked or unlocked state.

Include the LOCK TIME 'timestamp' clause and a date/time value to lock the role at the specified time, and unlock the role at the time indicated by the PASSWORD\_LOCK\_TIME parameter of the profile assigned to this role. If LOCK TIME is used with the ACCOUNT LOCK clause, the role can only be unlocked by a database superuser with the ACCOUNT UNLOCK clause.

Include the PASSWORD EXPIRE clause with the optional AT 'timestamp' keywords to specify a date/time when the password associated with the role will expire. If you omit the AT 'timestamp' keywords, the password will expire immediately.

Each login role may only have one profile. To discover the profile that is currently associated with a login role, query the profile column of the DBA\_USERS view.

**Parameters**

name

The name of the role.

profile\_name

The name of the profile associated with the role.

timestamp

The date and time at which the clause will be enforced. When specifying a value for timestamp, enclose the value in single-quotes.

**Examples**

The following example uses CREATE USER to create a login role named john who is associated with the acctg\_profile profile:

CREATE USER john PROFILE acctg\_profile IDENTIFIED BY “1safepwd”;

john can log in to the server, using the password 1safepwd.

The following example uses CREATE ROLE to create a login role named john who is associated with the acctg\_profile profile:

CREATE ROLE john PROFILE acctg\_profile LOGIN PASSWORD “1safepwd”;

john can log in to the server, using the password 1safepwd.

### Backing up Profile Management Functions

A profile may include a PASSWORD\_VERIFY\_FUNCTION clause that refers to a user-defined function that specifies the behavior enforced by Advanced Server. Profiles are global objects; they are shared by all of the databases within a cluster. While profiles are global objects, user-defined functions are database objects.

Invoking pg\_dumpall with the –g or –r option will create a script that recreates the definition of any existing profiles, but that does not recreate the user-defined functions that are referred to by the PASSWORD\_VERIFY\_FUNCTION clause. You should use the pg\_dump utility to explicitly dump (and later restore) the database in which those functions reside.

The script created by pg\_dump will contain a command that includes the clause and function name:

ALTER PROFILE… LIMIT PASSWORD\_VERIFY\_FUNCTION *function\_name*

to associate the restored function with the profile with which it was previously associated.

If the PASSWORD\_VERIFY\_FUNCTION clause is set to DEFAULT or NULL, the behavior will be replicated by the script generated by the pg\_dumpall –g or pg\_dumpall –r command.

# Enhanced SQL Features

Advanced Server includes enhanced SQL functionality that provides additional flexibility and convenience.

This chapter describes these additions.

## Synonyms

A synonym is an identifier that can be used to reference another database object in a SQL statement. A synonym is useful in cases where a database object would normally require full qualification by schema name to be properly referenced in a SQL statement. A synonym defined for that object simplifies the reference to a single, unqualified name.

Advanced Server supports synonyms for:

* Tables
* Views
* Sequences
* Procedures
* Functions
* Types
* Other synonyms

Neither the referenced schema or referenced object must exist at the time that you create the synonym; a synonym may refer to a non-existent object or schema. A synonym will become invalid if you drop the referenced object or schema. You must explicitly drop a synonym to remove it.

As with any other schema object, Advanced Server uses the search path to resolve unqualified synonym names. If you have two synonyms with the same name, an unqualified reference to a synonym will resolve to the first synonym with the given name in the search path. If public is in your search path, you can refer to a synonym in that schema without qualifying that name.

When Advanced Server executes an SQL command, the privileges of the current user are checked against the synonym’s underlying database object; if the user does not have the proper permissions for that object, the SQL command will fail.

Creating a Synonym

Use the CREATE SYNONYM command to create a synonym. The syntax is:

CREATE [OR REPLACE] [PUBLIC] SYNONYM [schema.]syn\_name   
 FOR object\_schema.object\_name;

Parameters:

syn\_name

*syn\_name* is the name of the synonym. A synonym name must be unique within a schema.

schema

*schema* specifies the name of the schema that the synonym resides in. If you do not specify a schema name, the synonym is created in the first existing schema in your search path.

object\_name

*object\_name* specifies the name of the object.

object\_schema

*object\_schema* specifies the name of the schema that the object resides in.

Include the REPLACE clause to replace an existing synonym definition with a new synonym definition.

Include the PUBLIC clause to create the synonym in the public schema. The CREATE PUBLIC SYNONYM command creates a synonym that resides in the public schema:

CREATE [OR REPLACE] PUBLIC SYNONYM syn\_name FOR object\_schema.object\_name;

This just a shorthand way to write:

CREATE [OR REPLACE] SYNONYM public.syn\_name FOR object\_schema.object\_name;

The following example creates a synonym named personnel that refers to the enterprisedb.emp table.

CREATE SYNONYM personnel FOR enterprisedb.emp;

Unless the synonym is schema qualified in the CREATE SYNONYM command, it will be created in the first existing schema in your search path. You can view your search path by executing the following command:

SHOW SEARCH\_PATH;

search\_path

-----------------------

development,accounting

(1 row)

In our example, if a schema named development does not exist, the synonym will be created in the schema named accounting.

Now, the emp table in the enterprisedb schema can be referenced in any SQL statement (DDL or DML), by using the synonym, personnel:

INSERT INTO personnel VALUES (8142,'ANDERSON','CLERK',7902,'17-DEC-06',1300,NULL,20);

SELECT \* FROM personnel;

empno | ename | job | mgr | hiredate | sal | comm | deptno

-------+----------+-----------+------+--------------------+---------+---------+--------

7369 | SMITH | CLERK | 7902 | 17-DEC-80 00:00:00 | 800.00 | | 20

7499 | ALLEN | SALESMAN | 7698 | 20-FEB-81 00:00:00 | 1600.00 | 300.00 | 30

7521 | WARD | SALESMAN | 7698 | 22-FEB-81 00:00:00 | 1250.00 | 500.00 | 30

7566 | JONES | MANAGER | 7839 | 02-APR-81 00:00:00 | 2975.00 | | 20

7654 | MARTIN | SALESMAN | 7698 | 28-SEP-81 00:00:00 | 1250.00 | 1400.00 | 30

7698 | BLAKE | MANAGER | 7839 | 01-MAY-81 00:00:00 | 2850.00 | | 30

7782 | CLARK | MANAGER | 7839 | 09-JUN-81 00:00:00 | 2450.00 | | 10

7788 | SCOTT | ANALYST | 7566 | 19-APR-87 00:00:00 | 3000.00 | | 20

7839 | KING | PRESIDENT | | 17-NOV-81 00:00:00 | 5000.00 | | 10

7844 | TURNER | SALESMAN | 7698 | 08-SEP-81 00:00:00 | 1500.00 | 0.00 | 30

7876 | ADAMS | CLERK | 7788 | 23-MAY-87 00:00:00 | 1100.00 | | 20

7900 | JAMES | CLERK | 7698 | 03-DEC-81 00:00:00 | 950.00 | | 30

7902 | FORD | ANALYST | 7566 | 03-DEC-81 00:00:00 | 3000.00 | | 20

7934 | MILLER | CLERK | 7782 | 23-JAN-82 00:00:00 | 1300.00 | | 10

8142 | ANDERSON | CLERK | 7902 | 17-DEC-06 00:00:00 | 1300.00 | | 20

(15 rows)

Deleting a Synonym

To delete a synonym, use the command, DROP SYNONYM. The syntax is:

DROP [PUBLIC] SYNONYM [schema.] syn\_name

Parameters:

syn\_name

*syn\_name* is the name of the synonym. A synonym name must be unique within a schema.

schema

*schema* specifies the name of the schema in which the synonym resides.

Like any other object that can be schema-qualified, you may have two synonyms with the same name in your search path. To disambiguate the name of the synonym that you are dropping, include a schema name. Unless a synonym is schema qualified in the DROP SYNONYM command, Advanced Server deletes the first instance of the synonym it finds in your search path.

You can optionally include the PUBLIC clause to drop a synonym that resides in the public schema. The DROP PUBLIC SYNONYM command drops a synonym that resides in the public schema:

DROP PUBLIC SYNONYM *syn*\_*name*;

The following example drops the synonym, personnel:

DROP SYNONYM personnel;

## Hierarchical Queries

A hierarchical query is a type of query that returns the rows of the result set in a hierarchical order based upon data forming a parent-child relationship. A hierarchy is typically represented by an inverted tree structure. The tree is comprised of interconnected nodes. Each node may be connected to none, one, or multiple child nodes. Each node is connected to one parent node except for the top node which has no parent. This node is the root node. Each tree has exactly one root node. Nodes that don’t have any children are called leaf nodes. A tree always has at least one leaf node - e.g., the trivial case where the tree is comprised of a single node. In this case it is both the root and the leaf.

In a hierarchical query the rows of the result set represent the nodes of one or more trees.

Note: It is possible that a single, given row may appear in more than one tree and thus appear more than once in the result set.

The hierarchical relationship in a query is described by the CONNECT BY clause which forms the basis of the order in which rows are returned in the result set. The context of where the CONNECT BY clause and its associated optional clauses appear in the SELECT command is shown below.

SELECT select\_list FROM table\_expression [ WHERE ...]

[ START WITH start\_expression ]

CONNECT BY { PRIOR parent\_expr = child\_expr |

child\_expr = PRIOR parent\_expr }

[ ORDER SIBLINGS BY column1 [ ASC | DESC ]

[, column2 [ ASC | DESC ] ] ...

[ GROUP BY ...]

[ HAVING ...]

[ other ...]

select\_list is one or more expressions that comprise the fields of the result set. table\_expression is one or more tables or views from which the rows of the result set originate. other is any additional legal SELECT command clauses. The clauses pertinent to hierarchical queries, START WITH, CONNECT BY, and ORDER SIBLINGS BY are described in the following sections.

Note: At this time, Advanced Server does not support the use of AND (or other operators) in the CONNECT BY clause.

### Defining the Parent/Child Relationship

For any given row, its parent and its children are determined by the CONNECT BY clause. The CONNECT BY clause must consist of two expressions compared with the equals (=) operator. In addition, one of these two expressions must be preceded by the keyword, PRIOR.

For any given row, to determine its children:

* Evaluate parent\_expr on the given row
* Evaluate child\_expr on any other row resulting from the evaluation of table\_expression
* If parent\_expr = child\_expr, then this row is a child node of the given parent row
* Repeat the process for all remaining rows in table\_expression. All rows that satisfy the equation in step 3 are the children nodes of the given parent row.

Note: The evaluation process to determine if a row is a child node occurs on every row returned by table\_expression before the WHERE clause is applied to table\_expression.

By iteratively repeating this process treating each child node found in the prior steps as a parent, an inverted tree of nodes is constructed. The process is complete when the final set of child nodes has no children of their own - these are the leaf nodes.

A SELECT command that includes a CONNECT BY clause typically includes the START WITH clause. The START WITH clause determines the rows that are to be the root nodes - i.e., the rows that are the initial parent nodes upon which the algorithm described previously is to be applied. This is further explained in the following section.

### Selecting the Root Nodes

The START WITH clause is used to determine the row(s) selected by table\_expression that are to be used as the root nodes. All rows selected by table\_expression where start\_expression evaluates to true become a root node of a tree. Thus, the number of potential trees in the result set is equal to the number of root nodes. As a consequence, if the START WITH clause is omitted, then every row returned by table\_expression is a root of its own tree.

### Organization Tree in the Sample Application

Consider the emp table of the sample application. The rows of the emp table form a hierarchy based upon the mgr column which contains the employee number of the employee’s manager. Each employee has at most, one manager. KING is the president of the company so he has no manager, therefore KING’s mgr column is null. Also, it is possible for an employee to act as a manager for more than one employee. This relationship forms a typical, tree-structured, hierarchical organization chart as illustrated below.

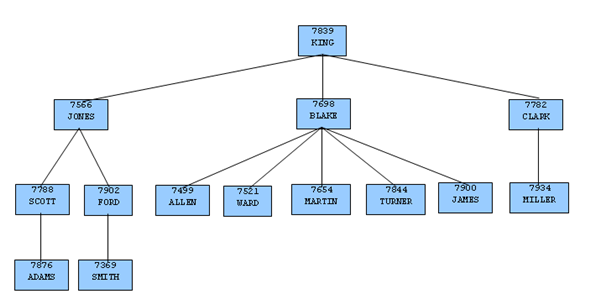


Figure 1 Employee Organization Hierarchy

To form a hierarchical query based upon this relationship, the SELECT command includes the clause, CONNECT BY PRIOR empno = mgr. For example, given the company president, KING, with employee number 7839, any employee whose mgr column is 7839 reports directly to KING which is true for JONES, BLAKE, and CLARK (these are the child nodes of KING). Similarly, for employee, JONES, any other employee with mgr column equal to 7566 is a child node of JONES - these are SCOTT and FORD in this example.

The top of the organization chart is KING so there is one root node in this tree. The START WITH mgr IS NULL clause selects only KING as the initial root node.

The complete SELECT command is shown below.

SELECT ename, empno, mgr

FROM emp

START WITH mgr IS NULL

CONNECT BY PRIOR empno = mgr;

The rows in the query output traverse each branch from the root to leaf moving in a top-to-bottom, left-to-right order. Below is the output from this query.

ename | empno | mgr

--------+-------+------

KING | 7839 |

JONES | 7566 | 7839

SCOTT | 7788 | 7566

ADAMS | 7876 | 7788

FORD | 7902 | 7566

SMITH | 7369 | 7902

BLAKE | 7698 | 7839

ALLEN | 7499 | 7698

WARD | 7521 | 7698

MARTIN | 7654 | 7698

TURNER | 7844 | 7698

JAMES | 7900 | 7698

CLARK | 7782 | 7839

MILLER | 7934 | 7782

(14 rows)

### Node Level

LEVEL is a pseudo-column that can be used wherever a column can appear in the SELECT command. For each row in the result set, LEVEL returns a non-zero integer value designating the depth in the hierarchy of the node represented by this row. The LEVEL for root nodes is 1. The LEVEL for direct children of root nodes is 2, and so on.

The following query is a modification of the previous query with the addition of the LEVEL pseudo-column. In addition, using the LEVEL value, the employee names are indented to further emphasize the depth in the hierarchy of each row.

SELECT LEVEL, LPAD (' ', 2 \* (LEVEL - 1)) || ename "employee", empno, mgr

FROM emp START WITH mgr IS NULL

CONNECT BY PRIOR empno = mgr;

The output from this query follows.

level | employee | empno | mgr

-------+-------------+-------+------

1 | KING | 7839 |

2 | JONES | 7566 | 7839

3 | SCOTT | 7788 | 7566

4 | ADAMS | 7876 | 7788

3 | FORD | 7902 | 7566

4 | SMITH | 7369 | 7902

2 | BLAKE | 7698 | 7839

3 | ALLEN | 7499 | 7698

3 | WARD | 7521 | 7698

3 | MARTIN | 7654 | 7698

3 | TURNER | 7844 | 7698

3 | JAMES | 7900 | 7698

2 | CLARK | 7782 | 7839

3 | MILLER | 7934 | 7782

(14 rows)

Nodes that share a common parent and are at the same level are called siblings. For example in the above output, employees ALLEN, WARD, MARTIN, TURNER, and JAMES are siblings since they are all at level three with parent, BLAKE. JONES, BLAKE, and CLARK are siblings since they are at level two and KING is their common parent.

### Ordering the Siblings

The result set can be ordered so the siblings appear in ascending or descending order by selected column value(s) using the ORDER SIBLINGS BY clause. This is a special case of the ORDER BY clause that can be used only with hierarchical queries.

The previous query is further modified with the addition of ORDER SIBLINGS BY ename ASC.

SELECT LEVEL, LPAD (' ', 2 \* (LEVEL - 1)) || ename "employee", empno, mgr

FROM emp START WITH mgr IS NULL

CONNECT BY PRIOR empno = mgr

ORDER SIBLINGS BY ename ASC;

The output from the prior query is now modified so the siblings appear in ascending order by name. Siblings BLAKE, CLARK, and JONES are now alphabetically arranged under KING. Siblings ALLEN, JAMES, MARTIN, TURNER, and WARD are alphabetically arranged under BLAKE, and so on.

level | employee | empno | mgr

-------+-------------+-------+------

1 | KING | 7839 |

2 | BLAKE | 7698 | 7839

3 | ALLEN | 7499 | 7698

3 | JAMES | 7900 | 7698

3 | MARTIN | 7654 | 7698

3 | TURNER | 7844 | 7698

3 | WARD | 7521 | 7698

2 | CLARK | 7782 | 7839

3 | MILLER | 7934 | 7782

2 | JONES | 7566 | 7839

3 | FORD | 7902 | 7566

4 | SMITH | 7369 | 7902

3 | SCOTT | 7788 | 7566

4 | ADAMS | 7876 | 7788

(14 rows)

This final example adds the WHERE clause and starts with three root nodes. After the node tree is constructed, the WHERE clause filters out rows in the tree to form the result set.

SELECT LEVEL, LPAD (' ', 2 \* (LEVEL - 1)) || ename "employee", empno, mgr

FROM emp WHERE mgr IN (7839, 7782, 7902, 7788)

START WITH ename IN ('BLAKE','CLARK','JONES')

CONNECT BY PRIOR empno = mgr

ORDER SIBLINGS BY ename ASC;

The output from the query shows three root nodes (level one) - BLAKE, CLARK, and JONES. In addition, rows that do not satisfy the WHERE clause have been eliminated from the output.

level | employee | empno | mgr

-------+-----------+-------+------

1 | BLAKE | 7698 | 7839

1 | CLARK | 7782 | 7839

2 | MILLER | 7934 | 7782

1 | JONES | 7566 | 7839

3 | SMITH | 7369 | 7902

3 | ADAMS | 7876 | 7788

(6 rows)

### Retrieving the Root Node with CONNECT\_BY\_ROOT

CONNECT\_BY\_ROOT is a unary operator that can be used to qualify a column in order to return the column’s value of the row considered to be the root node in relation to the current row.

Note: A unary operator operates on a single operand, which in the case of CONNECT\_BY\_ROOT, is the column name following the CONNECT\_BY\_ROOT keyword.

In the context of the SELECT list, the CONNECT\_BY\_ROOT operator is shown by the following.

SELECT [... ,] CONNECT\_BY\_ROOT column [, ...]

FROM table\_expression ...

The following are some points to note about the CONNECT\_BY\_ROOT operator.

* The CONNECT\_BY\_ROOT operator can be used in the SELECT list, the WHERE clause, the GROUP BY clause, the HAVING clause, the ORDER BY clause, and the ORDER SIBLINGS BY clause as long as the SELECT command is for a hierarchical query.
* The CONNECT\_BY\_ROOT operator cannot be used in the CONNECT BY clause or the START WITH clause of the hierarchical query.
* It is possible to apply CONNECT\_BY\_ROOT to an expression involving a column, but to do so, the expression must be enclosed within parentheses.

The following query shows the use of the CONNECT\_BY\_ROOT operator to return the employee number and employee name of the root node for each employee listed in the result set based on trees starting with employees BLAKE, CLARK, and JONES.

SELECT LEVEL, LPAD (' ', 2 \* (LEVEL - 1)) || ename "employee", empno, mgr,

CONNECT\_BY\_ROOT empno "mgr empno",

CONNECT\_BY\_ROOT ename "mgr ename"

FROM emp

START WITH ename IN ('BLAKE','CLARK','JONES')

CONNECT BY PRIOR empno = mgr

ORDER SIBLINGS BY ename ASC;

Note that the output from the query shows that all of the root nodes in columns mgr empno and mgr ename are one of the employees, BLAKE, CLARK, or JONES, listed in the START WITH clause.

level | employee | empno | mgr | mgr empno | mgr ename

-------+-----------+-------+------+-----------+-----------

1 | BLAKE | 7698 | 7839 | 7698 | BLAKE

2 | ALLEN | 7499 | 7698 | 7698 | BLAKE

2 | JAMES | 7900 | 7698 | 7698 | BLAKE

2 | MARTIN | 7654 | 7698 | 7698 | BLAKE

2 | TURNER | 7844 | 7698 | 7698 | BLAKE

2 | WARD | 7521 | 7698 | 7698 | BLAKE

1 | CLARK | 7782 | 7839 | 7782 | CLARK

2 | MILLER | 7934 | 7782 | 7782 | CLARK

1 | JONES | 7566 | 7839 | 7566 | JONES

2 | FORD | 7902 | 7566 | 7566 | JONES

3 | SMITH | 7369 | 7902 | 7566 | JONES

2 | SCOTT | 7788 | 7566 | 7566 | JONES

3 | ADAMS | 7876 | 7788 | 7566 | JONES

(13 rows)

The following is a similar query, but producing only one tree starting with the single, top-level, employee where the mgr column is null.

SELECT LEVEL, LPAD (' ', 2 \* (LEVEL - 1)) || ename "employee", empno, mgr,

CONNECT\_BY\_ROOT empno "mgr empno",

CONNECT\_BY\_ROOT ename "mgr ename"

FROM emp START WITH mgr IS NULL

CONNECT BY PRIOR empno = mgr

ORDER SIBLINGS BY ename ASC;

In the following output, all of the root nodes in columns mgr empno and mgr ename indicate KING as the root for this particular query.

level | employee | empno | mgr | mgr empno | mgr ename

-------+-------------+-------+------+-----------+-----------

1 | KING | 7839 | | 7839 | KING

2 | BLAKE | 7698 | 7839 | 7839 | KING

3 | ALLEN | 7499 | 7698 | 7839 | KING

3 | JAMES | 7900 | 7698 | 7839 | KING

3 | MARTIN | 7654 | 7698 | 7839 | KING

3 | TURNER | 7844 | 7698 | 7839 | KING

3 | WARD | 7521 | 7698 | 7839 | KING

2 | CLARK | 7782 | 7839 | 7839 | KING

3 | MILLER | 7934 | 7782 | 7839 | KING

2 | JONES | 7566 | 7839 | 7839 | KING

3 | FORD | 7902 | 7566 | 7839 | KING

4 | SMITH | 7369 | 7902 | 7839 | KING

3 | SCOTT | 7788 | 7566 | 7839 | KING

4 | ADAMS | 7876 | 7788 | 7839 | KING

(14 rows)

By contrast, the following example omits the START WITH clause thereby resulting in fourteen trees.

SELECT LEVEL, LPAD (' ', 2 \* (LEVEL - 1)) || ename "employee", empno, mgr,

CONNECT\_BY\_ROOT empno "mgr empno",

CONNECT\_BY\_ROOT ename "mgr ename"

FROM emp

CONNECT BY PRIOR empno = mgr

ORDER SIBLINGS BY ename ASC;

The following is the output from the query. Each node appears at least once as a root node under the mgr empno and mgr ename columns since even the leaf nodes form the top of their own trees.

level | employee | empno | mgr | mgr empno | mgr ename

-------+-------------+-------+------+-----------+-----------

1 | ADAMS | 7876 | 7788 | 7876 | ADAMS

1 | ALLEN | 7499 | 7698 | 7499 | ALLEN

1 | BLAKE | 7698 | 7839 | 7698 | BLAKE

2 | ALLEN | 7499 | 7698 | 7698 | BLAKE

2 | JAMES | 7900 | 7698 | 7698 | BLAKE

2 | MARTIN | 7654 | 7698 | 7698 | BLAKE

2 | TURNER | 7844 | 7698 | 7698 | BLAKE

2 | WARD | 7521 | 7698 | 7698 | BLAKE

1 | CLARK | 7782 | 7839 | 7782 | CLARK

2 | MILLER | 7934 | 7782 | 7782 | CLARK

1 | FORD | 7902 | 7566 | 7902 | FORD

2 | SMITH | 7369 | 7902 | 7902 | FORD

1 | JAMES | 7900 | 7698 | 7900 | JAMES

1 | JONES | 7566 | 7839 | 7566 | JONES

2 | FORD | 7902 | 7566 | 7566 | JONES

3 | SMITH | 7369 | 7902 | 7566 | JONES

2 | SCOTT | 7788 | 7566 | 7566 | JONES

3 | ADAMS | 7876 | 7788 | 7566 | JONES

1 | KING | 7839 | | 7839 | KING

2 | BLAKE | 7698 | 7839 | 7839 | KING

3 | ALLEN | 7499 | 7698 | 7839 | KING

3 | JAMES | 7900 | 7698 | 7839 | KING

3 | MARTIN | 7654 | 7698 | 7839 | KING

3 | TURNER | 7844 | 7698 | 7839 | KING

3 | WARD | 7521 | 7698 | 7839 | KING

2 | CLARK | 7782 | 7839 | 7839 | KING

3 | MILLER | 7934 | 7782 | 7839 | KING

2 | JONES | 7566 | 7839 | 7839 | KING

3 | FORD | 7902 | 7566 | 7839 | KING

4 | SMITH | 7369 | 7902 | 7839 | KING

3 | SCOTT | 7788 | 7566 | 7839 | KING

4 | ADAMS | 7876 | 7788 | 7839 | KING

1 | MARTIN | 7654 | 7698 | 7654 | MARTIN

1 | MILLER | 7934 | 7782 | 7934 | MILLER

1 | SCOTT | 7788 | 7566 | 7788 | SCOTT

2 | ADAMS | 7876 | 7788 | 7788 | SCOTT

1 | SMITH | 7369 | 7902 | 7369 | SMITH

1 | TURNER | 7844 | 7698 | 7844 | TURNER

1 | WARD | 7521 | 7698 | 7521 | WARD

(39 rows)

The following illustrates the unary operator effect of CONNECT\_BY\_ROOT. As shown in this example, when applied to an expression that is not enclosed in parentheses, the CONNECT\_BY\_ROOT operator affects only the term, ename, immediately following it. The subsequent concatenation of || ' manages ' || ename is not part of the CONNECT\_BY\_ROOT operation, hence the second occurrence of ename results in the value of the currently processed row while the first occurrence of ename results in the value from the root node.

SELECT LEVEL, LPAD (' ', 2 \* (LEVEL - 1)) || ename "employee", empno, mgr,

CONNECT\_BY\_ROOT ename || ' manages ' || ename "top mgr/employee"

FROM emp

START WITH ename IN ('BLAKE','CLARK','JONES')

CONNECT BY PRIOR empno = mgr

ORDER SIBLINGS BY ename ASC;

The following is the output from the query. Note the values produced under the top mgr/employee column.

level | employee | empno | mgr | top mgr/employee

-------+-----------+-------+------+----------------------

1 | BLAKE | 7698 | 7839 | BLAKE manages BLAKE

2 | ALLEN | 7499 | 7698 | BLAKE manages ALLEN

2 | JAMES | 7900 | 7698 | BLAKE manages JAMES

2 | MARTIN | 7654 | 7698 | BLAKE manages MARTIN

2 | TURNER | 7844 | 7698 | BLAKE manages TURNER

2 | WARD | 7521 | 7698 | BLAKE manages WARD

1 | CLARK | 7782 | 7839 | CLARK manages CLARK

2 | MILLER | 7934 | 7782 | CLARK manages MILLER

1 | JONES | 7566 | 7839 | JONES manages JONES

2 | FORD | 7902 | 7566 | JONES manages FORD

3 | SMITH | 7369 | 7902 | JONES manages SMITH

2 | SCOTT | 7788 | 7566 | JONES manages SCOTT

3 | ADAMS | 7876 | 7788 | JONES manages ADAMS

(13 rows)

The following example uses the CONNECT\_BY\_ROOT operator on an expression enclosed in parentheses.

SELECT LEVEL, LPAD (' ', 2 \* (LEVEL - 1)) || ename "employee", empno, mgr,

CONNECT\_BY\_ROOT ('Manager ' || ename || ' is emp # ' || empno)

"top mgr/empno"

FROM emp

START WITH ename IN ('BLAKE','CLARK','JONES')

CONNECT BY PRIOR empno = mgr

ORDER SIBLINGS BY ename ASC;

The following is the output of the query. Note that the values of both ename and empno are affected by the CONNECT\_BY\_ROOT operator and as a result, return the values from the root node as shown under the top mgr/empno column.

level | employee | empno | mgr | top mgr/empno

-------+-----------+-------+------+-----------------------------

1 | BLAKE | 7698 | 7839 | Manager BLAKE is emp # 7698

2 | ALLEN | 7499 | 7698 | Manager BLAKE is emp # 7698

2 | JAMES | 7900 | 7698 | Manager BLAKE is emp # 7698

2 | MARTIN | 7654 | 7698 | Manager BLAKE is emp # 7698

2 | TURNER | 7844 | 7698 | Manager BLAKE is emp # 7698

2 | WARD | 7521 | 7698 | Manager BLAKE is emp # 7698

1 | CLARK | 7782 | 7839 | Manager CLARK is emp # 7782

2 | MILLER | 7934 | 7782 | Manager CLARK is emp # 7782

1 | JONES | 7566 | 7839 | Manager JONES is emp # 7566

2 | FORD | 7902 | 7566 | Manager JONES is emp # 7566

3 | SMITH | 7369 | 7902 | Manager JONES is emp # 7566

2 | SCOTT | 7788 | 7566 | Manager JONES is emp # 7566

3 | ADAMS | 7876 | 7788 | Manager JONES is emp # 7566

(13 rows)

### Retrieving a Path with SYS\_CONNECT\_BY\_PATH

SYS\_CONNECT\_BY\_PATH is a function that works within a hierarchical query to retrieve the column values of a specified column that occur between the current node and the root node. The signature of the function is:

SYS\_CONNECT\_BY\_PATH (*column*, *delimiter*)

The function takes two arguments:

*column* is the name of a column that resides within a table specified in the hierarchical query that is calling the function.

*delimiter* is the varchar value that separates each entry in the specified column.

The following example returns a list of employee names, and their managers; if the manager has a manager, that name is appended to the result:

edb=# SELECT level, ename , SYS\_CONNECT\_BY\_PATH(ename, '/') managers   
 FROM emp   
 CONNECT BY PRIOR empno = mgr   
 START WITH mgr IS NULL   
 ORDER BY level, ename, managers;

level | ename | managers

-------+--------+-------------------------

1 | KING | /KING

2 | BLAKE | /KING/BLAKE

2 | CLARK | /KING/CLARK

2 | JONES | /KING/JONES

3 | ALLEN | /KING/BLAKE/ALLEN

3 | FORD | /KING/JONES/FORD

3 | JAMES | /KING/BLAKE/JAMES

3 | MARTIN | /KING/BLAKE/MARTIN

3 | MILLER | /KING/CLARK/MILLER

3 | SCOTT | /KING/JONES/SCOTT

3 | TURNER | /KING/BLAKE/TURNER

3 | WARD | /KING/BLAKE/WARD

4 | ADAMS | /KING/JONES/SCOTT/ADAMS

4 | SMITH | /KING/JONES/FORD/SMITH

(14 rows)

Within the result set:

* The level column displays the number of levels that the query returned.
* The ename column displays the employee name.
* The managers column contains the hierarchical list of managers.

The Advanced Server implementation of SYS\_CONNECT\_BY\_PATH does not support use of:

* SYS\_CONNECT\_BY\_PATH inside CONNECT\_BY\_PATH
* SYS\_CONNECT\_BY\_PATH inside SYS\_CONNECT\_BY\_PATH

## Extended Functions and Operators

This section describes the extended functions and operators provided in Advanced Server.

### Logical Operators

The usual logical operators are available: AND, OR, NOT

SQL uses a three-valued Boolean logic where the null value represents "unknown". Observe the following truth tables:

Table 3‑3‑1 AND/OR Truth Table

| a | b | a AND b | a OR b |
| --- | --- | --- | --- |
| True | True | True | True |
| True | False | False | True |
| True | Null | Null | True |
| False | False | False | False |
| False | Null | False | Null |
| Null | Null | Null | Null |

Table 3‑3‑2 NOT Truth Table

| a | NOT a |
| --- | --- |
| True | False |
| False | True |
| Null | Null |

The operators AND and OR are commutative, that is, you can switch the left and right operand without affecting the result.

### Comparison Operators

The usual comparison operators are shown in the following table.

Table 3‑3‑3 Comparison Operators

| Operator | Description |
| --- | --- |
| < | Less than |
| > | Greater than |
| <= | Less than or equal to |
| >= | Greater than or equal to |
| = | Equal |
| <> | Not equal |
| != | Not equal |

Comparison operators are available for all data types where this makes sense. All comparison operators are binary operators that return values of type BOOLEAN; expressions like 1 < 2 < 3 are not valid (because there is no < operator to compare a Boolean value with 3).

In addition to the comparison operators, the special BETWEEN construct is available.

a BETWEEN x AND y

is equivalent to

a >= x AND a <= y

Similarly,

a NOT BETWEEN x AND y

is equivalent to

a < x OR a > y

There is no difference between the two respective forms apart from the CPU cycles required to rewrite the first one into the second one internally.

To check whether a value is or is not null, use the constructs

expression IS NULL

expression IS NOT NULL

Do not write expression = NULL because NULL is not "equal to" NULL. (The null value represents an unknown value, and it is not known whether two unknown values are equal.) This behavior conforms to the SQL standard.

Some applications may expect that expression = NULL returns true if expression evaluates to the null value. It is highly recommended that these applications be modified to comply with the SQL standard.

### Mathematical Functions and Operators

Mathematical operators are provided for many Advanced Server types. For types without common mathematical conventions for all possible permutations (e.g., date/time types) the actual behavior is described in subsequent sections.

The following table shows the available mathematical operators.

Table 3‑3‑4 Mathematical Operators

| Operator | Description | Example | Result |
| --- | --- | --- | --- |
| + | Addition | 2 + 3 | 5 |
| - | Subtraction | 2 – 3 | -1 |
| \* | Multiplication | 2 \* 3 | 6 |
| / | Division (integer division truncates results) | 4 / 2 | 2 |
| \*\* | Exponentiation Operator | 2 \*\* 3 | 8 |

The following table shows the available mathematical functions. Many of these functions are provided in multiple forms with different argument types. Except where noted, any given form of a function returns the same data type as its argument. The functions working with DOUBLE PRECISION data are mostly implemented on top of the host system’s C library; accuracy and behavior in boundary cases may therefore vary depending on the host system.

Table 3‑3‑5 Mathematical Functions

| Function | Return Type | Description | Example | Result |
| --- | --- | --- | --- | --- |
| ABS(x) | Same as x | Absolute value | ABS(-17.4) | 17.4 |
| CEIL(DOUBLE PRECISION or NUMBER) | Same as input | Smallest integer not less than argument | CEIL(-42.8) | -42 |
| EXP(DOUBLE PRECISION or NUMBER) | Same as input | Exponential | EXP(1.0) | 2.7182818284590452 |
| FLOOR(DOUBLE PRECISION or NUMBER) | Same as input | Largest integer not greater than argument | FLOOR(-42.8) | 43 |
| LN(DOUBLE PRECISION or NUMBER) | Same as input | Natural logarithm | LN(2.0) | 0.6931471805599453 |
| LOG(b NUMBER, x NUMBER) | NUMBER | Logarithm to base b | LOG(2.0, 64.0) | 6.0000000000000000 |
| MOD(y, x) | Same as argument types | Remainder of y/x | MOD(9, 4) | 1 |
| NVL(x, y) | Same as argument types; where both arguments are of the same data type | If x is null, then NVL returns y | NVL(9, 0) | 9 |
| POWER(a DOUBLE PRECISION, b DOUBLE PRECISION) | DOUBLE PRECISION | a raised to the power of b | POWER(9.0, 3.0) | 729.0000000000000000 |
| POWER(a NUMBER, b NUMBER) | NUMBER | a raised to the power of b | POWER(9.0, 3.0) | 729.0000000000000000 |
| ROUND(DOUBLE PRECISION or NUMBER) | Same as input | Round to nearest integer | ROUND(42.4) | 42 |
| ROUND(v NUMBER, s INTEGER) | NUMBER | Round to s decimal places | ROUND(42.4382, 2) | 42.44 |
| SIGN(DOUBLE PRECISION or NUMBER) | Same as input | Sign of the argument (-1, 0, +1) | SIGN(-8.4) | -1 |
| SQRT(DOUBLE PRECISION or NUMBER) | Same as input | Square root | SQRT(2.0) | 1.414213562373095 |
| TRUNC(DOUBLE PRECISION or NUMBER) | Same as input | Truncate toward zero | TRUNC(42.8) | 42 |
| TRUNC(v NUMBER, s INTEGER) | NUMBER | Truncate to s decimal places | TRUNC(42.4382, 2) | 42.43 |
| WIDTH\_BUCKET(op NUMBER, b1 NUMBER, b2 NUMBER, count INTEGER) | INTEGER | Return the bucket to which op would be assigned in an equidepth histogram with count buckets, in the range b1 to b2 | WIDTH\_BUCKET(5.35, 0.024, 10.06, 5) | 3 |

The following table shows the available trigonometric functions. All trigonometric functions take arguments and return values of type DOUBLE PRECISION.

Table 3‑3‑6 Trigonometric Functions

| Function | Description |
| --- | --- |
| ACOS(x) | Inverse cosine |
| ASIN(x) | Inverse sine |
| ATAN(x) | Inverse tangent |
| ATAN2(x, y) | Inverse tangent of x/y |
| COS(x) | Cosine |
| SIN(x) | Sine |
| TAN(x) | Tangent |

### String Functions and Operators

This section describes functions and operators for examining and manipulating string values. Strings in this context include values of the types CHAR, VARCHAR2, and CLOB. Unless otherwise noted, all of the functions listed below work on all of these types, but be wary of potential effects of automatic padding when using the CHAR type. Generally, the functions described here also work on data of non-string types by converting that data to a string representation first.

Table 3‑3‑7 SQL String Functions and Operators

| Function | Return Type | Description | Example | Result |
| --- | --- | --- | --- | --- |
| string || string | CLOB | String concatenation | 'Enterprise' || 'DB' | EnterpriseDB |
| CONCAT(string, string) | CLOB | String concatenation | 'a' || 'b' | ab |
| HEXTORAW(varchar2) | RAW | Converts a VARCHAR2 value to a RAW value | HEXTORAW('303132') | '012' |
| RAWTOHEX(raw) | VARCHAR2 | Converts a RAW value to a HEXADECIMAL value | RAWTOHEX('012') | '303132' |
| INSTR(string, set, [ start [, occurrence ] ]) | INTEGER | Finds the location of a set of characters in a string, starting at position start in the string, string, and looking for the first, second, third and so on occurrences of the set. Returns 0 if the set is not found. | INSTR('PETER PIPER PICKED a PECK of PICKLED PEPPERS','PI',1,3) | 30 |
| INSTRB(string, set) | INTEGER | Returns the position of the set within the string. Returns 0 if set is not found. | INSTRB('PETER PIPER PICKED a PECK of PICKLED PEPPERS', 'PICK') | 13 |
| INSTRB(string, set, start) | INTEGER | Returns the position of the set within the string, beginning at start. Returns 0 if set is not found. | INSTRB('PETER PIPER PICKED a PECK of PICKLED PEPPERS','PICK', 14) | 30 |
| INSTRB(string, set, start, occurrence) | INTEGER | Returns the position of the specified occurrence of set within the string, beginning at start. Returns 0 if set is not found. | INSTRB('PETER PIPER PICKED a PECK of PICKLED PEPPERS','PICK', 1, 2) | 30 |
| LOWER(string) | CLOB | Convert string to lower case | LOWER('TOM') | tom |
| SUBSTR(string, start [, count ]) | CLOB | Extract substring starting from start and going for count characters. If count is not specified, the string is clipped from the start till the end. | SUBSTR('This is a test',6,2) | is |
| SUBSTRB(string, start [, count ]) | CLOB | Same as SUBSTR except start and count are in number of bytes. | SUBSTRB('abc',3) (assuming a double-byte character set) | c |
| SUBSTR2(string, start [, count ]) | CLOB | Alias for SUBSTR. | SUBSTR2('This is a test',6,2) | is |
| SUBSTR2(string, start [, count ]) | CLOB | Alias for SUBSTRB. | SUBSTR2('abc',3) (assuming a double-byte character set) | c |
| SUBSTR4(string, start [, count ]) | CLOB | Alias for SUBSTR. | SUBSTR4('This is a test',6,2) | is |
| SUBSTR4(string, start [, count ]) | CLOB | Alias for SUBSTRB. | SUBSTR4('abc',3) (assuming a double-byte character set) | c |
| SUBSTRC(string, start [, count ]) | CLOB | Alias for SUBSTR. | SUBSTRC('This is a test',6,2) | is |
| SUBSTRC(string, start [, count ]) | CLOB | Alias for SUBSTRB. | SUBSTRC('abc',3) (assuming a double-byte character set) | c |
| TRIM([ LEADING | TRAILING | BOTH ] [ characters ] FROM string) | CLOB | Remove the longest string containing only the characters (a space by default) from the start/end/both ends of the string. | TRIM(BOTH 'x' FROM 'xTomxx') | Tom |
| LTRIM(string [, set]) | CLOB | Removes all the characters specified in set from the left of a given string. If set is not specified, a blank space is used as default. | LTRIM('abcdefghi', 'abc') | defghi |
| RTRIM(string [, set]) | CLOB | Removes all the characters specified in set from the right of a given string. If set is not specified, a blank space is used as default. | RTRIM('abcdefghi', 'ghi') | abcdef |
| UPPER(string) | CLOB | Convert string to upper case | UPPER('tom') | TOM |

Additional string manipulation functions are available and are listed in the following table. Some of them are used internally to implement the SQL-standard string functions listed in Table 3‑3‑7.

Table 3‑3‑8 Other String Functions

| Function | Return Type | Description | Example | Result |
| --- | --- | --- | --- | --- |
| ASCII(string) | INTEGER | ASCII code of the first byte of the argument | ASCII('x') | 120 |
| CHR(INTEGER) | CLOB | Character with the given ASCII code | CHR(65) | A |
| DECODE(expr, expr1a, expr1b [, expr2a, expr2b ]... [, default ]) | Same as argument types of expr1b, expr2b,..., default | Finds first match of expr with expr1a, expr2a, etc. When match found, returns corresponding parameter pair, expr1b, expr2b, etc. If no match found, returns default. If no match found and default not specified, returns null. | DECODE(3, 1,'One', 2,'Two', 3,'Three', 'Not found') | Three |
| INITCAP(string) | CLOB | Convert the first letter of each word to uppercase and the rest to lowercase. Words are sequences of alphanumeric characters separated by non-alphanumeric characters. | INITCAP('hi THOMAS') | Hi Thomas |
| LENGTH | INTEGER | Returns the number of characters in a string value. | LENGTH('Côte d''Azur') | 11 |
| LENGTHC | INTEGER | This function is identical in functionality to LENGTH; the function name is supported for compatibility. | LENGTHC('Côte d''Azur') | 11 |
| LENGTH2 | INTEGER | This function is identical in functionality to LENGTH; the function name is supported for compatibility. | LENGTH2('Côte d''Azur') | 11 |
| LENGTH4 | INTEGER | This function is identical in functionality to LENGTH; the function name is supported for compatibility. | LENGTH4('Côte d''Azur') | 11 |
| LENGTHB | INTEGER | Returns the number of bytes required to hold the given value. | LENGTHB('Côte d''Azur') | 12 |
| LPAD(string, length INTEGER [, fill ]) | CLOB | Fill up string to size, length by prepending the characters, fill (a space by default). If string is already longer than length then it is truncated (on the right). | LPAD('hi', 5, 'xy') | xyxhi |
| REPLACE(string, search\_string [, replace\_string ] | CLOB | Replaces one value in a string with another. If you do not specify a value for replace\_string, the search\_string value when found, is removed. | REPLACE( 'GEORGE', 'GE', 'EG') | EGOREG |
| RPAD(string, length INTEGER [, fill ]) | CLOB | Fill up string to size, length by appending the characters, fill (a space by default). If string is already longer than length then it is truncated. | RPAD('hi', 5, 'xy') | hixyx |
| TRANSLATE(string, from, to) | CLOB | Any character in string that matches a character in the from set is replaced by the corresponding character in the to set. | TRANSLATE('12345', '14', 'ax') | a23x5 |

### Pattern Matching String Functions

Advanced Server offers support for the REGEXP\_COUNT, REGEXP\_INSTR and REGEXP\_SUBSTR functions. These functions search a string for a pattern specified by a regular expression, and return information about occurrences of the pattern within the string. The pattern should be a POSIX-style regular expression; for more information about forming a POSIX-style regular expression, please refer to the core documentation at:

<http://www.postgresql.org/docs/9.5/static/functions-matching.html>

#### REGEXP\_COUNT

REGEXP\_COUNT searches a string for a regular expression, and returns a count of the times that the regular expression occurs. The signature is:

INTEGER REGEXP\_COUNT  
(  
 srcstr TEXT,  
 pattern TEXT,  
 position DEFAULT 1  
 modifier DEFAULT NULL  
)

Parameters

srcstr

srcstr specifies the string to search.

pattern

pattern specifies the regular expression for which REGEXP\_COUNT will search.

position

position is an integer value that indicates the position in the source string at which REGEXP\_COUNT will begin searching. The default value is 1.

modifier

modifier specifies values that control the pattern matching behavior. The default value is NULL. For a complete list of the modifiers supported by Advanced Server, please refer to the PostgreSQL core documentation available at:

<http://www.postgresql.org/docs/9.5/static/functions-matching.html>

Example

In the following simple example, REGEXP\_COUNT returns a count of the number of times the letter i is used in the character string 'reinitializing':

edb=# SELECT REGEXP\_COUNT('reinitializing', 'i', 1) FROM DUAL;

regexp\_count

--------------

5

(1 row)

In the first example, the command instructs REGEXP\_COUNT begins counting in the first position; if we modify the command to start the count on the 6th position:

edb=# SELECT REGEXP\_COUNT('reinitializing', 'i', 6) FROM DUAL;

regexp\_count

--------------

3

(1 row)

REGEXP\_COUNT returns 3; the count now excludes any occurrences of the letter i that occur before the 6th position.

#### REGEXP\_INSTR

REGEXP\_INSTR searches a string for a POSIX-style regular expression. This function returns the position within the string where the match was located. The signature is:

INTEGER REGEXP\_INSTR  
(  
 srcstr TEXT,   
 pattern TEXT,   
 position INT DEFAULT 1,  
 occurrence INT DEFAULT 1,  
 returnparam INT DEFAULT 0,  
 modifier TEXT DEFAULT NULL,  
 subexpression INT DEFAULT 0,  
)

Parameters

srcstr

srcstr specifies the string to search.

pattern

pattern specifies the regular expression for which REGEXP\_INSTR will search.

position

position specifies an integer value that indicates the start position in a source string. The default value is 1.

occurrence

occurrence specifies which match is returned if more than one occurrence of the pattern occurs in the string that is searched. The default value is 1.

returnparam

returnparam is an integer value that specifies the location within the string that REGEXP\_INSTR should return. The default value is 0. Specify:

0 to return the location within the string of the first character that matches pattern.

A value greater than 0 to return the position of the first character following the end of the *pattern*.

modifier

modifier specifies values that control the pattern matching behavior. The default value is NULL. For a complete list of the modifiers supported by Advanced Server, please refer to the PostgreSQL core documentation available at:

<http://www.postgresql.org/docs/9.5/static/functions-matching.html>

subexpression

subexpression is an integer value that identifies the portion of the *pattern* that will be returned by REGEXP\_INSTR. The default value of *subexpression* is 0.

If you specify a value for subexpression, you must include one (or more) set of parentheses in the pattern that isolate a portion of the value being searched for. The value specified by subexpression indicates which set of parentheses should be returned; for example, if subexpression is 2, REGEXP\_INSTR will return the position of the second set of parentheses.

Example

In the following simple example, REGEXP\_INSTR searches a string that contains the a phone number for the first occurrence of a pattern that contains three consecutive digits:

edb=# SELECT REGEXP\_INSTR('800-555-1212', '[0-9][0-9][0-9]', 1, 1) FROM DUAL;

regexp\_instr

--------------

1

(1 row)

The command instructs REGEXP\_INSTR to return the position of the first occurrence. If we modify the command to return the start of the second occurrence of three consecutive digits:

edb=# SELECT REGEXP\_INSTR('800-555-1212', '[0-9][0-9][0-9]', 1, 2) FROM DUAL;

regexp\_instr

--------------

5

(1 row)

REGEXP\_INSTR returns 5; the second occurrence of three consecutive digits begins in the fifth position.

#### REGEXP\_SUBSTR

The REGEXP\_SUBSTR function searches a string for a pattern specified by a POSIX compliant regular expression. REGEXP\_SUBSTR returns the string that matches the pattern specified in the call to the function. The signature of the function is:

TEXT REGEXP\_SUBSTR  
(  
 srcstr TEXT,   
 pattern TEXT,   
 position INT DEFAULT 1,   
 occurrence INT DEFAULT 1,  
 modifier TEXT DEFAULT NULL,  
 subexpression INT DEFAULT 0   
)

Parameters

srcstr

srcstr specifies the string to search.

pattern

pattern specifies the regular expression for which REGEXP\_SUBSTR will search.

position

position specifies an integer value that indicates the start position in a source string. The default value is 1.

occurrence

occurrence specifies which match is returned if more than one occurrence of the pattern occurs in the string that is searched. The default value is 1.

modifier

modifier specifies values that control the pattern matching behavior. The default value is NULL. For a complete list of the modifiers supported by Advanced Server, refer to the PostgreSQL core documentation available at:

<http://www.postgresql.org/docs/9.5/static/functions-matching.html>

subexpression

subexpression is an integer value that identifies the portion of the pattern that will be returned by REGEXP\_SUBSTR. The default value of subexpression is 0.

If you specify a value for subexpression, you must include one (or more) set of parentheses in the pattern that isolate a portion of the value being searched for. The value specified by subexpression indicates which set of parentheses should be returned; for example, if subexpression is 2, REGEXP\_SUBSTR will return the value contained within the second set of parentheses.

Example

In the following simple example, REGEXP\_SUBSTR searches a string that contains a phone number for the first set of three consecutive digits:

edb=# SELECT REGEXP\_SUBSTR('800-555-1212', '[0-9][0-9][0-9]', 1, 1) FROM DUAL;

regexp\_substr

---------------

800

(1 row)

It locates the first occurrence of three digits and returns the string (800); if we modify the command to check for the second occurrence of three consecutive digits:

edb=# SELECT REGEXP\_SUBSTR('800-555-1212', '[0-9][0-9][0-9]', 1, 2) FROM DUAL;

regexp\_substr

---------------

555

(1 row)

REGEXP\_SUBSTR returns 555, the contents of the second substring.

### Pattern Matching Using the LIKE Operator

Advanced Server provides pattern matching using the traditional SQL LIKE operator. The syntax for the LIKE operator is as follows.

string LIKE pattern [ ESCAPE escape-character ]

string NOT LIKE pattern [ ESCAPE escape-character ]

Every pattern defines a set of strings. The LIKE expression returns TRUE if string is contained in the set of strings represented by pattern. As expected, the NOT LIKE expression returns FALSE if LIKE returns TRUE, and vice versa. An equivalent expression is NOT (string LIKE pattern).

If pattern does not contain percent signs or underscore, then the pattern only represents the string itself; in that case LIKE acts like the equals operator. An underscore (\_) in pattern stands for (matches) any single character; a percent sign (%) matches any string of zero or more characters.

Some examples:

'abc' LIKE 'abc' true

'abc' LIKE 'a%' true

'abc' LIKE '\_b\_' true

'abc' LIKE 'c' false

LIKE pattern matches always cover the entire string. To match a pattern anywhere within a string, the pattern must therefore start and end with a percent sign.

To match a literal underscore or percent sign without matching other characters, the respective character in pattern must be preceded by the escape character. The default escape character is the backslash but a different one may be selected by using the ESCAPE clause. To match the escape character itself, write two escape characters.

Note that the backslash already has a special meaning in string literals, so to write a pattern constant that contains a backslash you must write two backslashes in an SQL statement. Thus, writing a pattern that actually matches a literal backslash means writing four backslashes in the statement. You can avoid this by selecting a different escape character with ESCAPE; then a backslash is not special to LIKE anymore. (But it is still special to the string literal parser, so you still need two of them.)

It’s also possible to select no escape character by writing ESCAPE ''. This effectively disables the escape mechanism, which makes it impossible to turn off the special meaning of underscore and percent signs in the pattern.

### Data Type Formatting Functions

The Advanced Server formatting functions (described in Table 3‑3‑9) provide a powerful set of tools for converting various data types (date/time, integer, floating point, numeric) to formatted strings and for converting from formatted strings to specific data types. These functions all follow a common calling convention: the first argument is the value to be formatted and the second argument is a string template that defines the output or input format.

Table 3‑3‑9 Formatting Functions

| Function | Return Type | Description | Example | Result |
| --- | --- | --- | --- | --- |
| TO\_CHAR(DATE [, format ]) | VARCHAR2 | Convert a date/time to a string with output, format. If omitted default format is DD-MON-YY. | TO\_CHAR(SYSDATE, 'MM/DD/YYYY HH12:MI:SS AM') | 07/25/2007 09:43:02 AM |
| TO\_CHAR(TIMESTAMP [, format ]) | VARCHAR2 | Convert a timestamp to a string with output, format. If omitted default format is DD-MON-YY. | TO\_CHAR(CURRENT\_TIMESTAMP, 'MM/DD/YYYY HH12:MI:SS AM') | 08/13/2014 08:55:22 PM |
| TO\_CHAR(INTEGER [, format ]) | VARCHAR2 | Convert an integer to a string with output, format | TO\_CHAR(2412, '999,999S') | 2,412+ |
| TO\_CHAR(NUMBER [, format ]) | VARCHAR2 | Convert a decimal number to a string with output, format | TO\_CHAR(10125.35, '999,999.99') | 10,125.35 |
| TO\_CHAR(DOUBLE PRECISION, format) | VARCHAR2 | Convert a floating-point number to a string with output, format | TO\_CHAR(CAST(123.5282 AS REAL), '999.99') | 123.53 |
| TO\_DATE(string [, format ]) | DATE | Convert a date formatted string to a DATE data type | TO\_DATE('2007-07-04 13:39:10', 'YYYY-MM-DD HH24:MI:SS') | 04-JUL-07 13:39:10 |
| TO\_NUMBER(string [, format ]) | NUMBER | Convert a number formatted string to a NUMBER data type | TO\_NUMBER('2,412-', '999,999S') | -2412 |
| TO\_TIMESTAMP(string, format) | TIMESTAMP | Convert a timestamp formatted string to a TIMESTAMP data type | TO\_TIMESTAMP('05 Dec 2000 08:30:25 pm', 'DD Mon YYYY hh12:mi:ss pm') | 05-DEC-00 20:30:25 |

In an output template string (for TO\_CHAR), there are certain patterns that are recognized and replaced with appropriately-formatted data from the value to be formatted. Any text that is not a template pattern is simply copied verbatim. Similarly, in an input template string (for anything but TO\_CHAR), template patterns identify the parts of the input data string to be looked at and the values to be found there.

The following table shows the template patterns available for formatting date values using the TO\_CHAR and TO\_DATE functions.

Table 3‑3‑10 Template Date/Time Format Patterns

| Pattern | Description |
| --- | --- |
| HH | Hour of day (01-12) |
| HH12 | Hour of day (01-12) |
| HH24 | Hour of day (00-23) |
| MI | Minute (00-59) |
| SS | Second (00-59) |
| SSSSS | Seconds past midnight (0-86399) |
| FFn | Fractional seconds where n is an optional integer from 1 to 9 for the number of digits to return. If omitted, the default is 6. |
| AM or A.M. or PM or P.M. | Meridian indicator (uppercase) |
| am or a.m. or pm or p.m. | Meridian indicator (lowercase) |
| Y,YYY | Year (4 and more digits) with comma |
| YEAR | Year (spelled out) |
| SYEAR | Year (spelled out) (BC dates prefixed by a minus sign) |
| YYYY | Year (4 and more digits) |
| SYYYY | Year (4 and more digits) (BC dates prefixed by a minus sign) |
| YYY | Last 3 digits of year |
| YY | Last 2 digits of year |
| Y | Last digit of year |
| IYYY | ISO year (4 and more digits) |
| IYY | Last 3 digits of ISO year |
| IY | Last 2 digits of ISO year |
| I | Last 1 digit of ISO year |
| BC or B.C. or AD or A.D. | Era indicator (uppercase) |
| bc or b.c. or ad or a.d. | Era indicator (lowercase) |
| MONTH | Full uppercase month name |
| Month | Full mixed-case month name |
| month | Full lowercase month name |
| MON | Abbreviated uppercase month name (3 chars in English, localized lengths vary) |
| Mon | Abbreviated mixed-case month name (3 chars in English, localized lengths vary) |
| mon | Abbreviated lowercase month name (3 chars in English, localized lengths vary) |
| MM | Month number (01-12) |
| DAY | Full uppercase day name |
| Day | Full mixed-case day name |
| day | Full lowercase day name |
| DY | Abbreviated uppercase day name (3 chars in English, localized lengths vary) |
| Dy | Abbreviated mixed-case day name (3 chars in English, localized lengths vary) |
| dy | Abbreviated lowercase day name (3 chars in English, localized lengths vary) |
| DDD | Day of year (001-366) |
| DD | Day of month (01-31) |
| D | Day of week (1-7; Sunday is 1) |
| W | Week of month (1-5) (The first week starts on the first day of the month) |
| WW | Week number of year (1-53) (The first week starts on the first day of the year) |
| IW | ISO week number of year; the first Thursday of the new year is in week 1 |
| CC | Century (2 digits); the 21st century starts on 2001-01-01 |
| SCC | Same as CC except BC dates are prefixed by a minus sign |
| J | Julian Day (days since January 1, 4712 BC) |
| Q | Quarter |
| RM | Month in Roman numerals (I-XII; I=January) (uppercase) |
| rm | Month in Roman numerals (i-xii; i=January) (lowercase) |
| RR | First 2 digits of the year when given only the last 2 digits of the year. Result is based upon an algorithm using the current year and the given 2-digit year. The first 2 digits of the given 2-digit year will be the same as the first 2 digits of the current year with the following exceptions:  If the given 2-digit year is < 50 and the last 2 digits of the current year is >= 50, then the first 2 digits for the given year is 1 greater than the first 2 digits of the current year.  If the given 2-digit year is >= 50 and the last 2 digits of the current year is < 50, then the first 2 digits for the given year is 1 less than the first 2 digits of the current year. |
| RRRR | Only affects TO\_DATE function. Allows specification of 2-digit or 4-digit year. If 2-digit year given, then returns first 2 digits of year like RR format. If 4-digit year given, returns the given 4-digit year. |

Certain modifiers may be applied to any template pattern to alter its behavior. For example, FMMonth is the Month pattern with the FM modifier. The following table shows the modifier patterns for date/time formatting.

Table 3‑3‑11 Template Pattern Modifiers for Date/Time Formatting

| Modifier | Description | Example |
| --- | --- | --- |
| FM prefix | Fill mode (suppress padding blanks and zeros) | FMMonth |
| TH suffix | Uppercase ordinal number suffix | DDTH |
| th suffix | Lowercase ordinal number suffix | DDth |
| FX prefix | Fixed format global option (see usage notes) | FX Month DD Day |
| SP suffix | Spell mode | DDSP |

Usage notes for date/time formatting:

* FM suppresses leading zeroes and trailing blanks that would otherwise be added to make the output of a pattern fixed-width.
* TO\_TIMESTAMP and TO\_DATE skip multiple blank spaces in the input string if the FX option is not used. FX must be specified as the first item in the template. For example TO\_TIMESTAMP('2000    JUN', 'YYYY MON') is correct, but TO\_TIMESTAMP('2000    JUN', 'FXYYYY MON') returns an error, because TO\_TIMESTAMP expects one space only.
* Ordinary text is allowed in TO\_CHAR templates and will be output literally.
* In conversions from string to timestamp or date, the CC field is ignored if there is a YYY, YYYY or Y,YYY field. If CC is used with YY or Y then the year is computed as (CC-1)\*100+YY.

The following table shows the template patterns available for formatting numeric values.

Table 3‑3‑12 Template Patterns for Numeric Formatting

| Pattern | Description |
| --- | --- |
| 9 | Value with the specified number of digits |
| 0 | Value with leading zeroes |
| . (period) | Decimal point |
| , (comma) | Group (thousand) separator |
| $ | Dollar sign |
| PR | Negative value in angle brackets |
| S | Sign anchored to number (uses locale) |
| L | Currency symbol (uses locale) |
| D | Decimal point (uses locale) |
| G | Group separator (uses locale) |
| MI | Minus sign specified in right-most position (if number < 0) |
| RN or rn | Roman numeral (input between 1 and 3999) |
| V | Shift specified number of digits (see notes) |

Usage notes for numeric formatting:

* 9 results in a value with the same number of digits as there are 9s. If a digit is not available it outputs a space.
* TH does not convert values less than zero and does not convert fractional numbers.

V effectively multiplies the input values by 10n, where n is the number of digits following V. TO\_CHAR does not support the use of V combined with a decimal point. (E.g., 99.9V99 is not allowed.)

The following table shows some examples of the use of the TO\_CHAR and TO\_DATE functions.

Table 3‑3‑13 TO\_CHAR Examples

| Expression | Result |
| --- | --- |
| TO\_CHAR(CURRENT\_TIMESTAMP, 'Day, DD  HH12:MI:SS') | 'Tuesday  , 06  05:39:18' |
| TO\_CHAR(CURRENT\_TIMESTAMP, 'FMDay, FMDD  HH12:MI:SS') | 'Tuesday, 6  05:39:18' |
| TO\_CHAR(-0.1, '99.99') | '  -.10' |
| TO\_CHAR(-0.1, 'FM9.99') | '-.1' |
| TO\_CHAR(0.1, '0.9') | ' 0.1' |
| TO\_CHAR(12, '9990999.9') | '    0012.0' |
| TO\_CHAR(12, 'FM9990999.9') | '0012.' |
| TO\_CHAR(485, '999') | ' 485' |
| TO\_CHAR(-485, '999') | '-485' |
| TO\_CHAR(1485, '9,999') | ' 1,485' |
| TO\_CHAR(1485, '9G999') | ' 1,485' |
| TO\_CHAR(148.5, '999.999') | ' 148.500' |
| TO\_CHAR(148.5, 'FM999.999') | '148.5' |
| TO\_CHAR(148.5, 'FM999.990') | '148.500' |
| TO\_CHAR(148.5, '999D999') | ' 148.500' |
| TO\_CHAR(3148.5, '9G999D999') | ' 3,148.500' |
| TO\_CHAR(-485, '999S') | '485-' |
| TO\_CHAR(-485, '999MI') | '485-' |
| TO\_CHAR(485, '999MI') | '485 ' |
| TO\_CHAR(485, 'FM999MI') | '485' |
| TO\_CHAR(-485, '999PR') | '<485>' |
| TO\_CHAR(485, 'L999') | '$ 485' |
| TO\_CHAR(485, 'RN') | '        CDLXXXV' |
| TO\_CHAR(485, 'FMRN') | 'CDLXXXV' |
| TO\_CHAR(5.2, 'FMRN') | 'V' |
| TO\_CHAR(12, '99V999') | ' 12000' |
| TO\_CHAR(12.4, '99V999') | ' 12400' |
| TO\_CHAR(12.45, '99V9') | ' 125' |

#### IMMUTABLE TO\_CHAR(TIMESTAMP, format) Function

There are certain cases of the TO\_CHAR function that can result in usage of an IMMUTABLE form of the function. Basically, a function is IMMUTABLE if the function does not modify the database, and the function returns the same, consistent value dependent upon only its input parameters. That is, the settings of configuration parameters, the locale, the content of the database, etc. do not affect the results returned by the function.

For more information about function volatility categories VOLATILE, STABLE, and IMMUTABLE, please refer to the PostgreSQL core documentation available at:

<http://www.postgresql.org/docs/9.5/static/xfunc-volatility.html>

A particular advantage of an IMMUTABLE function is that it can be used in the CREATE INDEX command to create an index based on that function.

In order for the TO\_CHAR function to use the IMMUTABLE form the following conditions must be satisfied:

* The first parameter of the TO\_CHAR function must be of data type TIMESTAMP.
* The format specified in the second parameter of the TO\_CHAR function must not affect the return value of the function based on factors such as language, locale, etc. For example a format of 'YYYY-MM-DD HH24:MI:SS' can be used for an IMMUTABLE form of the function since, regardless of locale settings, the result of the function is the date and time expressed solely in numeric form. However, a format of 'DD-MON-YYYY' cannot be used for an IMMUTABLE form of the function because the 3-character abbreviation of the month may return different results depending upon the locale setting.

Format patterns that result in a non-immutable function include any variations of spelled out or abbreviated months (MONTH, MON), days (DAY, DY), median indicators (AM, PM), or era indicators (BC, AD).

Note: The condition specified in the second bullet point applies only for an installation of Advanced Server compatible with Oracle databases. For a PostgreSQL compatible installation of Advanced Server, the TO\_CHAR(TIMESTAMP, format) function is locale independent and thus categorized as IMMUTABLE unless the format forces locale dependence with the TM (translation mode) prefix. For example, 'DD-MON-YYYY' would allow the function to be IMMUTABLE, but 'DD-TMMON-YYYY' would not.

For more information about the PostgreSQL TM date/time formatting prefix, please refer to the PostgreSQL core documentation available at:

<http://www.postgresql.org/docs/9.5/static/functions-formatting.html>

For the following example, a table with a TIMESTAMP column is created.

CREATE TABLE ts\_tbl (ts\_col TIMESTAMP);

The following shows the successful creation of an index with the IMMUTABLE form of the TO\_CHAR function. This applies to both an installation compatible with Oracle databases as well as a PostgreSQL compatible installation.

edb=# CREATE INDEX ts\_idx ON ts\_tbl (TO\_CHAR(ts\_col,'YYYY-MM-DD HH24:MI:SS'));

CREATE INDEX

edb=# \dS ts\_idx

Index "public.ts\_idx"

Column | Type | Definition

---------+-------------------+-------------------------------------------------------------

to\_char | character varying | to\_char(ts\_col, 'YYYY-MM-DD HH24:MI:SS'::character varying)

btree, for table "public.ts\_tbl"

In an installation compatible with Oracle databases, the following results in an error because the format specified in the TO\_CHAR function prevents the use of the IMMUTABLE form since the 3-character month abbreviation, MON, may result in different return values based on the locale setting.

edb=# CREATE INDEX ts\_idx\_2 ON ts\_tbl (TO\_CHAR(ts\_col, 'DD-MON-YYYY'));

ERROR: functions in index expression must be marked IMMUTABLE

However, for a PostgreSQL compatible installation, the CREATE INDEX command rejected in the preceding example would be accepted because the function is IMMUTABLE since there is no TM prefix in the format.

postgres=# CREATE INDEX ts\_idx\_2 ON ts\_tbl (TO\_CHAR(ts\_col, 'DD-MON-YYYY'));

CREATE INDEX

postgres=# \d ts\_idx\_2

Index "postgres.ts\_idx\_2"

Column | Type | Definition

---------+------+--------------------------------------

to\_char | text | to\_char(ts\_col, 'DD-MON-YYYY'::text)

btree, for table "postgres.ts\_tbl"

But when the TM prefix is included, the function is not IMMUTABLE, and thus, the index is rejected.

postgres=# CREATE INDEX ts\_idx\_3 ON ts\_tbl (TO\_CHAR(ts\_col, 'DD-TMMON-YYYY'));

ERROR: functions in index expression must be marked IMMUTABLE

### Date/Time Functions and Operators

Table 3‑3‑15 shows the available functions for date/time value processing, with details appearing in the following subsections. Table 3‑3‑14 illustrates the behaviors of the basic arithmetic operators (+, -).

Table 3‑3‑14 Date/Time Operators

| Operator | Example | Result |
| --- | --- | --- |
| + | DATE '2001-09-28' + 7 | 05-OCT-01 00:00:00 |
| + | TIMESTAMP '2001-09-28 13:30:00' + 3 | 01-OCT-01 13:30:00 |
| - | DATE '2001-10-01' - 7 | 24-SEP-01 00:00:00 |
| - | TIMESTAMP '2001-09-28 13:30:00' - 3 | 25-SEP-01 13:30:00 |
| - | TIMESTAMP '2001-09-29 03:00:00' - TIMESTAMP '2001-09-27 12:00:00' | @ 1 day 15 hours |

In the date/time functions of Table 3‑3‑15 the use of the DATE and TIMESTAMP data types are interchangeable.

Table 3‑3‑15 Date/Time Functions

| Function | Return Type | Description | Example | Result |
| --- | --- | --- | --- | --- |
| ADD\_MONTHS(DATE, NUMBER) | DATE | Add months to a date; see Section 3.3.8.1. | ADD\_MONTHS('28-FEB-97', 3.8) | 31-MAY-97 00:00:00 |
| CURRENT\_DATE | DATE | Current date; see Section 3.3.8.8. | CURRENT\_DATE | 04-JUL-07 |
| CURRENT\_TIMESTAMP | TIMESTAMP | Returns the current date and time; see Section 3.3.8.8. | CURRENT\_TIMESTAMP | 04-JUL-07 15:33:23.484 |
| EXTRACT(field FROM TIMESTAMP) | DOUBLE PRECISION | Get subfield; see Section 3.3.8.2. | EXTRACT(hour FROM TIMESTAMP '2001-02-16 20:38:40') | 20 |
| LAST\_DAY(DATE) | DATE | Returns the last day of the month represented by the given date. If the given date contains a time portion, it is carried forward to the result unchanged. | LAST\_DAY('14-APR-98') | 30-APR-98 00:00:00 |
| LOCALTIMESTAMP [ (precision) ] | TIMESTAMP | Current date and time (start of current transaction); see Section 3.3.8.8. | LOCALTIMESTAMP | 04-JUL-07 15:33:23.484 |
| MONTHS\_BETWEEN(DATE, DATE) | NUMBER | Number of months between two dates; see Section 3.3.8.3. | MONTHS\_BETWEEN('28-FEB-07', '30-NOV-06') | 3 |
| NEXT\_DAY(DATE, dayofweek) | DATE | Date falling on dayofweek following specified date; see Section 3.3.8.4. | NEXT\_DAY('16-APR-07','FRI') | 20-APR-07 00:00:00 |
| NEW\_TIME(DATE, VARCHAR, VARCHAR) | DATE | Converts a date and time to an alternate time zone | NEW\_TIME(TO\_DATE '2005/05/29 01:45', 'AST', 'PST') | 2005/05/29  21:45:00 |
| NUMTODSINTERVAL(NUMBER, INTERVAL) | INTERVAL | Converts a number to a specified day or second interval; see Section 3.3.8.9. | SELECT numtodsinterval(100, ‘hour’); | 4 days 04:00:00 |
| NUMTOYMINTERVAL(NUMBER, INTERVAL) | INTERVAL | Converts a number to a specified year or month interval; see Section 3.3.8.10. | SELECT numtoyminterval(100, ‘month’); | 8 years 4 mons |
| ROUND(DATE [, format ]) | DATE | Date rounded according to format; see Section 3.3.8.6. | ROUND(TO\_DATE('29-MAY-05'),'MON') | 01-JUN-05 00:00:00 |
| SYS\_EXTRACT\_UTC(TIMESTAMP WITH TIME ZONE) | TIMESTAMP | Returns Coordinated Universal Time | SYS\_EXTRACT\_UTC(CAST('24-MAR-11 12:30:00PM -04:00' AS TIMESTAMP WITH TIME ZONE)) | 24-MAR-11 16:30:00 |
| SYSDATE | DATE | Returns current date and time | SYSDATE | 01-AUG-12 11:12:34 |
| SYSTIMESTAMP() | TIMESTAMP | Returns current date and time | SYSTIMESTAMP | 01-AUG-12 11:11:23.665229 -07:00 |
| TRUNC(DATE [format]) | DATE | Truncate according to format; see Section 3.3.8.7. | TRUNC(TO\_DATE('29-MAY-05'), 'MON') | 01-MAY-05 |
| 00:00:00 |

#### ADD\_MONTHS

The ADD\_MONTHS functions adds (or subtracts if the second parameter is negative) the specified number of months to the given date. The resulting day of the month is the same as the day of the month of the given date except when the day is the last day of the month in which case the resulting date always falls on the last day of the month.

Any fractional portion of the number of months parameter is truncated before performing the calculation.

If the given date contains a time portion, it is carried forward to the result unchanged.

The following are examples of the ADD\_MONTHS function.

SELECT ADD\_MONTHS('13-JUN-07',4) FROM DUAL;

add\_months

--------------------

13-OCT-07 00:00:00

(1 row)

SELECT ADD\_MONTHS('31-DEC-06',2) FROM DUAL;

add\_months

--------------------

28-FEB-07 00:00:00

(1 row)

SELECT ADD\_MONTHS('31-MAY-04',-3) FROM DUAL;

add\_months

--------------------

29-FEB-04 00:00:00

(1 row)

#### EXTRACT

The EXTRACT function retrieves subfields such as year or hour from date/time values. The EXTRACT function returns values of type DOUBLE PRECISION. The following are valid field names:

YEAR

The year field

SELECT EXTRACT(YEAR FROM TIMESTAMP '2001-02-16 20:38:40') FROM DUAL;

date\_part

-----------

2001

(1 row)

MONTH

The number of the month within the year (1 - 12)

SELECT EXTRACT(MONTH FROM TIMESTAMP '2001-02-16 20:38:40') FROM DUAL;

date\_part

-----------

2

(1 row)

DAY

The day (of the month) field (1 - 31)

SELECT EXTRACT(DAY FROM TIMESTAMP '2001-02-16 20:38:40') FROM DUAL;

date\_part

-----------

16

(1 row)

HOUR

The hour field (0 - 23)

SELECT EXTRACT(HOUR FROM TIMESTAMP '2001-02-16 20:38:40') FROM DUAL;

date\_part

-----------

20

(1 row)

MINUTE

The minutes field (0 - 59)

SELECT EXTRACT(MINUTE FROM TIMESTAMP '2001-02-16 20:38:40') FROM DUAL;

date\_part

-----------

38

(1 row)

SECOND

The seconds field, including fractional parts (0 - 59)

SELECT EXTRACT(SECOND FROM TIMESTAMP '2001-02-16 20:38:40') FROM DUAL;

date\_part

-----------

40

(1 row)

#### MONTHS\_BETWEEN

The MONTHS\_BETWEEN function returns the number of months between two dates. The result is a numeric value which is positive if the first date is greater than the second date or negative if the first date is less than the second date.

The result is always a whole number of months if the day of the month of both date parameters is the same, or both date parameters fall on the last day of their respective months.

The following are some examples of the MONTHS\_BETWEEN function.

SELECT MONTHS\_BETWEEN('15-DEC-06','15-OCT-06') FROM DUAL;

months\_between

----------------

2

(1 row)

SELECT MONTHS\_BETWEEN('15-OCT-06','15-DEC-06') FROM DUAL;

months\_between

----------------

-2

(1 row)

SELECT MONTHS\_BETWEEN('31-JUL-00','01-JUL-00') FROM DUAL;

months\_between

----------------

0.967741935

(1 row)

SELECT MONTHS\_BETWEEN('01-JAN-07','01-JAN-06') FROM DUAL;

months\_between

----------------

12

(1 row)

#### NEXT\_DAY

The NEXT\_DAY function returns the first occurrence of the given weekday strictly greater than the given date. At least the first three letters of the weekday must be specified - e.g., SAT. If the given date contains a time portion, it is carried forward to the result unchanged.

The following are examples of the NEXT\_DAY function.

SELECT NEXT\_DAY(TO\_DATE('13-AUG-07','DD-MON-YY'),'SUNDAY') FROM DUAL;

next\_day

--------------------

19-AUG-07 00:00:00

(1 row)

SELECT NEXT\_DAY(TO\_DATE('13-AUG-07','DD-MON-YY'),'MON') FROM DUAL;

next\_day

--------------------

20-AUG-07 00:00:00

(1 row)

#### NEW\_TIME

The NEW\_TIME function converts a date and time from one time zone to another. NEW\_TIME returns a value of type DATE. The syntax is:

NEW\_TIME(DATE, time\_zone1, time\_zone2)

time\_zone1 and time\_zone2 must be string values from the Time Zone column of the following table:

| Time Zone | Offset from UTC | Description |
| --- | --- | --- |
| AST | UTC+4 | Atlantic Standard Time |
| ADT | UTC+3 | Atlantic Daylight Time |
| BST | UTC+11 | Bering Standard Time |
| BDT | UTC+10 | Bering Daylight Time |
| CST | UTC+6 | Central Standard Time |
| CDT | UTC+5 | Central Daylight Time |
| EST | UTC+5 | Eastern Standard Time |
| EDT | UTC+4 | Eastern Daylight Time |
| GMT | UTC | Greenwich Mean Time |
| HST | UTC+10 | Alaska-Hawaii Standard Time |
| HDT | UTC+9 | Alaska-Hawaii Daylight Time |
| MST | UTC+7 | Mountain Standard Time |
| MDT | UTC+6 | Mountain Daylight Time |
| NST | UTC+3:30 | Newfoundland Standard Time |
| PST | UTC+8 | Pacific Standard Time |
| PDT | UTC+7 | Pacific Daylight Time |
| YST | UTC+9 | Yukon Standard Time |
| YDT | UTC+8 | Yukon Daylight Time |

Following is an example of the NEW\_TIME function.

SELECT NEW\_TIME(TO\_DATE('08-13-07 10:35:15','MM-DD-YY HH24:MI:SS'),'AST', 'PST') "Pacific Standard Time" FROM DUAL;

Pacific Standard Time

---------------------

13-AUG-07 06:35:15

(1 row)

#### ROUND

The ROUND function returns a date rounded according to a specified template pattern. If the template pattern is omitted, the date is rounded to the nearest day. The following table shows the template patterns for the ROUND function.

Table 3‑3‑16 Template Date Patterns for the ROUND Function

| Pattern | Description |
| --- | --- |
| CC, SCC | Returns January 1, cc01 where cc is first 2 digits of the given year if last 2 digits <= 50, or 1 greater than the first 2 digits of the given year if last 2 digits > 50; (for AD years) |
| SYYY, YYYY, YEAR, SYEAR, YYY, YY, Y | Returns January 1, yyyy where yyyy is rounded to the nearest year; rounds down on June 30, rounds up on July 1 |
| IYYY, IYY, IY, I | Rounds to the beginning of the ISO year which is determined by rounding down if the month and day is on or before June 30th, or by rounding up if the month and day is July 1st or later |
| Q | Returns the first day of the quarter determined by rounding down if the month and day is on or before the 15th of the second month of the quarter, or by rounding up if the month and day is on the 16th of the second month or later of the quarter |
| MONTH, MON, MM, RM | Returns the first day of the specified month if the day of the month is on or prior to the 15th; returns the first day of the following month if the day of the month is on the 16th or later |
| WW | Round to the nearest date that corresponds to the same day of the week as the first day of the year |
| IW | Round to the nearest date that corresponds to the same day of the week as the first day of the ISO year |
| W | Round to the nearest date that corresponds to the same day of the week as the first day of the month |
| DDD, DD, J | Rounds to the start of the nearest day; 11:59:59 AM or earlier rounds to the start of the same day; 12:00:00 PM or later rounds to the start of the next day |
| DAY, DY, D | Rounds to the nearest Sunday |
| HH, HH12, HH24 | Round to the nearest hour |
| MI | Round to the nearest minute |

Following are examples of usage of the ROUND function.

The following examples round to the nearest hundred years.

SELECT TO\_CHAR(ROUND(TO\_DATE('1950','YYYY'),'CC'),'DD-MON-YYYY') "Century" FROM DUAL;

Century

-------------

01-JAN-1901

(1 row)

SELECT TO\_CHAR(ROUND(TO\_DATE('1951','YYYY'),'CC'),'DD-MON-YYYY') "Century" FROM DUAL;

Century

-------------

01-JAN-2001

(1 row)

The following examples round to the nearest year.

SELECT TO\_CHAR(ROUND(TO\_DATE('30-JUN-1999','DD-MON-YYYY'),'Y'),'DD-MON-YYYY') "Year" FROM DUAL;

Year

-------------

01-JAN-1999

(1 row)

SELECT TO\_CHAR(ROUND(TO\_DATE('01-JUL-1999','DD-MON-YYYY'),'Y'),'DD-MON-YYYY') "Year" FROM DUAL;

Year

-------------

01-JAN-2000

(1 row)

The following examples round to the nearest ISO year. The first example rounds to 2004 and the ISO year for 2004 begins on December 29th of 2003. The second example rounds to 2005 and the ISO year for 2005 begins on January 3rd of that same year.

(An ISO year begins on the first Monday from which a 7 day span, Monday thru Sunday, contains at least 4 days of the new year. Thus, it is possible for the beginning of an ISO year to start in December of the prior year.)

SELECT TO\_CHAR(ROUND(TO\_DATE('30-JUN-2004','DD-MON-YYYY'),'IYYY'),'DD-MON-YYYY') "ISO Year" FROM DUAL;

ISO Year

-------------

29-DEC-2003

(1 row)

SELECT TO\_CHAR(ROUND(TO\_DATE('01-JUL-2004','DD-MON-YYYY'),'IYYY'),'DD-MON-YYYY') "ISO Year" FROM DUAL;

ISO Year

-------------

03-JAN-2005

(1 row)

The following examples round to the nearest quarter.

SELECT ROUND(TO\_DATE('15-FEB-07','DD-MON-YY'),'Q') "Quarter" FROM DUAL;

Quarter

--------------------

01-JAN-07 00:00:00

(1 row)

SELECT ROUND(TO\_DATE('16-FEB-07','DD-MON-YY'),'Q') "Quarter" FROM DUAL;

Quarter

--------------------

01-APR-07 00:00:00

(1 row)

The following examples round to the nearest month.

SELECT ROUND(TO\_DATE('15-DEC-07','DD-MON-YY'),'MONTH') "Month" FROM DUAL;

Month

--------------------

01-DEC-07 00:00:00

(1 row)

SELECT ROUND(TO\_DATE('16-DEC-07','DD-MON-YY'),'MONTH') "Month" FROM DUAL;

Month

--------------------

01-JAN-08 00:00:00

(1 row)

The following examples round to the nearest week. The first day of 2007 lands on a Monday so in the first example, January 18th is closest to the Monday that lands on January 15th. In the second example, January 19th is closer to the Monday that falls on January 22nd.

SELECT ROUND(TO\_DATE('18-JAN-07','DD-MON-YY'),'WW') "Week" FROM DUAL;

Week

--------------------

15-JAN-07 00:00:00

(1 row)

SELECT ROUND(TO\_DATE('19-JAN-07','DD-MON-YY'),'WW') "Week" FROM DUAL;

Week

--------------------

22-JAN-07 00:00:00

(1 row)

The following examples round to the nearest ISO week. An ISO week begins on a Monday. In the first example, January 1, 2004 is closest to the Monday that lands on December 29, 2003. In the second example, January 2, 2004 is closer to the Monday that lands on January 5, 2004.

SELECT ROUND(TO\_DATE('01-JAN-04','DD-MON-YY'),'IW') "ISO Week" FROM DUAL;

ISO Week

--------------------

29-DEC-03 00:00:00

(1 row)

SELECT ROUND(TO\_DATE('02-JAN-04','DD-MON-YY'),'IW') "ISO Week" FROM DUAL;

ISO Week

--------------------

05-JAN-04 00:00:00

(1 row)

The following examples round to the nearest week where a week is considered to start on the same day as the first day of the month.

SELECT ROUND(TO\_DATE('05-MAR-07','DD-MON-YY'),'W') "Week" FROM DUAL;

Week

--------------------

08-MAR-07 00:00:00

(1 row)

SELECT ROUND(TO\_DATE('04-MAR-07','DD-MON-YY'),'W') "Week" FROM DUAL;

Week

--------------------

01-MAR-07 00:00:00

(1 row)

The following examples round to the nearest day.

SELECT ROUND(TO\_DATE('04-AUG-07 11:59:59 AM','DD-MON-YY HH:MI:SS AM'),'J') "Day" FROM DUAL;

Day

--------------------

04-AUG-07 00:00:00

(1 row)

SELECT ROUND(TO\_DATE('04-AUG-07 12:00:00 PM','DD-MON-YY HH:MI:SS AM'),'J') "Day" FROM DUAL;

Day

--------------------

05-AUG-07 00:00:00

(1 row)

The following examples round to the start of the nearest day of the week (Sunday).

SELECT ROUND(TO\_DATE('08-AUG-07','DD-MON-YY'),'DAY') "Day of Week" FROM DUAL;

Day of Week

--------------------

05-AUG-07 00:00:00

(1 row)

SELECT ROUND(TO\_DATE('09-AUG-07','DD-MON-YY'),'DAY') "Day of Week" FROM DUAL;

Day of Week

--------------------

12-AUG-07 00:00:00

(1 row)

The following examples round to the nearest hour.

SELECT TO\_CHAR(ROUND(TO\_DATE('09-AUG-07 08:29','DD-MON-YY HH:MI'),'HH'),'DD-MON-YY HH24:MI:SS') "Hour" FROM DUAL;

Hour

--------------------

09-AUG-07 08:00:00

(1 row)

SELECT TO\_CHAR(ROUND(TO\_DATE('09-AUG-07 08:30','DD-MON-YY HH:MI'),'HH'),'DD-MON-YY HH24:MI:SS') "Hour" FROM DUAL;

Hour

--------------------

09-AUG-07 09:00:00

(1 row)

The following examples round to the nearest minute.

SELECT TO\_CHAR(ROUND(TO\_DATE('09-AUG-07 08:30:29','DD-MON-YY HH:MI:SS'),'MI'),'DD-MON-YY HH24:MI:SS') "Minute" FROM DUAL;

Minute

--------------------

09-AUG-07 08:30:00

(1 row)

SELECT TO\_CHAR(ROUND(TO\_DATE('09-AUG-07 08:30:30','DD-MON-YY HH:MI:SS'),'MI'),'DD-MON-YY HH24:MI:SS') "Minute" FROM DUAL;

Minute

--------------------

09-AUG-07 08:31:00

(1 row)

#### TRUNC

The TRUNC function returns a date truncated according to a specified template pattern. If the template pattern is omitted, the date is truncated to the nearest day. The following table shows the template patterns for the TRUNC function.

Table 3‑3‑17 Template Date Patterns for the TRUNC Function

| Pattern | Description |
| --- | --- |
| CC, SCC | Returns January 1, cc01 where cc is first 2 digits of the given year |
| SYYY, YYYY, YEAR, SYEAR, YYY, YY, Y | Returns January 1, yyyy where yyyy is the given year |
| IYYY, IYY, IY, I | Returns the start date of the ISO year containing the given date |
| Q | Returns the first day of the quarter containing the given date |
| MONTH, MON, MM, RM | Returns the first day of the specified month |
| WW | Returns the largest date just prior to, or the same as the given date that corresponds to the same day of the week as the first day of the year |
| IW | Returns the start of the ISO week containing the given date |
| W | Returns the largest date just prior to, or the same as the given date that corresponds to the same day of the week as the first day of the month |
| DDD, DD, J | Returns the start of the day for the given date |
| DAY, DY, D | Returns the start of the week (Sunday) containing the given date |
| HH, HH12, HH24 | Returns the start of the hour |
| MI | Returns the start of the minute |

Following are examples of usage of the TRUNC function.

The following example truncates down to the hundred years unit.

SELECT TO\_CHAR(TRUNC(TO\_DATE('1951','YYYY'),'CC'),'DD-MON-YYYY') "Century" FROM DUAL;

Century

-------------

01-JAN-1901

(1 row)

The following example truncates down to the year.

SELECT TO\_CHAR(TRUNC(TO\_DATE('01-JUL-1999','DD-MON-YYYY'),'Y'),'DD-MON-YYYY') "Year" FROM DUAL;

Year

-------------

01-JAN-1999

(1 row)

The following example truncates down to the beginning of the ISO year.

SELECT TO\_CHAR(TRUNC(TO\_DATE('01-JUL-2004','DD-MON-YYYY'),'IYYY'),'DD-MON-YYYY') "ISO Year" FROM DUAL;

ISO Year

-------------

29-DEC-2003

(1 row)

The following example truncates down to the start date of the quarter.

SELECT TRUNC(TO\_DATE('16-FEB-07','DD-MON-YY'),'Q') "Quarter" FROM DUAL;

Quarter

--------------------

01-JAN-07 00:00:00

(1 row)

The following example truncates to the start of the month.

SELECT TRUNC(TO\_DATE('16-DEC-07','DD-MON-YY'),'MONTH') "Month" FROM DUAL;

Month

--------------------

01-DEC-07 00:00:00

(1 row)

The following example truncates down to the start of the week determined by the first day of the year. The first day of 2007 lands on a Monday so the Monday just prior to January 19th is January 15th.

SELECT TRUNC(TO\_DATE('19-JAN-07','DD-MON-YY'),'WW') "Week" FROM DUAL;

Week

--------------------

15-JAN-07 00:00:00

(1 row)

The following example truncates to the start of an ISO week. An ISO week begins on a Monday. January 2, 2004 falls in the ISO week that starts on Monday, December 29, 2003.

SELECT TRUNC(TO\_DATE('02-JAN-04','DD-MON-YY'),'IW') "ISO Week" FROM DUAL;

ISO Week

--------------------

29-DEC-03 00:00:00

(1 row)

The following example truncates to the start of the week where a week is considered to start on the same day as the first day of the month.

SELECT TRUNC(TO\_DATE('21-MAR-07','DD-MON-YY'),'W') "Week" FROM DUAL;

Week

--------------------

15-MAR-07 00:00:00

(1 row)

The following example truncates to the start of the day.

SELECT TRUNC(TO\_DATE('04-AUG-07 12:00:00 PM','DD-MON-YY HH:MI:SS AM'),'J') "Day" FROM DUAL;

Day

--------------------

04-AUG-07 00:00:00

(1 row)

The following example truncates to the start of the week (Sunday).

SELECT TRUNC(TO\_DATE('09-AUG-07','DD-MON-YY'),'DAY') "Day of Week" FROM DUAL;

Day of Week

--------------------

05-AUG-07 00:00:00

(1 row)

The following example truncates to the start of the hour.

SELECT TO\_CHAR(TRUNC(TO\_DATE('09-AUG-07 08:30','DD-MON-YY HH:MI'),'HH'),'DD-MON-YY HH24:MI:SS') "Hour" FROM DUAL;

Hour

--------------------

09-AUG-07 08:00:00

(1 row)

The following example truncates to the minute.

SELECT TO\_CHAR(TRUNC(TO\_DATE('09-AUG-07 08:30:30','DD-MON-YY HH:MI:SS'),'MI'),'DD-MON-YY HH24:MI:SS') "Minute" FROM DUAL;

Minute

--------------------

09-AUG-07 08:30:00

(1 row)

#### CURRENT DATE/TIME

Advanced Server provides a number of functions that return values related to the current date and time. These functions all return values based on the start time of the current transaction.

* CURRENT\_DATE
* CURRENT\_TIMESTAMP
* LOCALTIMESTAMP
* LOCALTIMESTAMP(precision)

CURRENT\_DATE returns the current date and time based on the start time of the current transaction. The value of CURRENT\_DATE will not change if called multiple times within a transaction.

SELECT CURRENT\_DATE FROM DUAL;

date

-----------

06-AUG-07

CURRENT\_TIMESTAMP returns the current date and time. When called from a single SQL statement, it will return the same value for each occurrence within the statement. If called from multiple statements within a transaction, may return different values for each occurrence. If called from a function, may return a different value than the value returned by CURRENT\_TIMESTAMP in the caller.

SELECT CURRENT\_TIMESTAMP, CURRENT\_TIMESTAMP FROM DUAL;

current\_timestamp | current\_timestamp

----------------------------------+----------------------------------

02-SEP-13 17:52:28.361473 +05:00 | 02-SEP-13 17:52:28.361474 +05:00

LOCALTIMESTAMP can optionally be given a precision parameter which causes the result to be rounded to that many fractional digits in the seconds field. Without a precision parameter, the result is given to the full available precision.

SELECT LOCALTIMESTAMP FROM DUAL;

timestamp

------------------------

06-AUG-07 16:11:35.973

(1 row)

SELECT LOCALTIMESTAMP(2) FROM DUAL;

timestamp

-----------------------

06-AUG-07 16:11:44.58

(1 row)

Since these functions return the start time of the current transaction, their values do not change during the transaction. This is considered a feature: the intent is to allow a single transaction to have a consistent notion of the “current” time, so that multiple modifications within the same transaction bear the same time stamp. Other database systems may advance these values more frequently.

#### NUMTODSINTERVAL

The NUMTODSINTERVAL function converts a numeric value to a time interval that includes day through second interval units. When calling the function, specify the smallest fractional interval type to be included in the result set. The valid interval types are DAY, HOUR, MINUTE, and SECOND.

The following example converts a numeric value to a time interval that includes days and hours:

SELECT numtodsinterval(100, ‘hour’);

numtodsinterval

---------------

4 days 04:00:00

(1 row)

The following example converts a numeric value to a time interval that includes minutes and seconds:

SELECT numtodsinterval(100, ‘second’);

numtodsinterval

---------------

1 min 40 secs

(1 row)

#### NUMTOYMINTERVAL

The NUMTOYMINTERVAL function converts a numeric value to a time interval that includes year through month interval units. When calling the function, specify the smallest fractional interval type to be included in the result set. The valid interval types are YEAR and MONTH.

The following example converts a numeric value to a time interval that includes years and months:

SELECT numtoyminterval(100, ‘month’);

numtoyminterval

---------------

8 years 4 mons

(1 row)

The following example converts a numeric value to a time interval that includes years only:

SELECT numtoyminterval(100, ‘year’);

numtoyminterval

---------------

100 years

(1 row)

### Sequence Manipulation Functions

This section describes Advanced Server’s functions for operating on sequence objects. Sequence objects (also called sequence generators or just sequences) are special single-row tables created with the CREATE SEQUENCE command. A sequence object is usually used to generate unique identifiers for rows of a table. The sequence functions, listed below, provide simple, multiuser-safe methods for obtaining successive sequence values from sequence objects.

sequence.NEXTVAL

sequence.CURRVAL

sequence is the identifier assigned to the sequence in the CREATE SEQUENCE command. The following describes the usage of these functions.

NEXTVAL

Advance the sequence object to its next value and return that value. This is done atomically: even if multiple sessions execute NEXTVAL concurrently, each will safely receive a distinct sequence value.

CURRVAL

Return the value most recently obtained by NEXTVAL for this sequence in the current session. (An error is reported if NEXTVAL has never been called for this sequence in this session.) Notice that because this is returning a session-local value, it gives a predictable answer whether or not other sessions have executed NEXTVAL since the current session did.

If a sequence object has been created with default parameters, NEXTVAL calls on it will return successive values beginning with 1. Other behaviors can be obtained by using special parameters in the CREATE SEQUENCE command.

Important: To avoid blocking of concurrent transactions that obtain numbers from the same sequence, a NEXTVAL operation is never rolled back; that is, 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.

### Conditional Expressions

The following section describes the SQL-compliant conditional expressions available in Advanced Server.

#### CASE

The SQL CASE expression is a generic conditional expression, similar to if/else statements in other languages:

CASE WHEN condition THEN result

[ WHEN ... ]

[ ELSE result ]

END

CASE clauses can be used wherever an expression is valid. condition is an expression that returns a BOOLEAN result. If the result is TRUE then the value of the CASE expression is the result that follows the condition. If the result is FALSE any subsequent WHEN clauses are searched in the same manner. If no WHEN condition is TRUE then the value of the CASE expression is the result in the ELSE clause. If the ELSE clause is omitted and no condition matches, the result is NULL.

An example:

SELECT \* FROM test;

a

---

1

2

3

(3 rows)

SELECT a,

CASE WHEN a=1 THEN 'one'

WHEN a=2 THEN 'two'

ELSE 'other'

END

FROM test;

a | case

---+-------

1 | one

2 | two

3 | other

(3 rows)

The data types of all the result expressions must be convertible to a single output type.

The following “simple” CASE expression is a specialized variant of the general form above:

CASE expression

WHEN value THEN result

[ WHEN ... ]

[ ELSE result ]

END

The expression is computed and compared to all the value specifications in the WHEN clauses until one is found that is equal. If no match is found, the result in the ELSE clause (or a null value) is returned.

The example above can be written using the simple CASE syntax:

SELECT a,

CASE a WHEN 1 THEN 'one'

WHEN 2 THEN 'two'

ELSE 'other'

END

FROM test;

a | case

---+-------

1 | one

2 | two

3 | other

(3 rows)

A CASE expression does not evaluate any subexpressions that are not needed to determine the result. For example, this is a possible way of avoiding a division-by-zero failure:

SELECT ... WHERE CASE WHEN x <> 0 THEN y/x > 1.5 ELSE false END;

#### COALESCE

The COALESCE function returns the first of its arguments that is not null. Null is returned only if all arguments are null.

COALESCE(value [, value2 ] ... )

It is often used to substitute a default value for null values when data is retrieved for display or further computation. For example:

SELECT COALESCE(description, short\_description, '(none)') ...

Like a CASE expression, COALESCE will not evaluate arguments that are not needed to determine the result; that is, arguments to the right of the first non-null argument are not evaluated. This SQL-standard function provides capabilities similar to NVL and IFNULL, which are used in some other database systems.

#### NULLIF

The NULLIF function returns a null value if value1 and value2 are equal; otherwise it returns value1.

NULLIF(value1, value2)

This can be used to perform the inverse operation of the COALESCE example given above:

SELECT NULLIF(value1, '(none)') ...

If value1 is (none), return a null, otherwise return value1.

#### NVL

The NVL function returns the first of its arguments that is not null. NVL evaluates the first expression; if that expression evaluates to NULL, NVL returns the second expression.

NVL(expr1, expr2)

The return type is the same as the argument types; all arguments must have the same data type (or be coercible to a common type). NVL returns NULL if all arguments are NULL.

The following example computes a bonus for non-commissioned employees, If an employee is a commissioned employee, this expression returns the employees commission; if the employee is not a commissioned employee (that is, his commission is NULL), this expression returns a bonus that is 10% of his salary.

bonus = NVL(emp.commission, emp.salary \* .10)

#### NVL2

NVL2 evaluates an expression, and returns either the second or third expression, depending on the value of the first expression. If the first expression is not NULL, NVL2 returns the value in expr2; if the first expression is NULL, NVL2 returns the value in expr3.

NVL2(expr1, expr2, expr3)

The return type is the same as the argument types; all arguments must have the same data type (or be coercible to a common type).

The following example computes a bonus for commissioned employees - if a given employee is a commissioned employee, this expression returns an amount equal to 110% of his commission; if the employee is not a commissioned employee (that is, his commission is NULL), this expression returns 0.

bonus = NVL2(emp.commission, emp.commission \* 1.1, 0)

#### GREATEST and LEAST

The GREATEST and LEAST functions select the largest or smallest value from a list of any number of expressions.

GREATEST(value [, value2 ] ... )

LEAST(value [, value2 ] ... )

The expressions must all be convertible to a common data type, which will be the type of the result. Null values in the list are ignored. The result will be null only if all the expressions evaluate to null.

Note that GREATEST and LEAST are not in the SQL standard, but are a common extension.

### Aggregate Functions

Aggregate functions compute a single result value from a set of input values. The built-in aggregate functions are listed in the following tables.

Table 3‑3‑18 General-Purpose Aggregate Functions

| Function | Argument Type | Return Type | Description |
| --- | --- | --- | --- |
| AVG(expression) | INTEGER, REAL, DOUBLE PRECISION, NUMBER | NUMBER for any integer type, DOUBLE PRECISION for a floating-point argument, otherwise the same as the argument data type | The average (arithmetic mean) of all input values |
| COUNT(\*) |  | BIGINT | Number of input rows |
| COUNT(expression) | Any | BIGINT | Number of input rows for which the value of expression is not null |
| MAX(expression) | Any numeric, string, date/time, or bytea type | Same as argument type | Maximum value of expression across all input values |
| MIN(expression) | Any numeric, string, date/time, or bytea type | Same as argument type | Minimum value of expression across all input values |
| SUM(expression) | INTEGER, REAL, DOUBLE PRECISION, NUMBER | BIGINT for SMALLINT or INTEGER arguments, NUMBER for BIGINT arguments, DOUBLE PRECISION for floating-point arguments, otherwise the same as the argument data type | Sum of expression across all input values |

It should be noted that except for COUNT, these functions return a null value when no rows are selected. In particular, SUM of no rows returns null, not zero as one might expect. The COALESCE function may be used to substitute zero for null when necessary.

The following table shows the aggregate functions typically used in statistical analysis. (These are separated out merely to avoid cluttering the listing of more-commonly-used aggregates.) Where the description mentions N, it means the number of input rows for which all the input expressions are non-null. In all cases, null is returned if the computation is meaningless, for example when N is zero.

Table 3‑3‑19 Aggregate Functions for Statistics

| Function | Argument Type | Return Type | Description |
| --- | --- | --- | --- |
| CORR(Y, X) | DOUBLE PRECISION | DOUBLE PRECISION | Correlation coefficient |
| COVAR\_POP(Y, X) | DOUBLE PRECISION | DOUBLE PRECISION | Population covariance |
| COVAR\_SAMP(Y, X) | DOUBLE PRECISION | DOUBLE PRECISION | Sample covariance |
| REGR\_AVGX(Y, X) | DOUBLE PRECISION | DOUBLE PRECISION | Average of the independent variable (sum(X) / N) |
| REGR\_AVGY(Y, X) | DOUBLE PRECISION | DOUBLE PRECISION | Average of the dependent variable (sum(Y) / N) |
| REGR\_COUNT(Y, X) | DOUBLE PRECISION | DOUBLE PRECISION | Number of input rows in which both expressions are nonnull |
| REGR\_INTERCEPT(Y, X) | DOUBLE PRECISION | DOUBLE PRECISION | y-intercept of the least-squares-fit linear equation determined by the (X, Y) pairs |
| REGR\_R2(Y, X) | DOUBLE PRECISION | DOUBLE PRECISION | Square of the correlation coefficient |
| REGR\_SLOPE(Y, X) | DOUBLE PRECISION | DOUBLE PRECISION | Slope of the least-squares-fit linear equation determined by the (X, Y) pairs |
| REGR\_SXX(Y, X) | DOUBLE PRECISION | DOUBLE PRECISION | Sum (X2) – sum (X)2 / N (“sum of squares” of the independent variable) |
| REGR\_SXY(Y, X) | DOUBLE PRECISION | DOUBLE PRECISION | Sum (X\*Y) – sum (X) \* sum (Y) / N (“sum of products” of independent times dependent variable) |
| REGR\_SYY(Y, X) | DOUBLE PRECISION | DOUBLE PRECISION | Sum (Y2) – sum (Y)2 / N (“sum of squares” of the dependent variable) |
| STDDEV(expression) | INTEGER, REAL, DOUBLE PRECISION, NUMBER | DOUBLE PRECISION for floating-point arguments, otherwise NUMBER | Historic alias for STDDEV\_SAMP |
| STDDEV\_POP(expression) | INTEGER, REAL, DOUBLE PRECISION, NUMBER | DOUBLE PRECISION for floating-point arguments, otherwise NUMBER | Population standard deviation of the input values |
| STDDEV\_SAMP(expression) | INTEGER, REAL, DOUBLE PRECISION, NUMBER | DOUBLE PRECISION for floating-point arguments, otherwise NUMBER | Sample standard deviation of the input values |
| VARIANCE(expression) | INTEGER, REAL, DOUBLE PRECISION, NUMBER | DOUBLE PRECISION for floating-point arguments, otherwise NUMBER | Historical alias for VAR\_SAMP |
| VAR\_POP(expression) | INTEGER, REAL, DOUBLE PRECISION, NUMBER | DOUBLE PRECISION for floating-point arguments, otherwise NUMBER | Population variance of the input values (square of the population standard deviation) |
| VAR\_SAMP(expression) | INTEGER, REAL, DOUBLE PRECISION, NUMBER | DOUBLE PRECISION for floating-point arguments, otherwise NUMBER | Sample variance of the input values (square of the sample standard deviation) |

### Subquery Expressions

This section describes the SQL-compliant subquery expressions available in Advanced Server. All of the expression forms documented in this section return Boolean (TRUE/FALSE) results.

#### EXISTS

The argument of EXISTS is an arbitrary SELECT statement, or subquery. The subquery is evaluated to determine whether it returns any rows. If it returns at least one row, the result of EXISTS is TRUE; if the subquery returns no rows, the result of EXISTS is FALSE.

EXISTS(subquery)

The subquery can refer to variables from the surrounding query, which will act as constants during any one evaluation of the subquery.

The subquery will generally only be executed far enough to determine whether at least one row is returned, not all the way to completion. It is unwise to write a subquery that has any side effects (such as calling sequence functions); whether the side effects occur or not may be difficult to predict.

Since the result depends only on whether any rows are returned, and not on the contents of those rows, the output list of the subquery is normally uninteresting. A common coding convention is to write all EXISTS tests in the form EXISTS(SELECT 1 WHERE ...). There are exceptions to this rule however, such as subqueries that use INTERSECT.

This simple example is like an inner join on deptno, but it produces at most one output row for each dept row, even though there are multiple matching emp rows:

SELECT dname FROM dept WHERE EXISTS (SELECT 1 FROM emp WHERE emp.deptno = dept.deptno);

dname

------------

ACCOUNTING

RESEARCH

SALES

(3 rows)

#### IN

The right-hand side is a parenthesized subquery, which must return exactly one column. The left-hand expression is evaluated and compared to each row of the subquery result. The result of IN is TRUE if any equal subquery row is found. The result is FALSE if no equal row is found (including the special case where the subquery returns no rows).

expression IN (subquery)

Note that if the left-hand expression yields NULL, or if there are no equal right-hand values and at least one right-hand row yields NULL, the result of the IN construct will be NULL, not FALSE. This is in accordance with SQL’s normal rules for Boolean combinations of null values.

As with EXISTS, it’s unwise to assume that the subquery will be evaluated completely.

#### NOT IN

The right-hand side is a parenthesized subquery, which must return exactly one column. The left-hand expression is evaluated and compared to each row of the subquery result. The result of NOT IN is TRUE if only unequal subquery rows are found (including the special case where the subquery returns no rows). The result is FALSE if any equal row is found.

expression NOT IN (subquery)

Note that if the left-hand expression yields NULL, or if there are no equal right-hand values and at least one right-hand row yields NULL, the result of the NOT IN construct will be NULL, not TRUE. This is in accordance with SQL’s normal rules for Boolean combinations of null values.

As with EXISTS, it’s unwise to assume that the subquery will be evaluated completely.

#### ANY/SOME

The right-hand side is a parenthesized subquery, which must return exactly one column. The left-hand expression is evaluated and compared to each row of the subquery result using the given operator, which must yield a Boolean result. The result of ANY is TRUE if any true result is obtained. The result is FALSE if no true result is found (including the special case where the subquery returns no rows).

expression operator ANY (subquery)

expression operator SOME (subquery)

SOME is a synonym for ANY. IN is equivalent to = ANY.

Note that if there are no successes and at least one right-hand row yields NULL for the operator’s result, the result of the ANY construct will be NULL, not FALSE. This is in accordance with SQL’s normal rules for Boolean combinations of null values.

As with EXISTS, it’s unwise to assume that the subquery will be evaluated completely.

#### ALL

The right-hand side is a parenthesized subquery, which must return exactly one column. The left-hand expression is evaluated and compared to each row of the subquery result using the given operator, which must yield a Boolean result. The result of ALL is TRUE if all rows yield true (including the special case where the subquery returns no rows). The result is FALSE if any false result is found. The result is NULL if the comparison does not return FALSE for any row, and it returns NULL for at least one row.

expression operator ALL (subquery)

NOT IN is equivalent to <> ALL. As with EXISTS, it’s unwise to assume that the subquery will be evaluated completely.

### Uniform Resource Locator Functions

This section describes functions that perform operations based on the Uniform Resource Locator. The Uniform Resource Locator (URL) is a character string that provides the address of a network resource. Typical usage of URLs is for web pages, which are accessed using the Hypertext Transfer Protocol (HTTP) protocol.

#### EDB\_GET\_URL\_AS\_BYTEA

The EDB\_GET\_URL\_AS\_BYTEA function returns content from the user-specified URL in one continuous BYTEA string. The signature is:

BYTEA EDB\_GET\_URL\_AS\_BYTEA  
(  
 url TEXT  
)

Parameters

url

url is the Uniform Resource Locator from which the function will return content.

Example

The following function retrieves the content of the specified URL and displays the first few lines in chunks of 40 bytes (80 hexadecimal characters) per line. The function also returns the total content length in number of bytes.

CREATE OR REPLACE FUNCTION get\_url\_bytea(

p\_url TEXT

) RETURNS INTEGER

AS $$

DECLARE

v\_data BYTEA;

v\_line TEXT;

v\_start INTEGER;

v\_line\_count INTEGER := 0;

v\_line\_length INTEGER := 40;

BEGIN

v\_data := EDB\_GET\_URL\_AS\_BYTEA(p\_url);

FOR i IN 1 .. 10 LOOP

v\_start := (v\_line\_count \* v\_line\_length) + 1;

v\_line := SUBSTR(v\_data, v\_start, v\_line\_length);

RAISE INFO '%', v\_line;

v\_line\_count := v\_line\_count + 1;

END LOOP;

RETURN OCTET\_LENGTH(v\_data);

END;

$$ LANGUAGE 'plpgsql';

The following is the output from the example.

edb=# SELECT get\_url\_bytea('http://www.enterprisedb.com');

INFO: \x3c21444f43545950452068746d6c205055424c494320222d2f2f5733432f2f445444205848544d4c

INFO: \x20312e30205374726963742f2f454e220d0a202022687474703a2f2f7777772e77332e6f72672f54

INFO: \x522f7868746d6c312f4454442f7868746d6c312d7374726963742e647464223e0d0a3c68746d6c20

INFO: \x786d6c6e733d22687474703a2f2f7777772e77332e6f72672f313939392f7868746d6c2220786d6c

INFO: \x3a6c616e673d22656e22206c616e673d22656e22206469723d226c7472223e0d0a0d0a20203c212d

INFO: \x2d205f5f5f5f5f5f5f5f5f5f5f5f5f5f5f5f5f5f5f5f5f5f5f5f5f5f5f2048454144205f5f5f5f5f

INFO: \x5f5f5f5f5f5f5f5f5f5f5f5f5f5f5f5f5f5f5f5f5f5f202d2d3e0d0a0d0a20203c686561643e0a3c

INFO: \x6d65746120687474702d65717569763d22436f6e74656e742d547970652220636f6e74656e743d22

INFO: \x746578742f68746d6c3b20636861727365743d7574662d3822202f3e0d0a0d0a0d0a202020203c74

INFO: \x69746c653e456e74657270726973654442207c2054686520506f7374677265732044617461626173

get\_url\_bytea

---------------

84216

(1 row)

#### EDB\_GET\_URL\_AS\_TEXT

The EDB\_GET\_URL\_AS\_TEXT function returns content from the user-specified URL in one continuous TEXT string. The signature is:

TEXT EDB\_GET\_URL\_AS\_TEXT  
(  
 url TEXT  
)

Parameters

url

url is the Uniform Resource Locator from which the function will return content.

Example

The following function retrieves the content of the specified URL and displays the first few lines in chunks of 80 characters per line. The function also returns the total content length in number of bytes.

CREATE OR REPLACE FUNCTION get\_url\_text(

p\_url TEXT

) RETURNS INTEGER

AS $$

DECLARE

v\_data TEXT;

v\_line TEXT;

v\_start INTEGER;

v\_line\_count INTEGER := 0;

v\_line\_length INTEGER := 80;

BEGIN

v\_data := EDB\_GET\_URL\_AS\_TEXT(p\_url);

FOR i IN 1 .. 5 LOOP

v\_start := (v\_line\_count \* v\_line\_length) + 1;

v\_line := SUBSTR(v\_data, v\_start, v\_line\_length);

RAISE INFO '%', v\_line;

v\_line\_count := v\_line\_count + 1;

END LOOP;

RETURN OCTET\_LENGTH(v\_data);

END;

$$ LANGUAGE 'plpgsql';

The following is the output from the example.

edb=# SELECT get\_url\_text('http://www.enterprisedb.com');

INFO: <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"

"http://www.w3.org/T

INFO: R/xhtml1/DTD/xhtml1-strict.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" xml

INFO: :lang="en" lang="en" dir="ltr">

<!-- \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ HEAD \_\_\_\_\_

INFO: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ -->

<head>

<meta http-equiv="Content-Type" content="

INFO: text/html; charset=utf-8" />

<title>EnterpriseDB | The Postgres Databas

get\_url\_text

--------------

83892

(1 row)

## Table Partitioning

In a partitioned table, one logically large table is broken into smaller physical pieces. Partitioning can provide several benefits:

* Query performance can be improved dramatically in certain situations, particularly when most of the heavily accessed rows of the table are in a single partition or a small number of partitions. Partitioning allows you to omit the partition column from the front of an index, reducing index size and making it more likely that the heavily used parts of the index fits in memory.
* When a query or update accesses a large percentage of a single partition, performance may improve because the server will perform a sequential scan of the partition instead of using an index and random access reads scattered across the whole table.
* A bulk load (or unload) can be implemented by adding or removing partitions, if you plan that requirement into the partitioning design. ALTER TABLE is far faster than a bulk operation. It also entirely avoids the VACUUM overhead caused by a bulk DELETE.
* Seldom-used data can be migrated to less-expensifve (or slower) storage media.

Table partitioning is worthwhile only when a table would otherwise be very large. The exact point at which a table will benefit from partitioning depends on the application; a good rule of thumb is that the size of the table should exceed the physical memory of the database server.

This document discusses the aspects of table partitioning compatible with Oracle databases that are supported by Advanced Server.

The PostgreSQL 9.5 INSERT… ON CONFLICT DO NOTHING/UPDATE clause (commonly known as UPSERT) is not supported on Oracle-styled partitioned tables. If you include the ON CONFLICT DO NOTHING/UPDATE clause when invoking the INSERT command to add data to a partitioned table, the server will return an error.

### Selecting a Partition Type

When you create a partitioned table, you specify LIST, RANGE, or HASH partitioning rules. The partitioning rules provide a set of constraints that define the data that is stored in each partition. As new rows are added to the partitioned table, the server uses the partitioning rules to decide which partition should contain each row.

Advanced Server can also use the partitioning rules to enforce partition pruning, improving performance when responding to user queries. When selecting a partitioning type and partitioning keys for a table, you should take into consideration how the data that is stored within a table will be queried, and include often-queried columns in the partitioning rules.

***List Partitioning***

When you create a list-partitioned table, you specify a single partitioning key column. When adding a row to the table, the server compares the key values specified in the partitioning rule to the corresponding column within the row. If the column value matches a value in the partitioning rule, the row is stored in the partition named in the rule.

***Range Partitioning***

When you create a range-partitioned table, you specify one or more partitioning key columns. When you add a new row to the table, the server compares the value of the partitioning key (or keys) to the corresponding column (or columns) in a table entry. If the column values satisfy the conditions specified in the partitioning rule, the row is stored in the partition named in the rule.

***Hash Partitioning***

When you create a hash-partitioned table, you specify one or more partitioning key columns. Data is divided into (approx.) equal-sized partitions amongst the specified partitions. When you add a row to a hash-partitioned table, the server computes a hash value for the data in the specified column (or columns), and stores the row in a partition according to the hash value.

***Subpartitioning***

Subpartitioning breaks a partitioned table into smaller subsets that may or may not be stored on the same server. A table is typically subpartitioned by a different set of columns, and may be a different subpartitioning type than the parent partition. If one partition is subpartitioned, then each partition will have at least one subpartition.

If a table is subpartitioned, no data will be stored in any of the partition tables; the data will be stored instead in the corresponding subpartitions.

### Using Partition Pruning

Advanced Server's query planner uses *partition* *pruning* to compute an efficient plan to locate a row (or rows) that matches the conditions specified in the WHERE clause of a SELECT statement. To successfully prune partitions from an execution plan, the WHERE clause must constrain the information that is compared to the partitioning key column specified when creating the partitioned table. When querying a:

* list-partitioned table, partition pruning is effective when the WHERE clause compares a literal value to the partitioning key using operators like equal (=) or AND.
* range-partitioned table, partition pruning is effective when the WHERE clause compares a literal value to a partitioning key using operators such as equal (=), less than (<), or greater than (>).
* hash-partitioned table, partition pruning is effective when the WHERE clause compares a literal value to the partitioning key using an operator such as equal (=).

The partition pruning mechanism uses two optimization techniques:

* Fast Pruning
* Constraint exclusion

Partition pruning techniques limit the search for data to only those partitions in which the values for which you are searching might reside. Both pruning techniques remove partitions from a query's execution plan, increasing performance.

The difference between the fast pruning and constraint exclusion is that fast pruning understands the relationship between the partitions in an Oracle-partitioned table, while constraint exclusion does not. For example, when a query searches for a specific value within a list-partitioned table, fast pruning can reason that only a specific partition may hold that value, while constraint exclusion must examine the constraints defined for each partition. Fast pruning occurs early in the planning process to reduce the number of partitions that the planner must consider, while constraint exclusion occurs late in the planning process.

***Using Constraint Exclusion***

The constraint\_exclusion parameter controls constraint exclusion. The constraint\_exclusion parameter may have a value of on, off, or partition. To enable constraint exclusion, the parameter must be set to *either* partition or on. By default, the parameter is set to partition.

For more information about constraint exclusion, see:

<http://www.postgresql.org/docs/9.5/static/ddl-partitioning.html>

When constraint exclusion is enabled, the server examines the constraints defined for each partition to determine if that partition can satisfy a query.

When you execute a SELECT statement that *does not* contain a WHERE clause, the query planner must recommend an execution plan that searches the entire table. When you execute a SELECT statement that *does* contain a WHERE clause, the query planner determines in which partition that row would be stored, and sends query fragments to that partition, pruning the partitions that could not contain that row from the execution plan. If you are are not using partitioned tables, disabling constraint exclusion may improve performance.

**Fast Pruning**

Like constraint exclusion, fast pruning can only optimize queries that include a WHERE (or join) clause, and only when the qualifiers in the WHERE clause match a certain form. In both cases, the query planner will avoid searching for data within partitions that cannot possibly hold the data required by the query.

Fast pruning is controlled by a boolean configuration parameter named edb\_enable\_pruning. If edb\_enable\_pruning is ON, Advanced Server will fast prune certain queries. If edb\_enable\_pruning is OFF, the server will disable fast pruning.

Please note: Fast pruning cannot optimize queries against subpartitioned tables or optimize queries against range-partitioned tables that are partitioned on more than one column.

For LIST-partitioned tables, Advanced Server can fast prune queries that contain a WHERE clause that constrains a partitioning column to a literal value. For example, given a LIST-partitioned table such as:

CREATE TABLE sales\_hist(..., country text, ...)   
PARTITION BY LIST(country)

(  
 PARTITION americas VALUES('US', 'CA', 'MX'),  
 PARTITION europe VALUES('BE', 'NL', 'FR'),  
 PARTITION asia VALUES('JP', 'PK', 'CN'),  
 PARTITION others VALUES(DEFAULT)  
)

Fast pruning can reason about WHERE clauses such as:

WHERE country = 'US'

WHERE country IS NULL;

Given the first WHERE clause, fast pruning would eliminate partitions europe, asia, and others because those partitions cannot hold rows that satisfy the qualifier: WHERE country = 'US'.

Given the second WHERE clause, fast pruning would eliminate partitions americas, europe, and asia because because those partitions cannot hold rows where country IS NULL.

The operator specified in the WHERE clause must be an equal sign (=) or the equality operator appropriate for the data type of the partitioning column.

For range-partitioned tables, Advanced Server can fast prune queries that contain a WHERE clause that constrains a partitioning column to a literal value, but the operator may be any of the following:

>  
>=  
=  
<=  
<

Fast pruning will also reason about more complex expressions involving AND and BETWEEN operators, such as:

WHERE size > 100 AND size <= 200  
WHERE size BETWEEN 100 AND 200

But cannot prune based on expressions involving OR or IN.

For example, when querying a RANGE-partitioned table, such as:

CREATE TABLE boxes(id int, size int, color text)   
 PARTITION BY RANGE(size)  
(  
 PARTITION small VALUES LESS THAN(100),  
 PARTITION medium VALUES LESS THAN(200),  
 PARTITION large VALUES LESS THAN(300)  
)

Fast pruning can reason about WHERE clauses such as:

WHERE size > 100 -- scan partitions 'medium' and 'large'

WHERE size >= 100 -- scan partitions 'medium' and 'large'

WHERE size = 100 -- scan partition 'medium'

WHERE size <= 100 -- scan partitions 'small' and 'medium'

WHERE size < 100 -- scan partition 'small'

WHERE size > 100 AND size < 199 -- scan partition 'medium'

WHERE size BETWEEN 100 AND 199 -- scan partition 'medium'

WHERE color = 'red' AND size = 100 -- scan 'medium'

WHERE color = 'red' AND (size > 100 AND size < 199) -- scan 'medium'

In each case, fast pruning requires that the qualifier must refer to a partitioning column and literal value (or IS NULL/IS NOT NULL).

Note that fast pruning can also optimize DELETE and UPDATE statements containing WHERE clauses of the forms described above.

### Example - Partition Pruning

The EXPLAIN statement displays the execution plan of a statement. You can use the EXPLAIN statement to confirm that Advanced Server is pruning partitions from the execution plan of a query.

To demonstrate the efficiency of partition pruning, first create a simple table:

CREATE TABLE sales

(

dept\_no number,

part\_no varchar2,

country varchar2(20),

date date,

amount number

)

PARTITION BY LIST(country)

(

PARTITION europe VALUES('FRANCE', 'ITALY'),

PARTITION asia VALUES('INDIA', 'PAKISTAN'),

PARTITION americas VALUES('US', 'CANADA')

);

Then, perform a constrained query that includes the EXPLAIN statement:

EXPLAIN (COSTS OFF) SELECT \* FROM sales WHERE country = 'INDIA';

The resulting query plan shows that the server will scan only the sales\_asia table - the table in which a row with a country value of INDIA would be stored:

edb=# EXPLAIN (COSTS OFF) SELECT \* FROM sales WHERE country = 'INDIA';

QUERY PLAN

---------------------------------------------------

Append

-> Seq Scan on sales

Filter: ((country)::text = 'INDIA'::text)

-> Seq Scan on sales\_asia

Filter: ((country)::text = 'INDIA'::text)

(5 rows)

If you perform a query that searches for a row that matches a value not included in the partitioning key:

EXPLAIN (COSTS OFF) SELECT \* FROM sales WHERE dept\_no = '30';

The resulting query plan shows that the server must look in all of the partitions to locate the rows that satisfy the query:

edb=# EXPLAIN (COSTS OFF) SELECT \* FROM sales WHERE dept\_no = '30';

QUERY PLAN

-----------------------------------------

Append

-> Seq Scan on sales

Filter: (dept\_no = 30::numeric)

-> Seq Scan on sales\_europe

Filter: (dept\_no = 30::numeric)

-> Seq Scan on sales\_asia

Filter: (dept\_no = 30::numeric)

-> Seq Scan on sales\_americas

Filter: (dept\_no = 30::numeric)

(9 rows)

Constraint exclusion also applies when querying subpartitioned tables:

CREATE TABLE sales

(

dept\_no number,

part\_no varchar2,

country varchar2(20),

date date,

amount number

)

PARTITION BY RANGE(date) SUBPARTITION BY LIST (country)

(  
 PARTITION "2011" VALUES LESS THAN('01-JAN-2012')

(

SUBPARTITION europe\_2011 VALUES ('ITALY', 'FRANCE'),

SUBPARTITION asia\_2011 VALUES ('PAKISTAN', 'INDIA'),

SUBPARTITION americas\_2011 VALUES ('US', 'CANADA')

),

PARTITION "2012" VALUES LESS THAN('01-JAN-2013')

(

SUBPARTITION europe\_2012 VALUES ('ITALY', 'FRANCE'),

SUBPARTITION asia\_2012 VALUES ('PAKISTAN', 'INDIA'),

SUBPARTITION americas\_2012 VALUES ('US', 'CANADA')

),  
 PARTITION "2013" VALUES LESS THAN('01-JAN-2014')

(

SUBPARTITION europe\_2013 VALUES ('ITALY', 'FRANCE'),

SUBPARTITION asia\_2013 VALUES ('PAKISTAN', 'INDIA'),

SUBPARTITION americas\_2013 VALUES ('US', 'CANADA')

)

);

When you query the table, the query planner prunes any partitions or subpartitions from the search path that cannot possibly contain the desired result set:

edb=# EXPLAIN (COSTS OFF) SELECT \* FROM sales WHERE country = 'US' AND date = 'Dec 12, 2012';

QUERY PLAN

-----------------------------------------------------------------------------

Append

-> Seq Scan on sales

Filter: (((country)::text = 'US'::text) AND (date = '12-DEC-12 00:00:00'::timestamp without time zone))

-> Seq Scan on sales\_2012

Filter: (((country)::text = 'US'::text) AND (date = '12-DEC-12 00:00:00'::timestamp without time zone))

-> Seq Scan on sales\_americas\_2012

Filter: (((country)::text = 'US'::text) AND (date = '12-DEC-12 00:00:00'::timestamp without time zone))

(7 rows)

## Partitioning Command Syntax

The following sections provide information about using the table partitioning syntax supported by Advanced Server.

### CREATE TABLE…PARTITION BY

Use the PARTITION BY clause of the CREATE TABLE command to create a partitioned table with data distributed amongst one or more partitions (and subpartitions). The command syntax comes in the following forms:

***List Partitioning Syntax***

Use the first form to create a list-partitioned table:

CREATE TABLE [ schema. ]*table\_name*   
 *table\_definition*

PARTITION BY LIST(*column*)

[SUBPARTITION BY {RANGE|LIST} (*column*[, *column* ]...)]

(*list\_partition\_definition[, list\_partition\_definition]...*);

Where *list\_partition\_definition* is:

PARTITION [*partition\_name*]

VALUES (*value*[, *value*]...)

[TABLESPACE *tablespace\_name*]

[(*subpartition*, ...)]

***Range Partitioning Syntax***

Use the second form to create a range-partitioned table:

CREATE TABLE [ *schema*. ]*table\_name*   
 *table\_definition*  
 PARTITION BY RANGE(*column*[, *column* ]...)

[SUBPARTITION BY {RANGE|LIST} (*column*[, *column* ]...)]

(*range\_partition\_definition[, range\_partition\_definition]...*);

Where *range\_partition\_definition* is:

PARTITION [*partition\_name*]

VALUES LESS THAN (*value*[, *value*]...)

[TABLESPACE *tablespace\_name*]

[(*subpartition*, ...)]

***Hash Partitioning Syntax***

Use the third form to create a hash-partitioned table:

CREATE TABLE [ *schema*. ]*table\_name*

*table\_definition*

PARTITION BY HASH(*column*[, *column* ]...)  
 [SUBPARTITION BY {RANGE|LIST|HASH} (*column*[, *column* ]...)]  
 (*hash\_partition\_definition*[, *hash\_partition\_definition*]...);

Where *hash\_partition\_definition* is:

[PARTITION *partition\_name*]

[TABLESPACE *tablespace\_name*]  
 [(*subpartition*, ...)]

***Subpartitioning Syntax***

*subpartition* may be one of the following:

{*list*\_*subpartition* | *range*\_*subpartition* | *hash\_subpartition*}

where *list*\_*subpartition* is:

SUBPARTITION [*subpartition\_name*]

VALUES (*value*[, *value*]...)

[TABLESPACE *tablespace\_name*]

where *range*\_*subpartition* is:

SUBPARTITION [*subpartition\_name*]

VALUES LESS THAN (*value*[, *value*]...)

[TABLESPACE *tablespace\_name*]

where hash\_subpartition is:

[SUBPARTITION subpartition\_name]

[TABLESPACE tablespace\_name]

***Description***

The CREATE TABLE… PARTITION BY command creates a table with one or more partitions; each partition may have one or more subpartitions. There is no upper limit to the number of defined partitions, but if you include the PARTITION BY clause, you must specify at least one partitioning rule. The resulting table will be owned by the user that creates it.

Use the PARTITION BY LIST clause to divide a table into partitions based on the values entered in a specified column. Each partitioning rule must specify at least one literal value, but there is no upper limit placed on the number of values you may specify. Include a rule that specifies a matching value of DEFAULT to direct any un-qualified rows to the given partition; for more information about using the DEFAULT keyword, see [Section 3.6](#_Handling_Stray_Values_16).

Use the PARTITION BY RANGE clause to specify boundary rules by which to create partitions. Each partitioning rule must contain at least one column of a data type that has two operators (i.e., a greater-than or equal to operator, and a less-than operator). Range boundaries are evaluated against a LESS THAN clause and are non-inclusive; a date boundary of January 1, 2013 will include only those date values that fall on or before December 31, 2012.

Range partition rules must be specified in ascending order. INSERT commands that store rows with values that exceed the top boundary of a range-partitioned table will fail unless the partitioning rules include a boundary rule that specifies a value of MAXVALUE. If you do not include a MAXVALUE partitioning rule, any row that exceeds the maximum limit specified by the boundary rules will result in an error.

For more information about using the MAXVALUE keyword, see [Section 3.6](#_Handling_Stray_Values_11).

Use the TABLESPACE keyword to specify the name of a tablespace on which a partition or subpartition will reside; if you do not specify a tablespace, the partition or subpartition will reside in the default tablespace.

If a table definition includes the SUBPARTITION BY clause, each partition within that table will have at least one subpartition. Each subpartition may be explicitly defined or system-defined.

If the subpartition is system-defined, the server-generated subpartition will reside in the default tablespace, and the name of the subpartition will be assigned by the server. The server will create:

* A DEFAULT subpartition if the SUBPARTITION BY clause specifies LIST.
* A MAXVALUE subpartition if the SUBPARTITION BY clause specifies RANGE.

The server will generate a subpartition name that is a combination of the partition table name and a unique identifier. You can query the ALL\_TAB\_SUBPARTITIONS table to review a complete list of subpartition names.

**Parameters**

table\_name

The name (optionally schema-qualified) of the table to be created.

table\_definition

The column names, data types, and constraint information as described in the PostgreSQL core documentation for the CREATE TABLE statement, available at:

<http://www.postgresql.org/docs/9.5/static/sql-createtable.html>

partition\_name

The name of the partition to be created. Partition names must be unique amongst all partitions and subpartitions, and must follow the naming conventions for object identifiers.

subpartition\_name

The name of the subpartition to be created. Subpartition names must be unique amongst all partitions and subpartitions, and must follow the naming conventions for object identifiers.

column

The name of a column on which the partitioning rules are based. Each row will be stored in a partition that corresponds to the *value* of the specified column(s).

(*value*[, *value*]...)

Use value to specify a quoted literal value (or comma-delimited list of literal values) by which table entries will be grouped into partitions. Each partitioning rule must specify at least one value, but there is no limit placed on the number of values specified within a rule. value may be NULL, DEFAULT (if specifying a LIST partition), or MAXVALUE (if specifying a RANGE partition).

When specifying rules for a list-partitioned table, include the DEFAULT keyword in the last partition rule to direct any un-matched rows to the given partition. If you do not include a rule that includes a value of DEFAULT, any INSERT statement that attempts to add a row that does not match the specified rules of at least one partition will fail, and return an error.

When specifying rules for a list-partitioned table, include the MAXVALUE keyword in the last partition rule to direct any un-categorized rows to the given partition. If you do not include a MAXVALUE partition, any INSERT statement that attempts to add a row where the partitioning key is greater than the highest value specified will fail, and return an error.

tablespace\_name

The name of the tablespace in which the partition or subpartition resides.

#### Example - PARTITION BY LIST

The following example creates a partitioned table (sales) using the PARTITION BY LIST clause. The sales table stores information in three partitions (europe, asia, and americas):

CREATE TABLE sales

(

dept\_no number,

part\_no varchar2,

country varchar2(20),

date date,

amount number

)

PARTITION BY LIST(country)

(

PARTITION europe VALUES('FRANCE', 'ITALY'),

PARTITION asia VALUES('INDIA', 'PAKISTAN'),

PARTITION americas VALUES('US', 'CANADA')

);

The resulting table is partitioned by the value specified in the country column:

acctg=# SELECT partition\_name, high\_value from ALL\_TAB\_PARTITIONS;

partition\_name | high\_value

----------------+---------------------

americas | 'US', 'CANADA'

asia | 'INDIA', 'PAKISTAN'

europe | 'FRANCE', 'ITALY'

(3 rows)

* Rows with a value of US or CANADA in the country column are stored in the americas partition.
* Rows with a value of INDIA or PAKISTAN in the country column are stored in the asia partition.
* Rows with a value of FRANCE or ITALY in the country column are stored in the europe partition.

The server would evaluate the following statement against the partitioning rules, and store the row in the europe partition:

INSERT INTO sales VALUES (10, '9519a', 'FRANCE', '18-Aug-2012', '650000');

#### Example - PARTITION BY RANGE

The following example creates a partitioned table (sales) using the PARTITION BY RANGE clause. The sales table stores information in four partitions (q1\_2012, q2\_2012, q3\_2012 and q4\_2012) :

CREATE TABLE sales

(

dept\_no number,

part\_no varchar2,

country varchar2(20),

date date,

amount number

)

PARTITION BY RANGE(date)

(

PARTITION q1\_2012   
 VALUES LESS THAN('2012-Apr-01'),

PARTITION q2\_2012   
 VALUES LESS THAN('2012-Jul-01'),

PARTITION q3\_2012   
 VALUES LESS THAN('2012-Oct-01'),

PARTITION q4\_2012   
 VALUES LESS THAN('2013-Jan-01')  
);

The resulting table is partitioned by the value specified in the date column:

acctg=# SELECT partition\_name, high\_value from ALL\_TAB\_PARTITIONS;

partition\_name | high\_value

----------------+---------------

q4\_2012 | '2013-Jan-01'

q3\_2012 | '2012-Oct-01'

q2\_2012 | '2012-Jul-01'

q1\_2012 | '2012-Apr-01'

(4 rows)

* Any row with a value in the date column before April 1, 2012 is stored in a partition named q1\_2012.
* Any row with a value in the date column before July 1, 2012 is stored in a partition named q2\_2012.
* Any row with a value in the date column before October 1, 2012 is stored in a partition named q3\_2012.
* Any row with a value in the date column before January 1, 2013 is stored in a partition named q4\_2012.

The server would evaluate the following statement against the partitioning rules and store the row in the q3\_2012 partition:

INSERT INTO sales VALUES (10, '9519a', 'FRANCE', '18-Aug-2012', '650000');

#### Example - PARTITION BY HASH

The following example creates a partitioned table (sales) using the PARTITION BY HASH clause. The sales table stores information in three partitions (p1, p2, and p3:

CREATE TABLE sales

(

dept\_no number,

part\_no varchar2,

country varchar2(20),

date date,

amount number

)

PARTITION BY HASH (part\_no)

(

PARTITION p1,

PARTITION p2,

PARTITION p3

);

The table is partitioned by the hash value of the value specified in the part\_no column:

acctg=# SELECT partition\_name, partition\_position from ALL\_TAB\_PARTITIONS;

partition\_name | partition\_position

----------------+--------------------

p3 | 3

p2 | 2

p1 | 1

(3 rows)

The server will evaluate the hash value of the part\_no column, and distribute the rows into approximately equal partitions.

#### Example - PARTITION BY RANGE, SUBPARTITION BY LIST

The following example creates a partitioned table (sales) that is first partitioned by the transaction date; the range partitions (q1\_2012, q2\_2012, q3\_2012 and q4\_2012) are then list-subpartitioned using the value of the country column.

CREATE TABLE sales  
(  
 dept\_no number,  
 part\_no varchar2,  
 country varchar2(20),  
 date date,  
 amount number  
)  
PARTITION BY RANGE(date)  
 SUBPARTITION BY LIST(country)  
 (  
 PARTITION q1\_2012   
 VALUES LESS THAN('2012-Apr-01')  
 (  
 SUBPARTITION q1\_europe VALUES ('FRANCE', 'ITALY'),  
 SUBPARTITION q1\_asia VALUES ('INDIA', 'PAKISTAN'),  
 SUBPARTITION q1\_americas VALUES ('US', 'CANADA')  
 ),  
 PARTITION q2\_2012   
 VALUES LESS THAN('2012-Jul-01')  
 (  
 SUBPARTITION q2\_europe VALUES ('FRANCE', 'ITALY'),  
 SUBPARTITION q2\_asia VALUES ('INDIA', 'PAKISTAN'),  
 SUBPARTITION q2\_americas VALUES ('US', 'CANADA')  
 ),  
 PARTITION q3\_2012   
 VALUES LESS THAN('2012-Oct-01')  
 (  
 SUBPARTITION q3\_europe VALUES ('FRANCE', 'ITALY'),  
 SUBPARTITION q3\_asia VALUES ('INDIA', 'PAKISTAN'),  
 SUBPARTITION q3\_americas VALUES ('US', 'CANADA')  
 ),  
 PARTITION q4\_2012   
 VALUES LESS THAN('2013-Jan-01')   
 (  
 SUBPARTITION q4\_europe VALUES ('FRANCE', 'ITALY'),  
 SUBPARTITION q4\_asia VALUES ('INDIA', 'PAKISTAN'),  
 SUBPARTITION q4\_americas VALUES ('US', 'CANADA')  
 )  
);

This statement creates a table with four partitions; each partition has three subpartitions:

acctg=# SELECT subpartition\_name, high\_value, partition\_name FROM ALL\_TAB\_SUBPARTITIONS;

subpartition\_name | high\_value | partition\_name

-------------------+---------------------+----------------

q4\_asia | 'INDIA', 'PAKISTAN' | q4\_2012

q4\_europe | 'FRANCE', 'ITALY' | q4\_2012

q4\_americas | 'US', 'CANADA' | q4\_2012

q3\_americas | 'US', 'CANADA' | q3\_2012

q3\_asia | 'INDIA', 'PAKISTAN' | q3\_2012

q3\_europe | 'FRANCE', 'ITALY' | q3\_2012

q2\_americas | 'US', 'CANADA' | q2\_2012

q2\_asia | 'INDIA', 'PAKISTAN' | q2\_2012

q2\_europe | 'FRANCE', 'ITALY' | q2\_2012

q1\_americas | 'US', 'CANADA' | q1\_2012

q1\_asia | 'INDIA', 'PAKISTAN' | q1\_2012

q1\_europe | 'FRANCE', 'ITALY' | q1\_2012

(12 rows)

When a row is added to this table, the value in the date column is compared to the values specified in the range partitioning rules, and the server selects the partition in which the row should reside. The value in the country column is then compared to the values specified in the list subpartitioning rules; when the server locates a match for the value, the row is stored in the corresponding subpartition.

Any row added to the table will be stored in a subpartition, so the partitions will contain no data.

The server would evaluate the following statement against the partitioning and subpartitioning rulesand store the row in the q3\_europe partition:

INSERT INTO sales VALUES (10, '9519a', 'FRANCE', '18-Aug-2012', '650000');

### ALTER TABLE...ADD PARTITION

Use the ALTER TABLE… ADD PARTITION command to add a partition to an existing partitioned table. The syntax is:

ALTER TABLE *table*\_*name* ADD PARTITION *partition*\_*definition*;

Where *partition\_definition* is:

{*list\_partition* | *range\_partition* }

and *list*\_*partition* is:

PARTITION [*partition\_name*]

VALUES (*value*[, *value*]...)

[TABLESPACE *tablespace\_name*]

[(*subpartition*, ...)]

and *range\_partition* is:

PARTITION [*partition\_name*]

VALUES LESS THAN (*value*[, *value*]...)

[TABLESPACE *tablespace\_name*]

[(*subpartition*, ...)]

Where *subpartition* is:

{*list*\_*subpartition* | *range*\_*subpartition | hash\_subpartition*}

and *list*\_*subpartition* is:

SUBPARTITION [*subpartition\_name*]

VALUES (*value*[, *value*]...)

[TABLESPACE *tablespace\_name*]

and *range*\_*subpartition* is:

SUBPARTITION [*subpartition\_name*]

VALUES LESS THAN (*value*[, *value*]...)

[TABLESPACE *tablespace\_name*]

***Description***

The ALTER TABLE… ADD PARTITION command adds a partition to an existing partitioned table. There is no upper limit to the number of defined partitions in a partitioned table.

New partitions must be of the same type (LIST, RANGE or HASH) as existing partitions. The new partition rules must reference the same column specified in the partitioning rules that define the existing partition(s).

You cannot use the ALTER TABLE… ADD PARTITION statement to add a partition to a table with a MAXVALUE or DEFAULT rule. Note that you can alternatively use the ALTER TABLE… SPLIT PARTITION statement to split an existing partition, effectively increasing the number of partitions in a table.

RANGE partitions must be specified in ascending order. You cannot add a new partition that precedes existing partitions in a RANGE partitioned table.

Include the TABLESPACE clause to specify the tablespace in which the new partition will reside. If you do not specify a tablespace, the partition will reside in the default tablespace.

If the table is indexed, the index will be created on the new partition.

To use the ALTER TABLE... ADD PARTITION command you must be the table owner, or have superuser (or administrative) privileges.

***Parameters***

table\_name

The name (optionally schema-qualified) of the partitioned table.

partition\_name

The name of the partition to be created. Partition names must be unique amongst all partitions and subpartitions, and must follow the naming conventions for object identifiers.

subpartition\_name

The name of the subpartition to be created. Subpartition names must be unique amongst all partitions and subpartitions, and must follow the naming conventions for object identifiers.

(*value*[, *value*]...)

Use *value* to specify a quoted literal value (or comma-delimited list of literal values) by which rows will be distributed into partitions. Each partitioning rule must specify at least one *value*, but there is no limit placed on the number of values specified within a rule. *value* may also be NULL, DEFAULT (if specifying a LIST partition), or MAXVALUE (if specifying a RANGE partition).

For information about creating a DEFAULT or MAXVALUE partition, see [Section 3.6](#_Handling_Stray_Values_11).

tablespace\_name

The name of the tablespace in which a partition or subpartition resides.

#### Example - Adding a Partition to a LIST Partitioned Table

The example that follows adds a partition to the list-partitioned sales table. The table was created using the command:

CREATE TABLE sales

(

dept\_no number,

part\_no varchar2,

country varchar2(20),

date date,

amount number

)

PARTITION BY LIST(country)

(

PARTITION europe VALUES('FRANCE', 'ITALY'),

PARTITION asia VALUES('INDIA', 'PAKISTAN'),

PARTITION americas VALUES('US', 'CANADA')

);

The table contains three partitions (americas, asia, and europe) :

acctg=# SELECT partition\_name, high\_value FROM ALL\_TAB\_PARTITIONS;

partition\_name | high\_value

----------------+---------------------

americas | 'US', 'CANADA'

asia | 'INDIA', 'PAKISTAN'

europe | 'FRANCE', 'ITALY'

(3 rows)

The following command adds a partition named east\_asia to the sales table:

ALTER TABLE sales ADD PARTITION east\_asia   
 VALUES ('CHINA', 'KOREA');

After invoking the command, the table includes the east\_asia partition:

acctg=# SELECT partition\_name, high\_value FROM ALL\_TAB\_PARTITIONS;

partition\_name | high\_value

----------------+---------------------

east\_asia | 'CHINA', 'KOREA'

americas | 'US', 'CANADA'

asia | 'INDIA', 'PAKISTAN'

europe | 'FRANCE', 'ITALY'

(4 rows)

#### Example - Adding a Partition to a RANGE Partitioned Table

The example that follows adds a partition to a range-partitioned table named sales:

CREATE TABLE sales

(

dept\_no number,

part\_no varchar2,

country varchar2(20),

date date,

amount number

)

PARTITION BY RANGE(date)

(

PARTITION q1\_2012   
 VALUES LESS THAN('2012-Apr-01'),

PARTITION q2\_2012   
 VALUES LESS THAN('2012-Jul-01'),

PARTITION q3\_2012   
 VALUES LESS THAN('2012-Oct-01'),

PARTITION q4\_2012   
 VALUES LESS THAN('2013-Jan-01')

);

The table contains four partitions (q1\_2012, q2\_2012, q3\_2012, and q4\_2012):

acctg=# SELECT partition\_name, high\_value FROM ALL\_TAB\_PARTITIONS;

partition\_name | high\_value

----------------+---------------

q4\_2012 | '2013-Jan-01'

q3\_2012 | '2012-Oct-01'

q2\_2012 | '2012-Jul-01'

q1\_2012 | '2012-Apr-01'

(4 rows)

The following command adds a partition named q1\_2013 to the sales table:

ALTER TABLE sales ADD PARTITION q1\_2013   
 VALUES LESS THAN('01-APR-2013');

After invoking the command, the table includes the q1\_2013 partition:

acctg=# SELECT partition\_name, high\_value FROM ALL\_TAB\_PARTITIONS;

partition\_name | high\_value

----------------+---------------

q1\_2012 | '2012-Apr-01'

q2\_2012 | '2012-Jul-01'

q3\_2012 | '2012-Oct-01'

q4\_2012 | '2013-Jan-01'

q1\_2013 | '01-APR-2013'

(5 rows)

### ALTER TABLE… ADD SUBPARTITION

The ALTER TABLE… ADD SUBPARTITION command adds a subpartition to an existing subpartitioned partition. The syntax is:

ALTER TABLE *table*\_*name* MODIFY PARTITION *partition*\_*name*   
 ADD SUBPARTITION *subpartition*\_*definition;*

Where *subpartition\_definition* is:

{*list\_subpartition* | *range\_subpartition*}

and *list*\_*subpartition* is:

SUBPARTITION [*subpartition\_name*]

VALUES (*value*[, *value*]...)

[TABLESPACE *tablespace\_name*]

and *range\_subpartition* is:

SUBPARTITION [*subpartition\_name*]

VALUES LESS THAN (*value*[, *value*]...)

[TABLESPACE *tablespace\_name*]

**Description**

The ALTER TABLE… ADD SUBPARTITION command adds a subpartition to an existing partition; the partition must already be subpartitioned. There is no upper limit to the number of defined subpartitions.

New subpartitions must be of the same type (LIST, RANGE or HASH) as existing subpartitions. The new subpartition rules must reference the same column specified in the subpartitioning rules that define the existing subpartition(s).

You cannot use the ALTER TABLE… ADD SUBPARTITION statement to add a subpartition to a table with a MAXVALUE or DEFAULT rule , but you can split an existing subpartition with the ALTER TABLE… SPLIT SUBPARTITION statement, effectively adding a subpartition to a table.

You cannot add a new subpartition that precedes existing subpartitions in a range subpartitioned table; range subpartitions must be specified in ascending order.

Include the TABLESPACE clause to specify the tablespace in which the subpartition will reside. If you do not specify a tablespace, the subpartition will be created in the default tablespace.

If the table is indexed, the index will be created on the new subpartition.

To use the ALTER TABLE... ADD SUBPARTITION command you must be the table owner, or have superuser (or administrative) privileges.

**Parameters**

table\_name

The name (optionally schema-qualified) of the partitioned table in which the subpartition will reside.

partition\_name

The name of the partition in which the new subpartition will reside.

subpartition\_name

The name of the subpartition to be created. Subpartition names must be unique amongst all partitions and subpartitions, and must follow the naming conventions for object identifiers.

(*value*[, *value*]...)

Use value to specify a quoted literal value (or comma-delimited list of literal values) by which table entries will be grouped into partitions. Each partitioning rule must specify at least one value, but there is no limit placed on the number of values specified within a rule. value may also be NULL, DEFAULT (if specifying a LIST partition), or MAXVALUE (if specifying a RANGE partition).

For information about creating a DEFAULT or MAXVALUE partition, see [Section 3.6](#_Handling_Stray_Values_11).

tablespace\_name

The name of the tablespace in which the subpartition resides.

#### Example - Adding a Subpartition to a LIST-RANGE Partitioned Table

The following example adds a RANGE subpartition to the list-partitioned sales table. The sales table was created with the command:

CREATE TABLE sales

(

dept\_no number,

part\_no varchar2,

country varchar2(20),

date date,

amount number

)

PARTITION BY LIST(country)

SUBPARTITION BY RANGE(date)

(

PARTITION europe VALUES('FRANCE', 'ITALY')

(

SUBPARTITION europe\_2011   
 VALUES LESS THAN('2012-Jan-01'),

SUBPARTITION europe\_2012   
 VALUES LESS THAN('2013-Jan-01')

),

PARTITION asia VALUES('INDIA', 'PAKISTAN')

(

SUBPARTITION asia\_2011   
 VALUES LESS THAN('2012-Jan-01'),

SUBPARTITION asia\_2012   
 VALUES LESS THAN('2013-Jan-01')

),

PARTITION americas VALUES('US', 'CANADA')

(

SUBPARTITION americas\_2011   
 VALUES LESS THAN('2012-Jan-01'),

SUBPARTITION americas\_2012   
 VALUES LESS THAN('2013-Jan-01')

)

);

The sales table has three partitions, named europe, asia, and americas. Each partition has two range-defined subpartitions:

acctg=# SELECT partition\_name, subpartition\_name, high\_value FROM ALL\_TAB\_SUBPARTITIONS;

partition\_name | subpartition\_name | high\_value

----------------+-------------------+---------------

europe | europe\_2011 | '2012-Jan-01'

europe | europe\_2012 | '2013-Jan-01'

asia | asia\_2011 | '2012-Jan-01'

asia | asia\_2012 | '2013-Jan-01'

americas | americas\_2011 | '2012-Jan-01'

americas | americas\_2012 | '2013-Jan-01'

(6 rows)

The following command adds a subpartition named europe\_2013:

ALTER TABLE sales MODIFY PARTITION europe   
 ADD SUBPARTITION europe\_2013   
 VALUES LESS THAN('2014-Jan-01');

After invoking the command, the table includes a subpartition named europe\_2013:

acctg=# SELECT partition\_name, subpartition\_name, high\_value FROM ALL\_TAB\_SUBPARTITIONS;

partition\_name | subpartition\_name | high\_value

----------------+-------------------+---------------

europe | europe\_2011 | '2012-Jan-01'

europe | europe\_2012 | '2013-Jan-01'

europe | europe\_2013 | '2014-Jan-01'

asia | asia\_2011 | '2012-Jan-01'

asia | asia\_2012 | '2013-Jan-01'

americas | americas\_2011 | '2012-Jan-01'

americas | americas\_2012 | '2013-Jan-01'

(7 rows)

Note that when adding a new range subpartition, the subpartitioning rules must specify a range that falls *after* any existing subpartitions.

#### Example - Adding a Subpartition to a RANGE-LIST Partitioned Table

The following example adds a LIST subpartition to the RANGE partitioned sales table. The sales table was created with the command:

CREATE TABLE sales

(

dept\_no number,

part\_no varchar2,

country varchar2(20),

date date,

amount number

)

PARTITION BY RANGE(date)  
 SUBPARTITION BY LIST (country)  
 (  
 PARTITION first\_half\_2012 VALUES LESS THAN('01-JUL-2012')  
 (

SUBPARTITION europe VALUES ('ITALY', 'FRANCE'),

SUBPARTITION americas VALUES ('US', 'CANADA')

),

PARTITION second\_half\_2012 VALUES LESS THAN('01-JAN-2013')   
 (

SUBPARTITION asia VALUES ('INDIA', 'PAKISTAN')

)  
 );

After executing the above command, the sales table will have two partitions, named first\_half\_2012 and second\_half\_2012. The first\_half\_2012 partition has two subpartitions, named europe and americas, and the second\_half\_2012 partition has one partition, named asia:

acctg=# SELECT partition\_name, subpartition\_name, high\_value FROM ALL\_TAB\_SUBPARTITIONS;

partition\_name | subpartition\_name | high\_value

------------------+-------------------+---------------------

first\_half\_2012 | europe | 'ITALY', 'FRANCE'

first\_half\_2012 | americas | 'US', 'CANADA'

second\_half\_2012 | asia | 'INDIA', 'PAKISTAN'

(3 rows)

The following command adds a subpartition to the second\_half\_2012 partition, named east\_asia:

ALTER TABLE sales MODIFY PARTITION second\_half\_2012   
 ADD SUBPARTITION east\_asia VALUES ('CHINA')*;*

After invoking the command, the table includes a subpartition named east\_asia:

acctg=# SELECT partition\_name, subpartition\_name, high\_value FROM ALL\_TAB\_SUBPARTITIONS;

partition\_name | subpartition\_name | high\_value

------------------+-------------------+---------------------

first\_half\_2012 | europe | 'ITALY', 'FRANCE'

first\_half\_2012 | americas | 'US', 'CANADA'

second\_half\_2012 | asia | 'INDIA', 'PAKISTAN'

second\_half\_2012 | east\_asia | 'CHINA'

(4 rows)

### ALTER TABLE...SPLIT PARTITION

Use the ALTER TABLE… SPLIT PARTITION command to divide a single partition into two partitions, and redistribute the partition's contents between the new partitions. The command syntax comes in two forms.

The first form splits a RANGE partition into two partitions:

ALTER TABLE *table*\_*name* SPLIT PARTITION *partition*\_*name* AT (*range*\_*part*\_*value*)  
 INTO   
 (  
 PARTITION *new\_part1* [TABLESPACE *tablespace\_name*]*,* PARTITION *new\_part2* [TABLESPACE *tablespace\_name*] );

The second form splits a LIST partition into two partitions:

ALTER TABLE *table*\_*name* SPLIT PARTITION *partition*\_*name* VALUES (*value*[, *value*]...)   
 INTO   
 (  
 PARTITION *new\_part1*  [TABLESPACE *tablespace\_name*]*,*  PARTITION *new\_part2* [TABLESPACE *tablespace\_name*] );

**Description**

The ALTER TABLE...SPLIT PARTITION command adds a partition to an existing LIST or RANGE partitioned table. Please note that the ALTER TABLE… SPLIT PARTITION command cannot add a partition to a HASH partitioned table. There is no upper limit to the number of partitions that a table may have.

When you execute an ALTER TABLE...SPLIT PARTITION command, Advanced Server creates two new partitions, and redistributes the content of the old partition between them (as constrained by the partitioning rules).

Include the TABLESPACE clause to specify the tablespace in which a partition will reside. If you do not specify a tablespace, the partition will reside in the default tablespace.

If the table is indexed, the index will be created on the new partition.

To use the ALTER TABLE... SPLIT PARTITION command you must be the table owner, or have superuser (or administrative) privileges.

**Parameters**

table\_name

The name (optionally schema-qualified) of the partitioned table.

partition\_name

The name of the partition that is being split.

new\_part1

The name of the first new partition to be created. Partition names must be unique amongst all partitions and subpartitions, and must follow the naming conventions for object identifiers.

*new\_part1* will receive the rows that meet the subpartitioning constraints specified in the ALTER TABLE… SPLIT SUBPARTITION command.

new\_part2

The name of the second new partition to be created. Partition names must be unique amongst all partitions and subpartitions, and must follow the naming conventions for object identifiers.

*new\_part2* will receive the rows are not directed to *new\_part1* by the partitioning constraints specified in the ALTER TABLE… SPLIT PARTITION command.

*range\_part\_value*

Use *range\_part\_value* to specify the boundary rules by which to create the new partition. The partitioning rule must contain at least one column of a data type that has two operators (i.e., a greater-than-or-equal to operator, and a less-than operator). Range boundaries are evaluated against a LESS THAN clause and are non-inclusive; a date boundary of January 1, 2010 will include only those date values that fall on or before December 31, 2009.

(*value*[, *value*]...)

Use *value* to specify a quoted literal value (or comma-delimited list of literal values) by which rows will be distributed into partitions. Each partitioning rule must specify at least one value, but there is no limit placed on the number of values specified within a rule.

For information about creating a DEFAULT or MAXVALUE partition, see [Section 3.6](#_Handling_Stray_Values_17).

tablespace\_name

The name of the tablespace in which the partition or subpartition resides.

#### Example - Splitting a LIST Partition

Our example will divide one of the partitions in the list-partitioned sales table into two new partitions, and redistribute the contents of the partition between them. The sales table is created with the statement:

CREATE TABLE sales

(

dept\_no number,

part\_no varchar2,

country varchar2(20),

date date,

amount number

)

PARTITION BY LIST(country)

(

PARTITION europe VALUES('FRANCE', 'ITALY'),

PARTITION asia VALUES('INDIA', 'PAKISTAN'),

PARTITION americas VALUES('US', 'CANADA')

);

The table definition creates three partitions (europe, asia, and americas). The following command adds rows to each partition:

INSERT INTO sales VALUES

(10, '4519b', 'FRANCE', '17-Jan-2012', '45000'),

(20, '3788a', 'INDIA', '01-Mar-2012', '75000'),

(40, '9519b', 'US', '12-Apr-2012', '145000'),

(20, '3788a', 'PAKISTAN', '04-Jun-2012', '37500'),

(40, '4577b', 'US', '11-Nov-2012', '25000'),

(30, '7588b', 'CANADA', '14-Dec-2012', '50000'),

(30, '9519b', 'CANADA', '01-Feb-2012', '75000'),

(30, '4519b', 'CANADA', '08-Apr-2012', '120000'),

(40, '3788a', 'US', '12-May-2012', '4950'),

(10, '9519b', 'ITALY', '07-Jul-2012', '15000'),

(10, '9519a', 'FRANCE', '18-Aug-2012', '650000'),

(10, '9519b', 'FRANCE', '18-Aug-2012', '650000'),

(20, '3788b', 'INDIA', '21-Sept-2012', '5090'),

(40, '4788a', 'US', '23-Sept-2012', '4950'),

(40, '4788b', 'US', '09-Oct-2012', '15000'),

(20, '4519a', 'INDIA', '18-Oct-2012', '650000'),

(20, '4519b', 'INDIA', '2-Dec-2012', '5090');

The rows are distributed amongst the partitions:

acctg=# SELECT tableoid::regclass, \* FROM sales;

tableoid | dept\_no | part\_no | country | date | amount

----------------+---------+---------+----------+--------------------+-------

sales\_europe | 10 | 4519b | FRANCE | 17-JAN-12 00:00:00 | 45000

sales\_europe | 10 | 9519b | ITALY | 07-JUL-12 00:00:00 | 15000

sales\_europe | 10 | 9519a | FRANCE | 18-AUG-12 00:00:00 | 650000

sales\_europe | 10 | 9519b | FRANCE | 18-AUG-12 00:00:00 | 650000

sales\_asia | 20 | 3788a | INDIA | 01-MAR-12 00:00:00 | 75000

sales\_asia | 20 | 3788a | PAKISTAN | 04-JUN-12 00:00:00 | 37500

sales\_asia | 20 | 3788b | INDIA | 21-SEP-12 00:00:00 | 5090

sales\_asia | 20 | 4519a | INDIA | 18-OCT-12 00:00:00 | 650000

sales\_asia | 20 | 4519b | INDIA | 02-DEC-12 00:00:00 | 5090

sales\_americas | 40 | 9519b | US | 12-APR-12 00:00:00 | 145000

sales\_americas | 40 | 4577b | US | 11-NOV-12 00:00:00 | 25000

sales\_americas | 30 | 7588b | CANADA | 14-DEC-12 00:00:00 | 50000

sales\_americas | 30 | 9519b | CANADA | 01-FEB-12 00:00:00 | 75000

sales\_americas | 30 | 4519b | CANADA | 08-APR-12 00:00:00 | 120000

sales\_americas | 40 | 3788a | US | 12-MAY-12 00:00:00 | 4950

sales\_americas | 40 | 4788a | US | 23-SEP-12 00:00:00 | 4950

sales\_americas | 40 | 4788b | US | 09-OCT-12 00:00:00 | 15000

(17 rows)

The following command splits the americas partition into two partitions named us and canada:

ALTER TABLE sales SPLIT PARTITION americas   
 VALUES ('US')  
 INTO (PARTITION us, PARTITION canada);

A SELECT statement confirms that the rows have been redistributed across the correct partitions:

acctg=# SELECT tableoid::regclass, \* FROM sales;

tableoid | dept\_no | part\_no | country | date | amount

--------------+---------+---------+----------+--------------------+--------

sales\_europe | 10 | 4519b | FRANCE | 17-JAN-12 00:00:00 | 45000

sales\_europe | 10 | 9519b | ITALY | 07-JUL-12 00:00:00 | 15000

sales\_europe | 10 | 9519a | FRANCE | 18-AUG-12 00:00:00 | 650000

sales\_europe | 10 | 9519b | FRANCE | 18-AUG-12 00:00:00 | 650000

sales\_asia | 20 | 3788a | INDIA | 01-MAR-12 00:00:00 | 75000

sales\_asia | 20 | 3788a | PAKISTAN | 04-JUN-12 00:00:00 | 37500

sales\_asia | 20 | 3788b | INDIA | 21-SEP-12 00:00:00 | 5090

sales\_asia | 20 | 4519a | INDIA | 18-OCT-12 00:00:00 | 650000

sales\_asia | 20 | 4519b | INDIA | 02-DEC-12 00:00:00 | 5090

sales\_us | 40 | 9519b | US | 12-APR-12 00:00:00 | 145000

sales\_us | 40 | 4577b | US | 11-NOV-12 00:00:00 | 25000

sales\_us | 40 | 3788a | US | 12-MAY-12 00:00:00 | 4950

sales\_us | 40 | 4788a | US | 23-SEP-12 00:00:00 | 4950

sales\_us | 40 | 4788b | US | 09-OCT-12 00:00:00 | 15000

sales\_canada | 30 | 7588b | CANADA | 14-DEC-12 00:00:00 | 50000

sales\_canada | 30 | 9519b | CANADA | 01-FEB-12 00:00:00 | 75000

sales\_canada | 30 | 4519b | CANADA | 08-APR-12 00:00:00 | 120000

(17 rows)

#### Example - Splitting a RANGE Partition

This example divides the q4\_2012 partition (of the range-partitioned sales table) into two partitions, and redistribute the partition's contents. Use the following command to create the sales table:

CREATE TABLE sales

(

dept\_no number,

part\_no varchar2,

country varchar2(20),

date date,

amount number

)

PARTITION BY RANGE(date)

(

PARTITION q1\_2012   
 VALUES LESS THAN('2012-Apr-01'),

PARTITION q2\_2012   
 VALUES LESS THAN('2012-Jul-01'),

PARTITION q3\_2012   
 VALUES LESS THAN('2012-Oct-01'),

PARTITION q4\_2012   
 VALUES LESS THAN('2013-Jan-01')

);

The table definition creates four partitions (q1\_2012, q2\_2012, q3\_2012, and q4\_2012). The following command adds rows to each partition:

INSERT INTO sales VALUES

(10, '4519b', 'FRANCE', '17-Jan-2012', '45000'),

(20, '3788a', 'INDIA', '01-Mar-2012', '75000'),

(40, '9519b', 'US', '12-Apr-2012', '145000'),

(20, '3788a', 'PAKISTAN', '04-Jun-2012', '37500'),

(40, '4577b', 'US', '11-Nov-2012', '25000'),

(30, '7588b', 'CANADA', '14-Dec-2012', '50000'),

(30, '9519b', 'CANADA', '01-Feb-2012', '75000'),

(30, '4519b', 'CANADA', '08-Apr-2012', '120000'),

(40, '3788a', 'US', '12-May-2012', '4950'),

(10, '9519b', 'ITALY', '07-Jul-2012', '15000'),

(10, '9519a', 'FRANCE', '18-Aug-2012', '650000'),

(10, '9519b', 'FRANCE', '18-Aug-2012', '650000'),

(20, '3788b', 'INDIA', '21-Sept-2012', '5090'),

(40, '4788a', 'US', '23-Sept-2012', '4950'),

(40, '4788b', 'US', '09-Oct-2012', '15000'),

(20, '4519a', 'INDIA', '18-Oct-2012', '650000'),

(20, '4519b', 'INDIA', '2-Dec-2012', '5090');

A SELECT statement confirms that the rows are distributed amongst the partitions as expected:

acctg=# SELECT tableoid::regclass, \* FROM sales;

tableoid | dept\_no | part\_no | country | date | amount

---------------+---------+---------+----------+--------------------+--------

sales\_q1\_2012 | 10 | 4519b | FRANCE | 17-JAN-12 00:00:00 | 45000

sales\_q1\_2012 | 20 | 3788a | INDIA | 01-MAR-12 00:00:00 | 75000

sales\_q1\_2012 | 30 | 9519b | CANADA | 01-FEB-12 00:00:00 | 75000

sales\_q2\_2012 | 40 | 9519b | US | 12-APR-12 00:00:00 | 145000

sales\_q2\_2012 | 20 | 3788a | PAKISTAN | 04-JUN-12 00:00:00 | 37500

sales\_q2\_2012 | 30 | 4519b | CANADA | 08-APR-12 00:00:00 | 120000

sales\_q2\_2012 | 40 | 3788a | US | 12-MAY-12 00:00:00 | 4950

sales\_q3\_2012 | 10 | 9519b | ITALY | 07-JUL-12 00:00:00 | 15000

sales\_q3\_2012 | 10 | 9519a | FRANCE | 18-AUG-12 00:00:00 | 650000

sales\_q3\_2012 | 10 | 9519b | FRANCE | 18-AUG-12 00:00:00 | 650000

sales\_q3\_2012 | 20 | 3788b | INDIA | 21-SEP-12 00:00:00 | 5090

sales\_q3\_2012 | 40 | 4788a | US | 23-SEP-12 00:00:00 | 4950

sales\_q4\_2012 | 40 | 4577b | US | 11-NOV-12 00:00:00 | 25000

sales\_q4\_2012 | 30 | 7588b | CANADA | 14-DEC-12 00:00:00 | 50000

sales\_q4\_2012 | 40 | 4788b | US | 09-OCT-12 00:00:00 | 15000

sales\_q4\_2012 | 20 | 4519a | INDIA | 18-OCT-12 00:00:00 | 650000

sales\_q4\_2012 | 20 | 4519b | INDIA | 02-DEC-12 00:00:00 | 5090

(17 rows)

The following command splits the q4\_2012 partition into two partitions named q4\_2012\_p1 and q4\_2012\_p2:

ALTER TABLE sales SPLIT PARTITION q4\_2012  
 AT ('15-Nov-2012')  
 INTO   
 (  
 PARTITION q4\_2012\_p1,  
 PARTITION q4\_2012\_p2  
 );

A SELECT statement confirms that the rows have been redistributed across the new partitions:

acctg=# SELECT tableoid::regclass, \* FROM sales;

tableoid | dept\_no | part\_no | country | date |amount

------------------+---------+---------+----------+--------------------+------

sales\_q1\_2012 | 10 | 4519b | FRANCE | 17-JAN-12 00:00:00 | 45000

sales\_q1\_2012 | 20 | 3788a | INDIA | 01-MAR-12 00:00:00 | 75000

sales\_q1\_2012 | 30 | 9519b | CANADA | 01-FEB-12 00:00:00 | 75000

sales\_q2\_2012 | 40 | 9519b | US | 12-APR-12 00:00:00 |145000

sales\_q2\_2012 | 20 | 3788a | PAKISTAN | 04-JUN-12 00:00:00 | 37500

sales\_q2\_2012 | 30 | 4519b | CANADA | 08-APR-12 00:00:00 |120000

sales\_q2\_2012 | 40 | 3788a | US | 12-MAY-12 00:00:00 | 4950

sales\_q3\_2012 | 10 | 9519b | ITALY | 07-JUL-12 00:00:00 | 15000

sales\_q3\_2012 | 10 | 9519a | FRANCE | 18-AUG-12 00:00:00 |650000

sales\_q3\_2012 | 10 | 9519b | FRANCE | 18-AUG-12 00:00:00 |650000

sales\_q3\_2012 | 20 | 3788b | INDIA | 21-SEP-12 00:00:00 | 5090

sales\_q3\_2012 | 40 | 4788a | US | 23-SEP-12 00:00:00 | 4950

sales\_q4\_2012\_p1 | 40 | 4577b | US | 11-NOV-12 00:00:00 | 25000

sales\_q4\_2012\_p1 | 40 | 4788b | US | 09-OCT-12 00:00:00 | 15000

sales\_q4\_2012\_p1 | 20 | 4519a | INDIA | 18-OCT-12 00:00:00 |650000

sales\_q4\_2012\_p2 | 30 | 7588b | CANADA | 14-DEC-12 00:00:00 | 50000

sales\_q4\_2012\_p2 | 20 | 4519b | INDIA | 02-DEC-12 00:00:00 | 5090

(17 rows)

### ALTER TABLE...SPLIT SUBPARTITION

Use the ALTER TABLE… SPLIT SUBPARTITION command to divide a single subpartition into two subpartitions, and redistribute the subpartition's contents. The command comes in two variations.

The first variation splits a range subpartition into two subpartitions:

ALTER TABLE *table*\_*name* SPLIT SUBPARTITION *subpartition*\_*name*  
 AT (*range*\_*part*\_*value*)  
 INTO   
 (  
 SUBPARTITION *new*\_*subpart1*   
 [TABLESPACE *tablespace*\_*name*],  
 SUBPARTITION *new*\_*subpart2*   
 [TABLESPACE *tablespace*\_*name*]  
 );

The second variation splits a list subpartition into two subpartitions:

ALTER TABLE *table*\_*name* SPLIT SUBPARTITION *subpartition*\_*name* VALUES (*value*[, *value*]...)   
 INTO   
 (  
 SUBPARTITION *new*\_*subpart1*   
 [TABLESPACE *tablespace*\_*name*],  
 SUBPARTITION *new*\_*subpart2*   
 [TABLESPACE *tablespace*\_*name*]  
 );

**Description**

The ALTER TABLE...SPLIT SUBPARTITION command adds a subpartition to an existing subpartitioned table. There is no upper limit to the number of defined subpartitions. When you execute an ALTER TABLE...SPLIT SUBPARTITION command, Advanced Server creates two new subpartitions, moving any rows that contain values that are constrained by the specified subpartition rules into *new\_subpart1*, and any remaining rows into *new\_subpart2*.

The new subpartition rules must reference the column specified in the rules that define the existing subpartition(s).

Include the TABLESPACE clause to specify a tablespace in which a new subpartition will reside. If you do not specify a tablespace, the subpartition will be created in the default tablespace.

If the table is indexed, the index will be created on the new subpartition.

To use the ALTER TABLE... SPLIT SUBPARTITION command you must be the table owner, or have superuser (or administrative) privileges.

**Parameters**

table\_name

The name (optionally schema-qualified) of the partitioned table.

subpartition\_name

The name of the subpartition that is being split.

new\_subpart1

The name of the first new subpartition to be created. Subpartition names must be unique amongst all partitions and subpartitions, and must follow the naming conventions for object identifiers.

*new\_subpart1* will receive the rows that meet the subpartitioning constraints specified in the ALTER TABLE… SPLIT SUBPARTITION command.

new\_subpart2

The name of the second new subpartition to be created. Subpartition names must be unique amongst all partitions and subpartitions, and must follow the naming conventions for object identifiers.

*new\_subpart2* will receive the rows are not directed to *new\_subpart1* by the subpartitioning constraints specified in the ALTER TABLE… SPLIT SUBPARTITION command.

(*value*[, *value*]...)

Use *value* to specify a quoted literal value (or comma-delimited list of literal values) by which table entries will be grouped into partitions. Each partitioning rule must specify at least one value, but there is no limit placed on the number of values specified within a rule. *value* may also be NULL, DEFAULT (if specifying a LIST subpartition), or MAXVALUE (if specifying a RANGE subpartition).

For information about creating a DEFAULT or MAXVALUE partition, see [Section 3.6](#_Handling_Stray_Values_11).

tablespace\_name

The name of the tablespace in which the partition or subpartition resides.

#### Example - Splitting a LIST Subpartition

The following example splits a list subpartition, redistributing the subpartition's contents between two new subpartitions. The sample table (sales) was created with the command:

CREATE TABLE sales

(

dept\_no number,

part\_no varchar2,

country varchar2(20),

date date,

amount number

)

PARTITION BY RANGE(date)  
 SUBPARTITION BY LIST (country)  
 (  
 PARTITION first\_half\_2012 VALUES LESS THAN('01-JUL-2012')  
 (

SUBPARTITION p1\_europe VALUES ('ITALY', 'FRANCE'),

SUBPARTITION p1\_americas VALUES ('US', 'CANADA')

),

PARTITION second\_half\_2012 VALUES LESS THAN('01-JAN-2013')   
 (

SUBPARTITION p2\_europe VALUES ('ITALY', 'FRANCE'),

SUBPARTITION p2\_americas VALUES ('US', 'CANADA')

)  
 );

The sales table has two partitions, named first\_half\_2012, and second\_half\_2012. Each partition has two range-defined subpartitions that distribute the partition's contents into subpartitions based on the value of the country column:

acctg=# SELECT partition\_name, subpartition\_name, high\_value FROM ALL\_TAB\_SUBPARTITIONS;

partition\_name | subpartition\_name | high\_value

------------------+-------------------+-------------------

second\_half\_2012 | p2\_europe | 'ITALY', 'FRANCE'

first\_half\_2012 | p1\_europe | 'ITALY', 'FRANCE'

second\_half\_2012 | p2\_americas | 'US', 'CANADA'

first\_half\_2012 | p1\_americas | 'US', 'CANADA'

(4 rows)

The following command adds rows to each subpartition:

INSERT INTO sales VALUES

(10, '4519b', 'FRANCE', '17-Jan-2012', '45000'),

(40, '9519b', 'US', '12-Apr-2012', '145000'),

(40, '4577b', 'US', '11-Nov-2012', '25000'),

(30, '7588b', 'CANADA', '14-Dec-2012', '50000'),

(30, '9519b', 'CANADA', '01-Feb-2012', '75000'),

(30, '4519b', 'CANADA', '08-Apr-2012', '120000'),

(40, '3788a', 'US', '12-May-2012', '4950'),

(10, '9519b', 'ITALY', '07-Jul-2012', '15000'),

(10, '9519a', 'FRANCE', '18-Aug-2012', '650000'),

(10, '9519b', 'FRANCE', '18-Aug-2012', '650000'),

(40, '4788a', 'US', '23-Sept-2012', '4950'),

(40, '4788b', 'US', '09-Oct-2012', '15000');

A SELECT statement confirms that the rows are correctly distributed amongst the subpartitions:

acctg=# SELECT tableoid::regclass, \* FROM sales;

tableoid | dept\_no | part\_no | country| date |amount

-------------------+---------+---------+--------+--------------------+------

sales\_p1\_europe | 10 | 4519b | FRANCE | 17-JAN-12 00:00:00 | 45000

sales\_p1\_europe | 10 | 4519b | FRANCE | 17-JAN-12 00:00:00 | 45000

sales\_p1\_americas | 40 | 9519b | US | 12-APR-12 00:00:00 | 145000

sales\_p1\_americas | 30 | 9519b | CANADA | 01-FEB-12 00:00:00 | 75000

sales\_p1\_americas | 30 | 4519b | CANADA | 08-APR-12 00:00:00 | 120000

sales\_p1\_americas | 40 | 3788a | US | 12-MAY-12 00:00:00 | 4950

sales\_p2\_europe | 10 | 9519b | ITALY | 07-JUL-12 00:00:00 | 15000

sales\_p2\_europe | 10 | 9519a | FRANCE | 18-AUG-12 00:00:00 | 650000

sales\_p2\_europe | 10 | 9519b | FRANCE | 18-AUG-12 00:00:00 | 650000

sales\_p2\_americas | 40 | 4577b | US | 11-NOV-12 00:00:00 | 25000

sales\_p2\_americas | 30 | 7588b | CANADA | 14-DEC-12 00:00:00 | 50000

sales\_p2\_americas | 40 | 4788a | US | 23-SEP-12 00:00:00 | 4950

sales\_p2\_americas | 40 | 4788b | US | 09-OCT-12 00:00:00 | 15000

(13 rows)

The following command splits the p2\_americas subpartition into two new subpartitions, and redistributes the contents:

ALTER TABLE sales SPLIT SUBPARTITION p2\_americas VALUES ('US')

INTO   
 (  
 SUBPARTITION p2\_us,  
 SUBPARTITION p2\_canada  
 );

After invoking the command, the p2\_americas subpartition has been deleted; in it's place, the server has created two new subpartitions (p2\_us and p2\_canada):

acctg=# SELECT partition\_name, subpartition\_name, high\_value FROM ALL\_TAB\_SUBPARTITIONS;

partition\_name | subpartition\_name | high\_value

------------------+-------------------+-------------------

first\_half\_2012 | p1\_europe | 'ITALY', 'FRANCE'

first\_half\_2012 | p1\_americas | 'US', 'CANADA'

second\_half\_2012 | p2\_europe | 'ITALY', 'FRANCE'

second\_half\_2012 | p2\_canada | 'CANADA'

second\_half\_2012 | p2\_us | 'US'

(5 rows)

Querying the sales table demonstrates that the content of the p2\_americas subpartition has been redistributed:

acctg=# SELECT tableoid::regclass, \* FROM sales;

tableoid | dept\_no | part\_no | country | date |amount

-------------------+---------+---------+---------+--------------------+------

sales\_p1\_europe | 10 | 4519b | FRANCE | 17-JAN-12 00:00:00 | 45000

sales\_p1\_europe | 10 | 4519b | FRANCE | 17-JAN-12 00:00:00 | 45000

sales\_p1\_americas | 40 | 9519b | US | 12-APR-12 00:00:00 |145000

sales\_p1\_americas | 30 | 9519b | CANADA | 01-FEB-12 00:00:00 | 75000

sales\_p1\_americas | 30 | 4519b | CANADA | 08-APR-12 00:00:00 |120000

sales\_p1\_americas | 40 | 3788a | US | 12-MAY-12 00:00:00 | 4950

sales\_p2\_europe | 10 | 9519b | ITALY | 07-JUL-12 00:00:00 | 15000

sales\_p2\_europe | 10 | 9519a | FRANCE | 18-AUG-12 00:00:00 |650000

sales\_p2\_europe | 10 | 9519b | FRANCE | 18-AUG-12 00:00:00 |650000

sales\_p2\_us | 40 | 4577b | US | 11-NOV-12 00:00:00 | 25000

sales\_p2\_us | 40 | 4788a | US | 23-SEP-12 00:00:00 | 4950

sales\_p2\_us | 40 | 4788b | US | 09-OCT-12 00:00:00 | 15000

sales\_p2\_canada | 30 | 7588b | CANADA | 14-DEC-12 00:00:00 | 50000

(13 rows)

#### Example - Splitting a RANGE Subpartition

The following example splits a range subpartition, redistributing the subpartition's contents between two new subpartitions. The sample table (sales) was created with the command:

CREATE TABLE sales

(

dept\_no number,

part\_no varchar2,

country varchar2(20),

date date,

amount number

)

PARTITION BY LIST(country)

SUBPARTITION BY RANGE(date)

(

PARTITION europe VALUES('FRANCE', 'ITALY')

(

SUBPARTITION europe\_2011   
 VALUES LESS THAN('2012-Jan-01'),

SUBPARTITION europe\_2012   
 VALUES LESS THAN('2013-Jan-01')

),

PARTITION asia VALUES('INDIA', 'PAKISTAN')

(

SUBPARTITION asia\_2011   
 VALUES LESS THAN('2012-Jan-01'),

SUBPARTITION asia\_2012   
 VALUES LESS THAN('2013-Jan-01')

),

PARTITION americas VALUES('US', 'CANADA')

(

SUBPARTITION americas\_2011   
 VALUES LESS THAN('2012-Jan-01'),

SUBPARTITION americas\_2012   
 VALUES LESS THAN('2013-Jan-01')

)

);

The sales table has three partitions (europe, asia, and americas). Each partition has two range-defined subpartitions that sort the partitions contents into subpartitions by the value of the date column:

acctg=# SELECT partition\_name, subpartition\_name, high\_value FROM ALL\_TAB\_SUBPARTITIONS;

partition\_name | subpartition\_name | high\_value

----------------+-------------------+---------------

europe | europe\_2011 | '2012-Jan-01'

europe | europe\_2012 | '2013-Jan-01'

asia | asia\_2011 | '2012-Jan-01'

asia | asia\_2012 | '2013-Jan-01'

americas | americas\_2011 | '2012-Jan-01'

americas | americas\_2012 | '2013-Jan-01'

(6 rows)

The following command adds rows to each subpartition:

INSERT INTO sales VALUES

(10, '4519b', 'FRANCE', '17-Jan-2012', '45000'),

(20, '3788a', 'INDIA', '01-Mar-2012', '75000'),

(40, '9519b', 'US', '12-Apr-2012', '145000'),

(20, '3788a', 'PAKISTAN', '04-Jun-2012', '37500'),

(40, '4577b', 'US', '11-Nov-2012', '25000'),

(30, '7588b', 'CANADA', '14-Dec-2012', '50000'),

(30, '9519b', 'CANADA', '01-Feb-2012', '75000'),

(30, '4519b', 'CANADA', '08-Apr-2012', '120000'),

(40, '3788a', 'US', '12-May-2012', '4950'),

(10, '9519b', 'ITALY', '07-Jul-2012', '15000'),

(10, '9519a', 'FRANCE', '18-Aug-2012', '650000'),

(10, '9519b', 'FRANCE', '18-Aug-2012', '650000'),

(20, '3788b', 'INDIA', '21-Sept-2012', '5090'),

(40, '4788a', 'US', '23-Sept-2012', '4950'),

(40, '4788b', 'US', '09-Oct-2012', '15000'),

(20, '4519a', 'INDIA', '18-Oct-2012', '650000'),

(20, '4519b', 'INDIA', '2-Dec-2012', '5090');

A SELECT statement confirms that the rows are distributed amongst the subpartions:

acctg=# SELECT tableoid::regclass, \* FROM sales;

tableoid | dept\_no|part\_no| country | date |amount

---------------------+--------+-------+---------+--------------------+---

sales\_europe\_2012 | 10| 4519b | FRANCE | 17-JAN-12 00:00:00 | 45000

sales\_europe\_2012 | 10| 9519b | ITALY | 07-JUL-12 00:00:00 | 15000

sales\_europe\_2012 | 10| 9519a | FRANCE | 18-AUG-12 00:00:00 | 650000

sales\_europe\_2012 | 10| 9519b | FRANCE | 18-AUG-12 00:00:00 | 650000

sales\_asia\_2012 | 20| 3788a | INDIA | 01-MAR-12 00:00:00 | 75000

sales\_asia\_2012 | 20| 3788a | PAKISTAN| 04-JUN-12 00:00:00 | 37500

sales\_asia\_2012 | 20| 3788b | INDIA | 21-SEP-12 00:00:00 | 5090

sales\_asia\_2012 | 20| 4519a | INDIA | 18-OCT-12 00:00:00 | 650000

sales\_asia\_2012 | 20| 4519b | INDIA | 02-DEC-12 00:00:00 | 5090

sales\_americas\_2012 | 40| 9519b | US | 12-APR-12 00:00:00 | 145000

sales\_americas\_2012 | 40| 4577b | US | 11-NOV-12 00:00:00 | 25000

sales\_americas\_2012 | 30| 7588b | CANADA | 14-DEC-12 00:00:00 | 50000

sales\_americas\_2012 | 30| 9519b | CANADA | 01-FEB-12 00:00:00 | 75000

sales\_americas\_2012 | 30| 4519b | CANADA | 08-APR-12 00:00:00 | 120000

sales\_americas\_2012 | 40| 3788a | US | 12-MAY-12 00:00:00 | 4950

sales\_americas\_2012 | 40| 4788a | US | 23-SEP-12 00:00:00 | 4950

sales\_americas\_2012 | 40| 4788b | US | 09-OCT-12 00:00:00 | 15000

(17 rows)

The following command splits the americas\_2012 subpartition into two new subpartitions, and redistributes the contents:

ALTER TABLE sales   
 SPLIT SUBPARTITION americas\_2012   
 AT('2012-Jun-01')

INTO

(  
 SUBPARTITION americas\_p1\_2012,

SUBPARTITION americas\_p2\_2012

);

After invoking the command, the americas\_2012 subpartition has been deleted; in it's place, the server has created two new subpartitions (americas\_p1\_2012 and americas\_p2\_2012):

acctg=# SELECT partition\_name, subpartition\_name, high\_value FROM ALL\_TAB\_SUBPARTITIONS;

partition\_name | subpartition\_name | high\_value

----------------+-------------------+---------------

europe | europe\_2012 | '2013-Jan-01'

europe | europe\_2011 | '2012-Jan-01'

americas | americas\_2011 | '2012-Jan-01'

americas | americas\_p2\_2012 | '2013-Jan-01'

americas | americas\_p1\_2012 | '2012-Jun-01'

asia | asia\_2012 | '2013-Jan-01'

asia | asia\_2011 | '2012-Jan-01'

(7 rows)

Querying the sales table demonstrates that the subpartition's contents are redistributed:

acctg=# SELECT tableoid::regclass, \* FROM sales;

tableoid | dept\_no|part\_no|country | date |amount

-----------------------+--------+-------+--------+-------------------+------- sales\_europe\_2012 | 10| 4519b |FRANCE | 17-JAN-12 00:00:00| 45000

sales\_europe\_2012 | 10| 9519b |ITALY | 07-JUL-12 00:00:00| 15000

sales\_europe\_2012 | 10| 9519a |FRANCE | 18-AUG-12 00:00:00| 650000

sales\_europe\_2012 | 10| 9519b |FRANCE | 18-AUG-12 00:00:00| 650000

sales\_asia\_2012 | 20| 3788a |INDIA | 01-MAR-12 00:00:00| 75000

sales\_asia\_2012 | 20| 3788a |PAKISTAN| 04-JUN-12 00:00:00| 37500

sales\_asia\_2012 | 20| 3788b |INDIA | 21-SEP-12 00:00:00| 5090

sales\_asia\_2012 | 20| 4519a |INDIA | 18-OCT-12 00:00:00| 650000

sales\_asia\_2012 | 20| 4519b |INDIA | 02-DEC-12 00:00:00| 5090

sales\_americas\_p1\_2012| 40| 9519b |US | 12-APR-12 00:00:00| 145000

sales\_americas\_p1\_2012| 30| 9519b |CANADA | 01-FEB-12 00:00:00| 75000

sales\_americas\_p1\_2012| 30| 4519b |CANADA | 08-APR-12 00:00:00| 120000

sales\_americas\_p1\_2012| 40| 3788a |US | 12-MAY-12 00:00:00| 4950

sales\_americas\_p2\_2012| 40| 4577b |US | 11-NOV-12 00:00:00| 25000

sales\_americas\_p2\_2012| 30| 7588b |CANADA | 14-DEC-12 00:00:00| 50000

sales\_americas\_p2\_2012| 40| 4788a |US | 23-SEP-12 00:00:00| 4950

sales\_americas\_p2\_2012| 40| 4788b |US | 09-OCT-12 00:00:00| 15000

(17 rows)

### ALTER TABLE… EXCHANGE PARTITION

The ALTER TABLE…EXCHANGE PARTITION command swaps an existing table with a partition or subpartition. If you plan to add a large quantity of data to a partitioned table, you can use the ALTER TABLE… EXCHANGE PARTITION command to implement a bulk load. You can also use the ALTER TABLE… EXCHANGE PARTITION command to remove old or unneeded data for storage.

The command syntax is available in two forms. The first form swaps a table for a partition:

ALTER TABLE *target\_table*   
 EXCHANGE PARTITION *target\_partition*   
 WITH TABLE *source\_table*   
 [(WITH | WITHOUT) VALIDATION];

The second form swaps a table for a subpartition:

ALTER TABLE *target\_table*   
 EXCHANGE SUBPARTITION *target*\_sub*partition*   
 WITH TABLE *source\_table*   
 [(WITH | WITHOUT) VALIDATION];

This command makes no distinction between a partition and a subpartition:

* You can exchange a partition with the EXCHANGE PARTITION or EXCHANGE SUBPARTITION clause.
* You can exchange a subpartition with EXCHANGE PARTITION or EXCHANGE SUBPARTITION clause.

**Description**

When the ALTER TABLE… EXCHANGE PARTITION command completes, the data originally located in the *target\_partition* will be located in the *source\_table*, and the data originally located in the *source\_table* will be located in the *target\_partition*.

The ALTER TABLE… EXCHANGE PARTITION command can exchange partitions in a LIST, RANGE or HASH partitioned table. The structure of the *source\_table* must match the structure of the *target\_table* (both tables must have matching columns and data types), and the data contained within the table must adhere to the partitioning constraints.

Advanced Server accepts the WITHOUT VALIDATION clause, but ignores it; the new table is always validated.

You must own a table to invoke ALTER TABLE… EXCHANGE PARTITION or ALTER TABLE… EXCHANGE SUBPARTITION against that table.

**Parameters**

target\_table

The name (optionally schema-qualified) of the table in which the partition resides.

target\_partition

The name of the partition or subpartition to be replaced.

source\_table

The name of the table that will replace the *target\_partition*.

#### Example - Exchanging a Table for a Partition

The example that follows demonstrates swapping a table for a partition (americas) of the sales table. You can create the sales table with the following command:

CREATE TABLE sales

(

dept\_no number,

part\_no varchar2,

country varchar2(20),

date date,

amount number

)

PARTITION BY LIST(country)

(

PARTITION europe VALUES('FRANCE', 'ITALY'),

PARTITION asia VALUES('INDIA', 'PAKISTAN'),

PARTITION americas VALUES('US', 'CANADA')

);

Use the following command to add sample data to the sales table:

INSERT INTO sales VALUES

(40, '9519b', 'US', '12-Apr-2012', '145000'),

(10, '4519b', 'FRANCE', '17-Jan-2012', '45000'),

(20, '3788a', 'INDIA', '01-Mar-2012', '75000'),

(20, '3788a', 'PAKISTAN', '04-Jun-2012', '37500'),

(10, '9519b', 'ITALY', '07-Jul-2012', '15000'),

(10, '9519a', 'FRANCE', '18-Aug-2012', '650000'),

(10, '9519b', 'FRANCE', '18-Aug-2012', '650000'),

(20, '3788b', 'INDIA', '21-Sept-2012', '5090'),

(20, '4519a', 'INDIA', '18-Oct-2012', '650000'),

(20, '4519b', 'INDIA', '2-Dec-2012', '5090');

Querying the sales table shows that only one row resides in the americas partition:

acctg=# SELECT tableoid::regclass, \* FROM sales;

tableoid | dept\_no| part\_no | country | date | amount

---------------+--------+---------+---------+-------------------+-----------

sales\_europe | 10| 4519b | FRANCE | 17-JAN-12 00:00:00| 45000

sales\_europe | 10| 9519b | ITALY | 07-JUL-12 00:00:00| 15000

sales\_europe | 10| 9519a | FRANCE | 18-AUG-12 00:00:00| 650000

sales\_europe | 10| 9519b | FRANCE | 18-AUG-12 00:00:00| 650000

sales\_asia | 20| 3788a | INDIA | 01-MAR-12 00:00:00| 75000

sales\_asia | 20| 3788a | PAKISTAN| 04-JUN-12 00:00:00| 37500

sales\_asia | 20| 3788b | INDIA | 21-SEP-12 00:00:00| 5090

sales\_asia | 20| 4519a | INDIA | 18-OCT-12 00:00:00| 650000

sales\_asia | 20| 4519b | INDIA | 02-DEC-12 00:00:00| 5090

sales\_americas| 40| 9519b | US | 12-APR-12 00:00:00| 145000

(10 rows)

The following command creates a table (n\_america) that matches the definition of the sales table:

CREATE TABLE n\_america

(

dept\_no number,

part\_no varchar2,

country varchar2(20),

date date,

amount number

);

The following command adds data to the n\_america table. The data conforms to the partitioning rules of the americas partition:

INSERT INTO n\_america VALUES

(40, '9519b', 'US', '12-Apr-2012', '145000'),

(40, '4577b', 'US', '11-Nov-2012', '25000'),

(30, '7588b', 'CANADA', '14-Dec-2012', '50000'),

(30, '9519b', 'CANADA', '01-Feb-2012', '75000'),

(30, '4519b', 'CANADA', '08-Apr-2012', '120000'),

(40, '3788a', 'US', '12-May-2012', '4950'),

(40, '4788a', 'US', '23-Sept-2012', '4950'),

(40, '4788b', 'US', '09-Oct-2012', '15000');

The following command swaps the table into the partitioned table:

ALTER TABLE sales   
 EXCHANGE PARTITION americas   
 WITH TABLE n\_america;

Querying the sales table shows that the contents of the n\_america table has been exchanged for the content of the americas partition:

acctg=# SELECT tableoid::regclass, \* FROM sales;

tableoid | dept\_no| part\_no | country | date | amount

---------------+--------+--------+----------+--------------------+-----------

sales\_europe | 10| 4519b | FRANCE | 17-JAN-12 00:00:00 | 45000

sales\_europe | 10| 9519b | ITALY | 07-JUL-12 00:00:00 | 15000

sales\_europe | 10| 9519a | FRANCE | 18-AUG-12 00:00:00 | 650000

sales\_europe | 10| 9519b | FRANCE | 18-AUG-12 00:00:00 | 650000

sales\_asia | 20| 3788a | INDIA | 01-MAR-12 00:00:00 | 75000

sales\_asia | 20| 3788a | PAKISTAN| 04-JUN-12 00:00:00 | 37500

sales\_asia | 20| 3788b | INDIA | 21-SEP-12 00:00:00 | 5090

sales\_asia | 20| 4519a | INDIA | 18-OCT-12 00:00:00 | 650000

sales\_asia | 20| 4519b | INDIA | 02-DEC-12 00:00:00 | 5090

sales\_americas| 40| 9519b | US | 12-APR-12 00:00:00 | 145000

sales\_americas| 40| 4577b | US | 11-NOV-12 00:00:00 | 25000

sales\_americas| 30| 7588b | CANADA | 14-DEC-12 00:00:00 | 50000

sales\_americas| 30| 9519b | CANADA | 01-FEB-12 00:00:00 | 75000

sales\_americas| 30| 4519b | CANADA | 08-APR-12 00:00:00 | 120000

sales\_americas| 40| 3788a | US | 12-MAY-12 00:00:00 | 4950

sales\_americas| 40| 4788a | US | 23-SEP-12 00:00:00 | 4950

sales\_americas| 40| 4788b | US | 09-OCT-12 00:00:00 | 15000

(17 rows)

Querying the n\_america table shows that the row that was previously stored in the americas partition has been moved to the n\_america table:

acctg=# SELECT tableoid::regclass, \* FROM n\_america;

tableoid | dept\_no | part\_no | country | date | amount

-----------+---------+---------+---------+--------------------+------------

n\_america | 40 | 9519b | US | 12-APR-12 00:00:00 | 145000

(1 row)

### ALTER TABLE… MOVE PARTITION

Use the ALTER TABLE… MOVE PARTITION command to move a partition or subpartition to a different tablespace. The command takes two forms.

The first form moves a partition to a new tablespace:

ALTER TABLE *table*\_*name*   
 MOVE PARTITION *partition*\_*name*   
 TABLESPACE *tablespace*\_*name*;

The second form moves a subpartition to a new tablespace:

ALTER TABLE *table*\_*name*   
 MOVE SUBPARTITION sub*partition*\_*name*   
 TABLESPACE *tablespace*\_*name*;

The command syntax makes no distinctions between a partition and a subpartition:

* You can move a partition with the MOVE PARTITION or MOVE SUBPARTITION clause.
* You can move a subpartition with MOVE PARTITION or MOVE SUBPARTITION clause.

**Description**

The ALTER TABLE…MOVE PARTITION command moves a partition or subpartition from it's current tablespace to a different tablespace. The ALTER TABLE… MOVE PARTITION command can move partitions (or subpartitions) of a LIST, RANGE or HASH partitioned (or subpartitioned) table. You must own a table to invoke ALTER TABLE… MOVE PARTITION or ALTER TABLE… MOVE SUBPARTITION.

**Parameters**

table\_name

The name (optionally schema-qualified) of the table in which the partition resides.

partition\_name

The name of the partition or subpartition to be moved.

tablespace\_name

The name of the tablespace to which the partition or subpartition will be moved.

#### Example - Moving a Partition to a Different Tablespace

The following example moves a partition of the sales table from one tablespace to another. First, create the sales table with the command:

CREATE TABLE sales  
(  
 dept\_no number,  
 part\_no varchar2,  
 country varchar2(20),  
 date date,  
 amount number  
)  
PARTITION BY RANGE(date)  
(  
 PARTITION q1\_2012 VALUES LESS THAN ('2012-Apr-01'),  
 PARTITION q2\_2012 VALUES LESS THAN ('2012-Jul-01'),  
 PARTITION q3\_2012 VALUES LESS THAN ('2012-Oct-01'),  
 PARTITION q4\_2012 VALUES LESS THAN ('2013-Jan-01') TABLESPACE ts\_1,  
 PARTITION q1\_2013 VALUES LESS THAN ('2013-Mar-01') TABLESPACE ts\_2  
);

Querying the ALL\_TAB\_PARTITIONS view confirms that the partitions reside on the expected servers and tablespaces:

acctg=# SELECT partition\_name, tablespace\_name FROM ALL\_TAB\_PARTITIONS;

partition\_name | tablespace\_name

----------------+-------------+-----------------

q1\_2013 | ts\_2

q4\_2012 | ts\_1

q3\_2012 |

q2\_2012 |

q1\_2012 |

(5 rows)

After preparing the target tablespace, invoke the ALTER TABLE… MOVE PARTITION command to move the q1\_2013 partition from a tablespace named ts\_2 to a tablespace named ts\_3:

ALTER TABLE sales MOVE PARTITION q1\_2013 TABLESPACE ts\_3;

Querying the ALL\_TAB\_PARTITIONS view shows that the move was successful:

acctg=# SELECT partition\_name, tablespace\_name FROM ALL\_TAB\_PARTITIONS;

partition\_name | tablespace\_name

----------------+-----------------

q1\_2013 | ts\_3

q4\_2012 | ts\_1

q3\_2012 |

q2\_2012 |

q1\_2012 |

(5 rows)

### ALTER TABLE… RENAME PARTITION

Use the ALTER TABLE… RENAME PARTITION command to rename a table partition. The syntax takes two forms:

ALTER TABLE *table*\_*name*   
 RENAME PARTITION *partition*\_*name*   
 TO *new*\_*name*;

and

ALTER TABLE *table*\_*name*   
 RENAME SUBPARTITION *subpartition*\_*name*   
 TO *new*\_*name*;

This command makes no distinctions between a partition and a subpartition:

* You can rename a partition with the RENAME PARTITION or RENAME SUBPARTITION clause.
* You can rename a subpartition with RENAME PARTITION or RENAME SUBPARTITION clause.

**Description**

The ALTER TABLE… RENAME PARTITION and ALTER TABLE… RENAME SUBPARTITION commands rename a partition or subpartition. You must own the specified table to invoke ALTER TABLE… RENAME PARTITION or ALTER TABLE… RENAME SUBPARTITION.

**Parameters**

table\_name

The name (optionally schema-qualified) of the table in which the partition resides.

partition\_name

The name of the partition or subpartition to be renamed.

new\_name

The new name of the partition or subpartition.

#### Example - Renaming a Partition

The following command creates a list-partitioned table named sales:

CREATE TABLE sales

(

dept\_no number,

part\_no varchar2,

country varchar2(20),

date date,

amount number

)

PARTITION BY LIST(country)

(

PARTITION europe VALUES('FRANCE', 'ITALY'),

PARTITION asia VALUES('INDIA', 'PAKISTAN'),

PARTITION americas VALUES('US', 'CANADA')

);

Query the ALL\_TAB\_PARTITIONS view to display the partition names:

acctg=# SELECT partition\_name, high\_value FROM ALL\_TAB\_PARTITIONS;

partition\_name | high\_value

----------------+---------------------

europe | 'FRANCE', 'ITALY'

asia | 'INDIA', 'PAKISTAN'

americas | 'US', 'CANADA'

(3 rows)

The following command renames the americas partition to n\_america:

ALTER TABLE sales   
 RENAME PARTITION americas TO n\_america;

Querying the ALL\_TAB\_PARTITIONS view demonstrates that the partition has been successfully renamed:

acctg=# SELECT partition\_name, high\_value FROM ALL\_TAB\_PARTITIONS;

partition\_name | high\_value

----------------+---------------------

europe | 'FRANCE', 'ITALY'

asia | 'INDIA', 'PAKISTAN'

n\_america | 'US', 'CANADA'

(3 rows)

### DROP TABLE

Use the PostgreSQL DROP TABLE command to remove a partitioned table definition, it's partitions and subpartitions, and delete the table contents. The syntax is:

DROP TABLE *table*\_*name*

**Description**

The DROP TABLE command removes an entire table, and the data that resides in that table. When you delete a table, any partitions or subpartitions (of that table) are deleted as well.

To use the DROP TABLE command, you must be the owner of the partitioning root, a member of a group that owns the table, the schema owner, or a database superuser.

**Parameters**

table\_name

The name (optionally schema-qualified) of the partitioned table.

**Example**

To delete a table, connect to the controller node (the host of the partitioning root), and invoke the DROP TABLE command. For example, to delete the sales table, invoke the following command:

DROP TABLE sales;

The server will confirm that the table has been dropped:

acctg=# drop table sales;

DROP TABLE

acctg=#

For more information about the DROP TABLE command, please refer to the PostgreSQL core documentation at:

<http://www.postgresql.org/docs/9.5/static/sql-droptable.html>

### ALTER TABLE… DROP PARTITION

Use the ALTER TABLE… DROP PARTITION command to delete a partition definition, and the data stored in that partition. The syntax is:

ALTER TABLE *table*\_*name* DROP PARTITION *partition*\_*name;*

**Parameters**

table\_name

The name (optionally schema-qualified) of the partitioned table.

partition\_name

The name of the partition to be deleted.

**Description**

The ALTER TABLE… DROP PARTITION command deletes a partition and any data stored on that partition. The ALTER TABLE… DROP PARTITION command can drop partitions of a LIST or RANGE partitioned table; please note that this command does not work on a HASH partitioned table. When you delete a partition, any subpartitions (of that partition) are deleted as well.

To use the DROP PARTITION clause, you must be the owner of the partitioning root, a member of a group that owns the table, or have database superuser or administrative privileges.

#### Example - Deleting a Partition

The example that follows deletes a partition of the sales table. Use the following command to create the sales table:

CREATE TABLE sales

(

dept\_no number,

part\_no varchar2,

country varchar2(20),

date date,

amount number

)

PARTITION BY LIST(country)

(

PARTITION europe VALUES('FRANCE', 'ITALY'),

PARTITION asia VALUES('INDIA', 'PAKISTAN'),

PARTITION americas VALUES('US', 'CANADA')

);

Querying the ALL\_TAB\_PARTITIONS view displays the partition names:

acctg=# SELECT partition\_name, server\_name, high\_value FROM ALL\_TAB\_PARTITIONS;

partition\_name | server\_name | high\_value

----------------+-------------+---------------------

europe | seattle | 'FRANCE', 'ITALY'

asia | chicago | 'INDIA', 'PAKISTAN'

americas | boston | 'US', 'CANADA'

(3 rows)

To delete the americas partition from the sales table, invoke the following command:

ALTER TABLE sales DROP PARTITION americas;

Querying the ALL\_TAB\_PARTITIONS view demonstrates that the partition has been successfully deleted:

acctg=# SELECT partition\_name, server\_name, high\_value FROM ALL\_TAB\_PARTITIONS;

partition\_name | high\_value

----------------+---------------------

asia | 'INDIA', 'PAKISTAN'

europe | 'FRANCE', 'ITALY'

(2 rows)

### ALTER TABLE… DROP SUBPARTITION

Use the ALTER TABLE… DROP SUBPARTITION command to drop a subpartition definition, and the data stored in that subpartition. The syntax is:

ALTER TABLE *table*\_*name* DROP SUBPARTITION *subpartition*\_*name;*

**Parameters**

table\_name

The name (optionally schema-qualified) of the partitioned table.

subpartition\_name

The name of the subpartition to be deleted.

**Description**

The ALTER TABLE… DROP SUBPARTITION command deletes a subpartition, and the data stored in that subpartition. To use the DROP SUBPARTITION clause, you must be the owner of the partitioning root, a member of a group that owns the table, or have superuser or administrative privileges.

#### Example - Deleting a Subpartition

The example that follows deletes a subpartition of the sales table. Use the following command to create the sales table:

CREATE TABLE sales

(

dept\_no number,

part\_no varchar2,

country varchar2(20),

date date,

amount number

)  
PARTITION BY RANGE(date)  
 SUBPARTITION BY LIST (country)  
 (  
 PARTITION first\_half\_2012 VALUES LESS THAN('01-JUL-2012')  
 (

SUBPARTITION europe VALUES ('ITALY', 'FRANCE'),

SUBPARTITION americas VALUES ('CANADA', 'US'),

SUBPARTITION asia VALUES ('PAKISTAN', 'INDIA')

),

PARTITION second\_half\_2012 VALUES LESS THAN('01-JAN-2013')   
 );

Querying the ALL\_TAB\_SUBPARTITIONS view displays the subpartition names:

acctg=# SELECT subpartition\_name, high\_value, server\_name FROM ALL\_TAB\_SUBPARTITIONS; subpartition\_name | high\_value | server\_name

-------------------+---------------------+-------------

europe | 'ITALY', 'FRANCE' | chicago

americas | 'CANADA', 'US' | seattle

asia | 'PAKISTAN', 'INDIA' | boston

(3 rows)

To delete the americas subpartition from the sales table, invoke the following command:

ALTER TABLE sales DROP SUBPARTITION americas;

Querying the ALL\_TAB\_SUBPARTITIONS view demonstrates that the subpartition has been successfully deleted:

acctg=# SELECT subpartition\_name, high\_value FROM ALL\_TAB\_SUBPARTITIONS;

subpartition\_name | high\_value

-------------------+---------------------

europe | 'ITALY', 'FRANCE'

asia | 'PAKISTAN', 'INDIA'

(2 rows)

### TRUNCATE TABLE

Use the TRUNCATE TABLE command to remove the contents of a table, while preserving the table definition. When you truncate a table, any partitions or subpartitions of that table are also truncated. The syntax is:

TRUNCATE TABLE *table*\_*name*

**Description**

The TRUNCATE TABLE command removes an entire table, and the data that resides in that table. When you delete a table, any partitions or subpartitions (of that table) are deleted as well.

To use the TRUNCATE TABLE command, you must be the owner of the partitioning root, a member of a group that owns the table, the schema owner, or a database superuser.

**Parameters**

table\_name

The name (optionally schema-qualified) of the partitioned table.

#### Example - Emptying a Table

The example that follows removes the data from the sales table. Use the following command to create the sales table:

CREATE TABLE sales

(

dept\_no number,

part\_no varchar2,

country varchar2(20),

date date,

amount number

)

PARTITION BY LIST(country)

(

PARTITION europe VALUES('FRANCE', 'ITALY'),

PARTITION asia VALUES('INDIA', 'PAKISTAN'),

PARTITION americas VALUES('US', 'CANADA')

);

Populate the sales table with the command:

INSERT INTO sales VALUES

(10, '4519b', 'FRANCE', '17-Jan-2012', '45000'),

(20, '3788a', 'INDIA', '01-Mar-2012', '75000'),

(40, '9519b', 'US', '12-Apr-2012', '145000'),

(20, '3788a', 'PAKISTAN', '04-Jun-2012', '37500'),

(40, '4577b', 'US', '11-Nov-2012', '25000'),

(30, '7588b', 'CANADA', '14-Dec-2012', '50000'),

(30, '9519b', 'CANADA', '01-Feb-2012', '75000'),

(30, '4519b', 'CANADA', '08-Apr-2012', '120000'),

(40, '3788a', 'US', '12-May-2012', '4950'),

(10, '9519b', 'ITALY', '07-Jul-2012', '15000'),

(10, '9519a', 'FRANCE', '18-Aug-2012', '650000'),

(10, '9519b', 'FRANCE', '18-Aug-2012', '650000'),

(20, '3788b', 'INDIA', '21-Sept-2012', '5090'),

(40, '4788a', 'US', '23-Sept-2012', '4950'),

(40, '4788b', 'US', '09-Oct-2012', '15000'),

(20, '4519a', 'INDIA', '18-Oct-2012', '650000'),

(20, '4519b', 'INDIA', '2-Dec-2012', '5090');

Querying the sales table shows that the partitions are populated with data:

acctg=# SELECT tableoid::regclass, \* FROM sales;

tableoid |dept\_no | part\_no | country | date | amount

--------------+--------+---------+----------+--------------------+-----------

sales\_europe | 10 | 4519b | FRANCE | 17-JAN-12 00:00:00 | 45000

sales\_europe | 10 | 9519b | ITALY | 07-JUL-12 00:00:00 | 15000

sales\_europe | 10 | 9519a | FRANCE | 18-AUG-12 00:00:00 | 650000

sales\_europe | 10 | 9519b | FRANCE | 18-AUG-12 00:00:00 | 650000

sales\_asia | 20 | 3788a | INDIA | 01-MAR-12 00:00:00 | 75000

sales\_asia | 20 | 3788a | PAKISTAN | 04-JUN-12 00:00:00 | 37500

sales\_asia | 20 | 3788b | INDIA | 21-SEP-12 00:00:00 | 5090

sales\_asia | 20 | 4519a | INDIA | 18-OCT-12 00:00:00 | 650000

sales\_asia | 20 | 4519b | INDIA | 02-DEC-12 00:00:00 | 5090

sales\_americas| 40 | 9519b | US | 12-APR-12 00:00:00 | 145000

sales\_americas| 40 | 4577b | US | 11-NOV-12 00:00:00 | 25000

sales\_americas| 30 | 7588b | CANADA | 14-DEC-12 00:00:00 | 50000

sales\_americas| 30 | 9519b | CANADA | 01-FEB-12 00:00:00 | 75000

sales\_americas| 30 | 4519b | CANADA | 08-APR-12 00:00:00 | 120000

sales\_americas| 40 | 3788a | US | 12-MAY-12 00:00:00 | 4950

sales\_americas| 40 | 4788a | US | 23-SEP-12 00:00:00 | 4950

sales\_americas| 40 | 4788b | US | 09-OCT-12 00:00:00 | 15000

(17 rows)

To delete the contents of the sales table, invoke the following command:

TRUNCATE TABLE sales;

Now, querying the sales table shows that the data has been removed but the structure is intact:

acctg=# SELECT tableoid::regclass, \* FROM sales;

tableoid | dept\_no | part\_no | country | date | amount

----------+---------+---------+---------+----------+------------

(0 rows)

For more information about the TRUNCATE TABLE command, please see the PostgreSQL documentation at:

<http://www.postgresql.org/docs/9.5/static/sql-truncate.html>

### ALTER TABLE… TRUNCATE PARTITION

Use the ALTER TABLE… TRUNCATE PARTITION command to remove the data from the specified partition, leaving the partition structure intact. The syntax is:

ALTER TABLE *table\_name* TRUNCATE PARTITION *partition\_name* [{DROP|REUSE} STORAGE]

**Description**

Use the ALTER TABLE… TRUNCATE PARTITION command to remove the data from the specified partition, leaving the partition structure intact. When you truncate a partition, any subpartitions of that partition are also truncated.

ALTER TABLE… TRUNCATE PARTITION will not cause ON DELETE triggers that might exist for the table to fire, but it will fire ON TRUNCATE triggers. If an ON TRUNCATE trigger is defined for the partition, all BEFORE TRUNCATE triggers are fired before any truncation happens, and all AFTER TRUNCATE triggers are fired after the last truncation occurs.

You must have the TRUNCATE privilege on a table to invoke ALTER TABLE… TRUNCATE PARTITION.

**Parameters**

table\_name

The name (optionally schema-qualified) of the partitioned table.

partition\_name

The name of the partition to be deleted.

DROP STORAGE and REUSE STORAGE are included for compatibility only; the clauses are parsed and ignored.

#### Example - Emptying a Partition

The example that follows removes the data from a partition of the sales table. Use the following command to create the sales table:

CREATE TABLE sales

(

dept\_no number,

part\_no varchar2,

country varchar2(20),

date date,

amount number

)

PARTITION BY LIST(country)

(

PARTITION europe VALUES('FRANCE', 'ITALY'),

PARTITION asia VALUES('INDIA', 'PAKISTAN'),

PARTITION americas VALUES('US', 'CANADA')

);

Populate the sales table with the command:

INSERT INTO sales VALUES

(10, '4519b', 'FRANCE', '17-Jan-2012', '45000'),

(20, '3788a', 'INDIA', '01-Mar-2012', '75000'),

(40, '9519b', 'US', '12-Apr-2012', '145000'),

(20, '3788a', 'PAKISTAN', '04-Jun-2012', '37500'),

(40, '4577b', 'US', '11-Nov-2012', '25000'),

(30, '7588b', 'CANADA', '14-Dec-2012', '50000'),

(30, '9519b', 'CANADA', '01-Feb-2012', '75000'),

(30, '4519b', 'CANADA', '08-Apr-2012', '120000'),

(40, '3788a', 'US', '12-May-2012', '4950'),

(10, '9519b', 'ITALY', '07-Jul-2012', '15000'),

(10, '9519a', 'FRANCE', '18-Aug-2012', '650000'),

(10, '9519b', 'FRANCE', '18-Aug-2012', '650000'),

(20, '3788b', 'INDIA', '21-Sept-2012', '5090'),

(40, '4788a', 'US', '23-Sept-2012', '4950'),

(40, '4788b', 'US', '09-Oct-2012', '15000'),

(20, '4519a', 'INDIA', '18-Oct-2012', '650000'),

(20, '4519b', 'INDIA', '2-Dec-2012', '5090');

Querying the sales table shows that the partitions are populated with data:

acctg=# SELECT tableoid::regclass, \* FROM sales;

tableoid | dept\_no | part\_no | country | date | amount

----------------+---------+---------+----------+--------------------+--------

sales\_europe | 10 | 4519b | FRANCE | 17-JAN-12 00:00:00 | 45000

sales\_europe | 10 | 9519b | ITALY | 07-JUL-12 00:00:00 | 15000

sales\_europe | 10 | 9519a | FRANCE | 18-AUG-12 00:00:00 | 650000

sales\_europe | 10 | 9519b | FRANCE | 18-AUG-12 00:00:00 | 650000

sales\_asia | 20 | 3788a | INDIA | 01-MAR-12 00:00:00 | 75000

sales\_asia | 20 | 3788a | PAKISTAN | 04-JUN-12 00:00:00 | 37500

sales\_asia | 20 | 3788b | INDIA | 21-SEP-12 00:00:00 | 5090

sales\_asia | 20 | 4519a | INDIA | 18-OCT-12 00:00:00 | 650000

sales\_asia | 20 | 4519b | INDIA | 02-DEC-12 00:00:00 | 5090

sales\_americas | 40 | 9519b | US | 12-APR-12 00:00:00 | 145000

sales\_americas | 40 | 4577b | US | 11-NOV-12 00:00:00 | 25000

sales\_americas | 30 | 7588b | CANADA | 14-DEC-12 00:00:00 | 50000

sales\_americas | 30 | 9519b | CANADA | 01-FEB-12 00:00:00 | 75000

sales\_americas | 30 | 4519b | CANADA | 08-APR-12 00:00:00 | 120000

sales\_americas | 40 | 3788a | US | 12-MAY-12 00:00:00 | 4950

sales\_americas | 40 | 4788a | US | 23-SEP-12 00:00:00 | 4950

sales\_americas | 40 | 4788b | US | 09-OCT-12 00:00:00 | 15000

(17 rows)

To delete the contents of the americas partition, invoke the following command:

ALTER TABLE sales TRUNCATE PARTITION americas;

Now, querying the sales table shows that the content of the americas partition has been removed:

acctg=# SELECT tableoid::regclass, \* FROM sales;

tableoid | dept\_no | part\_no | country | date | amount

--------------+---------+---------+----------+--------------------+--------

sales\_europe | 10 | 4519b | FRANCE | 17-JAN-12 00:00:00 | 45000

sales\_europe | 10 | 9519b | ITALY | 07-JUL-12 00:00:00 | 15000

sales\_europe | 10 | 9519a | FRANCE | 18-AUG-12 00:00:00 | 650000

sales\_europe | 10 | 9519b | FRANCE | 18-AUG-12 00:00:00 | 650000

sales\_asia | 20 | 3788a | INDIA | 01-MAR-12 00:00:00 | 75000

sales\_asia | 20 | 3788a | PAKISTAN | 04-JUN-12 00:00:00 | 37500

sales\_asia | 20 | 3788b | INDIA | 21-SEP-12 00:00:00 | 5090

sales\_asia | 20 | 4519a | INDIA | 18-OCT-12 00:00:00 | 650000

sales\_asia | 20 | 4519b | INDIA | 02-DEC-12 00:00:00 | 5090

(9 rows)

While the rows have been removed, the structure of the americas partition is still intact:

acctg=# SELECT partition\_name, high\_value FROM ALL\_TAB\_PARTITIONS;

partition\_name | high\_value

----------------+---------------------

europe | 'FRANCE', 'ITALY'

asia | 'INDIA', 'PAKISTAN'

americas | 'US', 'CANADA'

(3 rows)

### ALTER TABLE… TRUNCATE SUBPARTITION

Use the ALTER TABLE… TRUNCATE SUBPARTITION command to remove all of the data from the specified subpartition, leaving the subpartition structure intact. The syntax is:

ALTER TABLE *table\_name*   
 TRUNCATE SUBPARTITION *subpartition\_name* [{DROP|REUSE} STORAGE]

**Description**

The ALTER TABLE… TRUNCATE SUBPARTITION command removes all data from a specified subpartition, leaving the subpartition structure intact.

ALTER TABLE… TRUNCATE SUBPARTITION will not cause ON DELETE triggers that might exist for the table to fire, but it will fire ON TRUNCATE triggers. If an ON TRUNCATE trigger is defined for the subpartition, all BEFORE TRUNCATE triggers are fired before any truncation happens, and all AFTER TRUNCATE triggers are fired after the last truncation occurs.

You must have the TRUNCATE privilege on a table to invoke ALTER TABLE… TRUNCATE SUBPARTITION.

**Parameters**

table\_name

The name (optionally schema-qualified) of the partitioned table.

subpartition\_name

The name of the subpartition to be truncated.

The DROP STORAGE and REUSE STORAGE clauses are included for compatibility only; the clauses are parsed and ignored.

#### Example - Emptying a Subpartition

The example that follows removes the data from a subpartition of the sales table. Use the following command to create the sales table:

CREATE TABLE sales

(

dept\_no number,

part\_no varchar2,

country varchar2(20),

date date,

amount number

)

PARTITION BY RANGE(date) SUBPARTITION BY LIST (country)

(  
 PARTITION "2011" VALUES LESS THAN('01-JAN-2012')

(

SUBPARTITION europe\_2011 VALUES ('ITALY', 'FRANCE'),

SUBPARTITION asia\_2011 VALUES ('PAKISTAN', 'INDIA'),

SUBPARTITION americas\_2011 VALUES ('US', 'CANADA')

),

PARTITION "2012" VALUES LESS THAN('01-JAN-2013')

(

SUBPARTITION europe\_2012 VALUES ('ITALY', 'FRANCE'),

SUBPARTITION asia\_2012 VALUES ('PAKISTAN', 'INDIA'),

SUBPARTITION americas\_2012 VALUES ('US', 'CANADA')

),  
 PARTITION "2013" VALUES LESS THAN('01-JAN-2014')

(

SUBPARTITION europe\_2013 VALUES ('ITALY', 'FRANCE'),

SUBPARTITION asia\_2013 VALUES ('PAKISTAN', 'INDIA'),

SUBPARTITION americas\_2013 VALUES ('US', 'CANADA')

)

);

Populate the sales table with the command:

INSERT INTO sales VALUES

(10, '4519b', 'FRANCE', '17-Jan-2011', '45000'),

(20, '3788a', 'INDIA', '01-Mar-2012', '75000'),

(40, '9519b', 'US', '12-Apr-2012', '145000'),

(20, '3788a', 'PAKISTAN', '04-Jun-2012', '37500'),

(40, '4577b', 'US', '11-Nov-2012', '25000'),

(30, '7588b', 'CANADA', '14-Dec-2011', '50000'),

(30, '4519b', 'CANADA', '08-Apr-2012', '120000'),

(40, '3788a', 'US', '12-May-2011', '4950'),

(20, '3788a', 'US', '04-Apr-2012', '37500'),

(40, '4577b', 'INDIA', '11-Jun-2011', '25000'),

(10, '9519b', 'ITALY', '07-Jul-2012', '15000'),

(20, '4519b', 'INDIA', '2-Dec-2012', '5090');

Querying the sales table shows that the rows have been distributed amongst the subpartitions:

acctg=# SELECT tableoid::regclass, \* FROM sales;

tableoid | dept\_no| part\_no| country | date |amount

--------------------+--------+--------+----------+-------------------+-------

sales\_2011\_europe | 10| 4519b | FRANCE | 17-JAN-11 00:00:00| 45000

sales\_2011\_asia | 40| 4577b | INDIA | 11-JUN-11 00:00:00| 25000

sales\_2011\_americas| 30| 7588b | CANADA | 14-DEC-11 00:00:00| 50000

sales\_2011\_americas| 40| 3788a | US | 12-MAY-11 00:00:00| 4950

sales\_2012\_europe | 10| 9519b | ITALY | 07-JUL-12 00:00:00| 15000

sales\_2012\_asia | 20| 3788a | INDIA | 01-MAR-12 00:00:00| 75000

sales\_2012\_asia | 20| 3788a | PAKISTAN | 04-JUN-12 00:00:00| 37500

sales\_2012\_asia | 20| 4519b | INDIA | 02-DEC-12 00:00:00| 5090

sales\_2012\_americas| 40| 9519b | US | 12-APR-12 00:00:00| 145000

sales\_2012\_americas| 40| 4577b | US | 11-NOV-12 00:00:00| 25000

sales\_2012\_americas| 30| 4519b | CANADA | 08-APR-12 00:00:00| 120000

sales\_2012\_americas| 20| 3788a | US | 04-APR-12 00:00:00| 37500

(12 rows)

To delete the contents of the 2012\_americas partition, invoke the following command:

ALTER TABLE sales TRUNCATE SUBPARTITION "americas\_2012";

Now, querying the sales table shows that the content of the americas\_2012 partition has been removed:

acctg=# SELECT tableoid::regclass, \* FROM sales;

tableoid | dept\_no|part\_no| country | date | amount

--------------------+--------+-------+----------+--------------------+-------

sales\_2011\_europe | 10| 4519b | FRANCE | 17-JAN-11 00:00:00 | 45000

sales\_2011\_asia | 40| 4577b | INDIA | 11-JUN-11 00:00:00 | 25000

sales\_2011\_americas| 30| 7588b | CANADA | 14-DEC-11 00:00:00 | 50000

sales\_2011\_americas| 40| 3788a | US | 12-MAY-11 00:00:00 | 4950

sales\_2012\_europe | 10| 9519b | ITALY | 07-JUL-12 00:00:00 | 15000

sales\_2012\_asia | 20| 3788a | INDIA | 01-MAR-12 00:00:00 | 75000

sales\_2012\_asia | 20| 3788a | PAKISTAN | 04-JUN-12 00:00:00 | 37500

sales\_2012\_asia | 20| 4519b | INDIA | 02-DEC-12 00:00:00 | 5090

(8 rows)

While the rows have been removed, the structure of the 2012\_americas partition is still intact:

acctg=# SELECT subpartition\_name, high\_value FROM ALL\_TAB\_SUBPARTITIONS;

subpartition\_name | high\_value

-------------------+---------------------

2013\_europe | 'ITALY', 'FRANCE'

2012\_europe | 'ITALY', 'FRANCE'

2011\_europe | 'ITALY', 'FRANCE'

2013\_asia | 'PAKISTAN', 'INDIA'

2012\_asia | 'PAKISTAN', 'INDIA'

2011\_asia | 'PAKISTAN', 'INDIA'

2013\_americas | 'US', 'CANADA'

2012\_americas | 'US', 'CANADA'

2011\_americas | 'US', 'CANADA'

(9

rows)

## Handling Stray Values in a LIST or RANGE Partitioned Table

A DEFAULT or MAXVALUE partition or subpartition will capture any rows that do not meet the other partitioning rules defined for a table.

***Defining a DEFAULT Partition***

A DEFAULT partition will capture any rows that do not fit into any other partition in a LIST partitioned (or subpartitioned) table. If you do not include a DEFAULT rule, any row that does not match one of the values in the partitioning constraints will result in an error. Each LIST partition or subpartition may have its own DEFAULT rule.

The syntax of a DEFAULT rule is:

PARTITION [*partition*\_*name*] VALUES (DEFAULT)

Where *partition\_name* specifies the name of the partition or subpartition that will store any rows that do not match the rules specified for other partititions.

The last example created a list partitioned table in which the server decided which partition to store the data based upon the value of the country column. If you attempt to add a row in which the value of the country column contains a value not listed in the rules, Advanced Server reports an error:

acctg=# INSERT INTO sales VALUES

acctg-# (40, '3000x', 'IRELAND', '01-Mar-2012', '45000');

ERROR: inserted partition key does not map to any partition

The following example creates the same table, but adds a DEFAULT partition. The server will store any rows that do not match a value specified in the partitioning rules for europe, asia, or americas partitions in the others partition:

CREATE TABLE sales

(

dept\_no number,

part\_no varchar2,

country varchar2(20),

date date,

amount number

)

PARTITION BY LIST(country)

(

PARTITION europe VALUES('FRANCE', 'ITALY'),

PARTITION asia VALUES('INDIA', 'PAKISTAN'),

PARTITION americas VALUES('US', 'CANADA'),

PARTITION others VALUES (DEFAULT)

);

To test the DEFAULT partition, add row with a value in the country column that does not match one of the countries specified in the partitioning constraints:

INSERT INTO sales VALUES  
 (40, '3000x', 'IRELAND', '01-Mar-2012', '45000');

Querying the contents of the sales table confirms that the previously rejected row is now stored in the sales\_others partition:

acctg=# SELECT tableoid::regclass, \* FROM sales;

tableoid | dept\_no | part\_no | country | date | amount

----------------+---------+---------+----------+--------------------+--------

sales\_europe | 10 | 4519b | FRANCE | 17-JAN-12 00:00:00 | 45000

sales\_europe | 10 | 9519b | ITALY | 07-JUL-12 00:00:00 | 15000

sales\_europe | 10 | 9519a | FRANCE | 18-AUG-12 00:00:00 | 650000

sales\_europe | 10 | 9519b | FRANCE | 18-AUG-12 00:00:00 | 650000

sales\_asia | 20 | 3788a | INDIA | 01-MAR-12 00:00:00 | 75000

sales\_asia | 20 | 3788a | PAKISTAN | 04-JUN-12 00:00:00 | 37500

sales\_asia | 20 | 3788b | INDIA | 21-SEP-12 00:00:00 | 5090

sales\_asia | 20 | 4519a | INDIA | 18-OCT-12 00:00:00 | 650000

sales\_asia | 20 | 4519b | INDIA | 02-DEC-12 00:00:00 | 5090

sales\_americas | 40 | 9519b | US | 12-APR-12 00:00:00 | 145000

sales\_americas | 40 | 4577b | US | 11-NOV-12 00:00:00 | 25000

sales\_americas | 30 | 7588b | CANADA | 14-DEC-12 00:00:00 | 50000

sales\_americas | 30 | 9519b | CANADA | 01-FEB-12 00:00:00 | 75000

sales\_americas | 30 | 4519b | CANADA | 08-APR-12 00:00:00 | 120000

sales\_americas | 40 | 3788a | US | 12-MAY-12 00:00:00 | 4950

sales\_americas | 40 | 4788a | US | 23-SEP-12 00:00:00 | 4950

sales\_americas | 40 | 4788b | US | 09-OCT-12 00:00:00 | 15000

sales\_others | 40 | 3000x | IRELAND | 01-MAR-12 00:00:00 | 45000

(18 rows)

Please note that Advanced Server does not have a way to re-assign the contents of a DEFAULT partition or subpartition:

* You cannot use the ALTER TABLE… ADD PARTITION command to add a partition to a table with a DEFAULT rule, but you can use the ALTER TABLE… SPLIT PARTITION command to split an existing partition.
* You cannot use the ALTER TABLE… ADD SUBPARTITION command to add a subpartition to a table with a DEFAULT rule, but you can use the ALTER TABLE… SPLIT SUBPARTITION command to split an existing subpartition.

***Defining a MAXVALUE Partition***

A MAXVALUE partition (or subpartition) will capture any rows that do not fit into any other partition in a range-partitioned (or subpartitioned) table. If you do not include a MAXVALUE rule, any row that exceeds the maximum limit specified by the partitioning rules will result in an error. Each partition or subpartition may have its own MAXVALUE partition.

The syntax of a MAXVALUE rule is:

PARTITION [*partition*\_*name*] VALUES LESS THAN (MAXVALUE)

Where *partition\_name* specifies the name of the partition that will store any rows that do not match the rules specified for other partititions.

The last example created a range-partitioned table in which the data was partitioned based upon the value of the date column. If you attempt to add a row with a date that exceeds a date listed in the partitioning constraints, Advanced Server reports an error:

acctg=# INSERT INTO sales VALUES

acctg-# (40, '3000x', 'IRELAND', '01-Mar-2013', '45000');

ERROR: inserted partition key does not map to any partition

The following CREATE TABLE command creates the same table, but with a MAXVALUE partition. Instead of throwing an error, the server will store any rows that do not match the previous partitioning constraints in the others partition:

CREATE TABLE sales  
(  
 dept\_no number,  
 part\_no varchar2,  
 country varchar2(20),  
 date date,  
 amount number  
)  
PARTITION BY RANGE(date)  
(  
 PARTITION q1\_2012 VALUES LESS THAN('2012-Apr-01'),  
 PARTITION q2\_2012 VALUES LESS THAN('2012-Jul-01'),  
 PARTITION q3\_2012 VALUES LESS THAN('2012-Oct-01'),  
 PARTITION q4\_2012 VALUES LESS THAN('2013-Jan-01')  
 PARTITION others VALUES LESS THAN (MAXVALUE)  
);

To test the MAXVALUE partition, add a row with a value in the date column that exceeds the last date value listed in a partitioning rule. The server will store the row in the others partition:

INSERT INTO sales VALUES  
 (40, '3000x', 'IRELAND', '01-Mar-2013', '45000');

Querying the contents of the sales table confirms that the previously rejected row is now stored in the sales\_others partition :

acctg=# SELECT tableoid::regclass, \* FROM sales;

tableoid | dept\_no | part\_no | country | date | amount

---------------+---------+---------+----------+--------------------+---------

sales\_q1\_2012 | 10 | 4519b | FRANCE | 17-JAN-12 00:00:00 | 45000

sales\_q1\_2012 | 20 | 3788a | INDIA | 01-MAR-12 00:00:00 | 75000

sales\_q1\_2012 | 30 | 9519b | CANADA | 01-FEB-12 00:00:00 | 75000

sales\_q2\_2012 | 40 | 9519b | US | 12-APR-12 00:00:00 | 145000

sales\_q2\_2012 | 20 | 3788a | PAKISTAN | 04-JUN-12 00:00:00 | 37500

sales\_q2\_2012 | 30 | 4519b | CANADA | 08-APR-12 00:00:00 | 120000

sales\_q2\_2012 | 40 | 3788a | US | 12-MAY-12 00:00:00 | 4950

sales\_q3\_2012 | 10 | 9519b | ITALY | 07-JUL-12 00:00:00 | 15000

sales\_q3\_2012 | 10 | 9519a | FRANCE | 18-AUG-12 00:00:00 | 650000

sales\_q3\_2012 | 10 | 9519b | FRANCE | 18-AUG-12 00:00:00 | 650000

sales\_q3\_2012 | 20 | 3788b | INDIA | 21-SEP-12 00:00:00 | 5090

sales\_q3\_2012 | 40 | 4788a | US | 23-SEP-12 00:00:00 | 4950

sales\_q4\_2012 | 40 | 4577b | US | 11-NOV-12 00:00:00 | 25000

sales\_q4\_2012 | 30 | 7588b | CANADA | 14-DEC-12 00:00:00 | 50000

sales\_q4\_2012 | 40 | 4788b | US | 09-OCT-12 00:00:00 | 15000

sales\_q4\_2012 | 20 | 4519a | INDIA | 18-OCT-12 00:00:00 | 650000

sales\_q4\_2012 | 20 | 4519b | INDIA | 02-DEC-12 00:00:00 | 5090

sales\_others | 40 | 3000x | IRELAND | 01-MAR-13 00:00:00 | 45000

(18 rows)

Please note that Advanced Server does not have a way to re-assign the contents of a MAXVALUE partition or subpartition:

* You cannot use the ALTER TABLE… ADD PARTITION statement to add a partition to a table with a MAXVALUE rule, but you can use the ALTER TABLE… SPLIT PARTITION statement to split an existing partition.
* You cannot use the ALTER TABLE… ADD SUBPARTITION statement to add a subpartition to a table with a MAXVALUE rule , but you can split an existing subpartition with the ALTER TABLE… SPLIT SUBPARTITION statement.

## Specifying Multiple Partitioning Keys in a RANGE Partitioned Table

You can often improve performance by specifying multiple key columns for a RANGE partitioned table. If you often select rows using comparison operators (based on a greater-than or less-than value) on a small set of columns, consider using those columns in RANGE partitioning rules.

***Specifying Multiple Keys in a Range-Partitioned Table***

Range-partitioned table definitions may include multiple columns in the partitioning key. To specify multiple partitioning keys for a range-partitioned table, include the column names in a comma-separated list after the PARTITION BY RANGE clause:

CREATE TABLE sales

(

dept\_no number,

part\_no varchar2,

country varchar2(20),

sale\_year number,

sale\_month number,

sale\_day number,

amount number

)

PARTITION BY RANGE(sale\_year, sale\_month)

(

PARTITION q1\_2012   
 VALUES LESS THAN(2012, 4),

PARTITION q2\_2012   
 VALUES LESS THAN(2012, 7),

PARTITION q3\_2012   
 VALUES LESS THAN(2012, 10),

PARTITION q4\_2012   
 VALUES LESS THAN(2013, 1)

);

If a table is created with multiple partitioning keys, you must specify multiple key values when querying the table to take full advantage of partition pruning:

acctg=# EXPLAIN SELECT \* FROM sales WHERE sale\_year = 2012 AND sale\_month = 8;

QUERY PLAN

----------------------------------------------------------------------------- Result (cost=0.00..14.35 rows=2 width=250)

-> Append (cost=0.00..14.35 rows=2 width=250)

-> Seq Scan on sales (cost=0.00..0.00 rows=1 width=250)

Filter: ((sale\_year = 2012::numeric) AND (sale\_month = 8::numeric))

-> Seq Scan on sales\_q3\_2012 sales (cost=0.00..14.35 rows=1 width=250)

Filter: ((sale\_year = 2012::numeric) AND (sale\_month = 8::numeric))

(6 rows)

Since all rows with a value of 8 in the sale\_month column and a value of 2012 in the sale\_year column will be stored in the q3\_2012 partition, Advanced Server searches only that partition.

## Retrieving Information about a Partitioned Table

Advanced Server provides five system catalog views that you can use to view information about the structure of partitioned tables.

***Querying the Partitioning Views***

You can query the following views to retrieve information about partitioned and subpartitioned tables:

* ALL\_PART\_TABLES
* ALL\_TAB\_PARTITIONS
* ALL\_TAB\_SUBPARTITIONS
* ALL\_PART\_KEY\_COLUMNS
* ALL\_SUBPART\_KEY\_COLUMNS

The structure of each view is explained in [Section 13.5.1](#_Table_Partitioning_Views_3), *Table Partitioning Views*. If you are using the EDB-PSQL client, you can also discover the structure of a view by entering:

\d *view*\_*name*

Where *view\_name* specifies the name of the table partitioning view.

Querying a view can provide information about the structure of a partitioned or subpartitioned table. For example, the following code snippit displays the system-assigned names of a subpartitioned table:

acctg=# SELECT subpartition\_name, partition\_name FROM ALL\_TAB\_SUBPARTITIONS;

subpartition\_name | partition\_name

-------------------+----------------

SYS\_SUBP107 | americas

SYS\_SUBP104 | asia

SYS\_SUBP101 | europe

SYS\_SUBP108 | americas

SYS\_SUBP105 | asia

SYS\_SUBP102 | europe

SYS\_SUBP109 | americas

SYS\_SUBP106 | asia

SYS\_SUBP103 | europe

(9 rows)

### Table Partitioning Views - Reference

Query the following catalog views to review detailed information about your partitioned tables.

#### ALL\_PART\_TABLES

The following table lists the information available in the ALL\_PART\_TABLES view:

|  |  |  |
| --- | --- | --- |
| Column | Type | Description |
| owner | name | The owner of the table. |
| table\_name | name | The name of the table. |
| schema\_name | name | The schema in which the table resides. |
| partitioning\_type | text | RANGE, LIST or HASH |
| subpartitioning\_type | text | RANGE, LIST, HASH, or NONE |
| partition\_count | bigint | The number of partitions. |
| def\_subpartition\_count | integer | The default subpartition count - this will always be 0. |
| partitioning\_key\_count | integer | The number of columns listed in the partition by clause. |
| subpartitioning\_key\_count | integer | The number of columns in the subpartition by clause. |
| status | character varying(8) | This column will always be VALID. |
| def\_tablespace\_name | character varying(30) | This column will always be NULL. |
| def\_pct\_free | numeric | This column will always be NULL. |
| def\_pct\_used | numeric | This column will always be NULL. |
| def\_ini\_trans | numeric | This column will always be NULL. |
| def\_max\_trans | numeric | This column will always be NULL. |
| def\_initial\_extent | character varying(40) | This column will always be NULL. |
| def\_next\_extent | character varying(40) | This column will always be NULL. |
| def\_min\_extents | character varying(40) | This column will always be NULL. |
| def\_max\_extents | character varying(40) | This column will always be NULL. |
| def\_pct\_increase | character varying(40) | This column will always be NULL. |
| def\_freelists | numeric | This column will always be NULL. |
| def\_freelist\_groups | numeric | This column will always be NULL. |
| def\_logging | character varying(7) | This column will always be YES |
| def\_compression | character varying(8) | This column will always be NONE |
| def\_buffer\_pool | character varying(7) | This column will always be DEFAULT |
| ref\_ptn\_constraint\_name | character varying(30) | This column will always be NULL. |
| interval | character varying(1000) | This column will always be NULL. |

#### ALL\_TAB\_PARTITIONS

The following table lists the information available in the ALL\_TAB\_PARTITIONS view:

|  |  |  |
| --- | --- | --- |
| Column | Type | Description |
| table\_owner | name | The owner of the table. |
| table\_name | name | The name of the table. |
| schema\_name | name | The schema in which the table resides. |
| composite | text | YES if the table is subpartioned; NO if it is not subpartitioned. |
| partition\_name | name | The name of the partition. |
| subpartition\_count | bigint | The number of subpartitions for this partition. |
| high\_value | text | The partition limit for RANGE partitions, or the partition value for LIST partitions. |
| high\_value\_length | integer | The length of high\_value. |
| partition\_position | integer | The ordinal position of this partition. |
| tablespace\_name | name | The tablespace in which this partition resides. |
| pct\_free | numeric | This column will always be 0. |
| pct\_used | numeric | This column will always be 0. |
| ini\_trans | numeric | This column will always be 0. |
| max\_trans | numeric | This column will always be 0. |
| initial\_extent | numeric | This column will always be NULL. |
| next\_extent | numeric | This column will always be NULL. |
| min\_extent | numeric | This column will always be 0. |
| max\_extent | numeric | This column will always be 0. |
| pct\_increase | numeric | This column will always be 0. |
| freelists | numeric | This column will always be NULL. |
| freelist\_groups | numeric | This column will always be NULL. |
| logging | character varying(7) | This column will always be YES. |
| compression | character varying(8) | This column will always be NONE. |
| num\_rows | numeric | The approx. number of rows in this partition. |
| blocks | integer | The approx. number of blocks in this partition. |
| empty\_blocks | numeric | This column will always be NULL. |
| avg\_space | numeric | This column will always be NULL. |
| chain\_cnt | numeric | This column will always be NULL. |
| avg\_row\_len | numeric | This column will always be NULL. |
| sample\_size | numeric | This column will always be NULL. |
| last\_analyzed | timestamp without time zone | This column will always be NULL. |
| buffer\_pool | character varying(7) | This column will always be NULL. |
| global\_stats | character varying(3) | This column will always be YES. |
| user\_stats | character varying(3) | This column will always be NO. |
| backing\_table | regclass | OID of the backing table for this partition. |
| server\_name | name | The name of the server on which the partition resides. |

#### ALL\_TAB\_SUBPARTITIONS

The following table lists the information available in the ALL\_TAB\_SUBPARTITIONS view:

|  |  |  |
| --- | --- | --- |
| Column | Type | Description |
| table\_owner | name | The name of the owner of the table. |
| table\_name | name | The name of the table. |
| schema\_name | name | The name of the schema in which the table resides. |
| partition\_name | name | The name of the partition. |
| high\_value | text | The subpartition limit for RANGE, or the subpartition value for LIST subpartitions. |
| high\_value\_length | integer | The length of high\_value. |
| subpartition\_name | name | The name of the subpartition. |
| subpartition\_position | integer | The ordinal position of this subpartition. |
| tablespace\_name | name | The tablespace in which this subpartition resides. |
| pct\_free | numeric | This column will always be 0. |
| pct\_used | numeric | This column will always be 0. |
| ini\_trans | numeric | This column will always be 0. |
| max\_trans | numeric | This column will always be 0. |
| initial\_extent | numeric | This column will always be NULL. |
| next\_extent | numeric | This column will always be NULL. |
| min\_extent | numeric | This column will always be 0. |
| max\_extent | numeric | This column will always be 0. |
| pct\_increase | numeric | This column will always be 0. |
| freelists | numeric | This column will always be NULL. |
| freelist\_groups | numeric | This column will always be NULL. |
| logging | character varying(7) | This column will always be YES. |
| compression | character varying(8) | This column will always be NONE. |
| num\_rows | numeric | The approx. number of rows in this subpartition. |
| blocks | integer | The approx. number of blocks in this subpartition. |
| empty\_blocks | numeric | This column will always be NULL. |
| avg\_space | numeric | This column will always be NULL. |
| chain\_cnt | numeric | This column will always be NULL. |
| avg\_row\_len | numeric | This column will always be NULL. |
| sample\_size | numeric | This column will always be NULL. |
| last\_analyzed | timestamp without time zone | This column will always be NULL. |
| buffer\_pool | character varying(7) | This column will always be NULL. |
| global\_stats | character varying(3) | This column will always be YES. |
| user\_stats | character varying(3) | Thiscolumn will always be NO. |
| backing\_table | regclass | OID of the backing table for this subpartition. |
| server\_name | name | The name of the server on which the subpartition resides. |

#### ALL\_PART\_KEY\_COLUMNS

The following table lists the information available in the ALL\_PART\_KEY\_COLUMNS view:

|  |  |  |
| --- | --- | --- |
| Column | Type | Description |
| owner | name | The name of the table owner. |
| name | name | The name of the table. |
| schema | name | The name of the schema on which the table resides. |
| object\_type | character(5) | This column will always be TABLE. |
| column\_name | name | The name of the partitioning key column. |
| column\_position | integer | The position of this column within the partitioning key (the first column has a column position of 1, the second column has a column position of 2...) |

#### ALL\_SUBPART\_KEY\_COLUMNS

The following table lists the information available in the ALL\_SUBPART\_KEY\_COLUMNS view:

|  |  |  |
| --- | --- | --- |
| Column | Type | Description |
| owner | name | The name of the table owner. |
| name | name | The name of the table. |
| schema | name | The name of the schema on which the table resides. |
| object\_type | character(5) | This column will always be TABLE. |
| column\_name | name | The name of the partitioning key column. |
| column\_position | integer | The position of this column within the subpartitioning key (the first column has a column position of 1, the second column has a column position of 2...) |

# Security

The chapter describes various features providing for added security.

## Protecting Against SQL Injection Attacks

Advanced Server provides protection against SQL injection attacks. A SQL injection attack is an attempt to compromise a database by running SQL statements whose results provide clues to the attacker as to the content, structure, or security of that database.

Preventing a SQL injection attack is normally the responsibility of the application developer. The database administrator typically has little or no control over the potential threat. The difficulty for database administrators is that the application must have access to the data to function properly.

SQL/Protect is a module that allows a database administrator to protect a database from SQL injection attacks. SQL/Protect provides a layer of security in addition to the normal database security policies by examining incoming queries for common SQL injection profiles.

SQL/Protect gives the control back to the database administrator by alerting the administrator to potentially dangerous queries and by blocking these queries.

### SQL/Protect Overview

This section contains an introduction to the different types of SQL injection attacks and describes how SQL/Protect guards against them.

#### Types of SQL Injection Attacks

There are a number of different techniques used to perpetrate SQL injection attacks. Each technique is characterized by a certain signature. SQL/Protect examines queries for the following signatures:

Unauthorized Relations

While Advanced Server allows administrators to restrict access to relations (tables, views, etc.), many administrators do not perform this tedious task. SQL/Protect provides a learn mode that tracks the relations a user accesses.

This allows administrators to examine the workload of an application, and for SQL/Protect to learn which relations an application should be allowed to access for a given user or group of users in a role.

When SQL/Protect is switched to either passive or active mode, the incoming queries are checked against the list of learned relations.

Utility Commands

A common technique used in SQL injection attacks is to run utility commands, which are typically SQL Data Definition Language (DDL) statements. An example is creating a user-defined function that has the ability to access other system resources.

SQL/Protect can block the running of all utility commands, which are not normally needed during standard application processing.

SQL Tautology

The most frequent technique used in SQL injection attacks is issuing a tautological WHERE clause condition (that is, using a condition that is always true).

The following is an example:

WHERE password = 'x' OR 'x'='x'

Attackers will usually start identifying security weaknesses using this technique. SQL/Protect can block queries that use a tautological conditional clause.

Unbounded DML Statements

A dangerous action taken during SQL injection attacks is the running of unbounded DML statements. These are UPDATE and DELETE statements with no WHERE clause. For example, an attacker may update all users’ passwords to a known value or initiate a denial of service attack by deleting all of the data in a key table.

#### Monitoring SQL Injection Attacks

This section describes how SQL/Protect monitors and reports on SQL injection attacks.

##### Protected Roles

Monitoring for SQL injection attacks involves analyzing SQL statements originating in database sessions where the current user of the session is a protected role. A protected role is an Advanced Server user or group that the database administrator has chosen to monitor using SQL/Protect. (In Advanced Server, users and groups are collectively referred to as roles.)

Each protected role can be customized for the types of SQL injection attacks (discussed in Section 4.1.1.1) for which it is to be monitored, thus providing different levels of protection by role and significantly reducing the user maintenance load for DBAs.

Note: A role with the superuser privilege cannot be made a protected role. If a protected non-superuser role is subsequently altered to become a superuser, certain behaviors are exhibited whenever an attempt is made by that superuser to issue any command:

* A warning message is issued by SQL/Protect on every command issued by the protected superuser.
* The statistic in column superusers of edb\_sql\_protect\_stats is incremented with every command issued by the protected superuser. See Section 4.1.1.2.2 for information on the edb\_sql\_protect\_stats view.
* When SQL/Protect is in active mode, all commands issued by the protected superuser are prevented from running.

A protected role that has the superuser privilege should either be altered so that it is no longer a superuser, or it should be reverted back to an unprotected role.

##### Attack Attempt Statistics

Each usage of a command by a protected role that is considered an attack by SQL/Protect is recorded. Statistics are collected by type of SQL injection attack as discussed in Section 4.1.1.1.

These statistics are accessible from view edb\_sql\_protect\_stats that can be easily monitored to identify the start of a potential attack.

The columns in edb\_sql\_protect\_stats monitor the following:

* username. Name of the protected role.
* superusers. Number of SQL statements issued when the protected role is a superuser. In effect, any SQL statement issued by a protected superuser increases this statistic. See Section 4.1.1.2.1 for information on protected superusers.
* relations. Number of SQL statements issued referencing relations that were not learned by a protected role. (That is, relations that are not in a role’s protected relations list.)
* commands. Number of DDL statements issued by a protected role.
* tautology. Number of SQL statements issued by a protected role that contained a tautological condition.
* dml. Number of UPDATE and DELETE statements issued by a protected role that did not contain a WHERE clause.

This gives database administrators the opportunity to react proactively in preventing theft of valuable data or other malicious actions.

If a role is protected in more than one database, the role’s statistics for attacks in each database are maintained separately and are viewable only when connected to the respective database.

Note: SQL/Protect statistics are maintained in memory while the database server is running. When the database server is shut down, the statistics are saved to a binary file named edb\_sqlprotect.stat in the data/global subdirectory of the Advanced Server home directory.

##### Attack Attempt Queries

Each usage of a command by a protected role that is considered an attack by SQL/Protect is recorded in view edb\_sql\_protect\_queries.

View edb\_sql\_protect\_queries contains the following columns:

* username. Database user name of the attacker used to log into the database server.
* ip\_address. IP address of the machine from which the attack was initiated.
* port. Port number from which the attack originated.
* machine\_name. Name of the machine, if known, from which the attack originated.
* date\_time. Date and time at which the query was received by the database server. The time is stored to the precision of a minute.
* query. The query string sent by the attacker.

The maximum number of offending queries that are saved in edb\_sql\_protect\_queries is controlled by configuration parameter edb\_sql\_protect.max\_queries\_to\_save.

If a role is protected in more than one database, the role’s queries for attacks in each database are maintained separately and are viewable only when connected to the respective database.

### Configuring SQL/Protect

The library file (sqlprotect.so on Linux, sqlprotect.dll on Windows) necessary to run SQL/Protect should already be installed in the lib subdirectory of your Advanced Server home directory.

You will also need the SQL script file sqlprotect.sql located in the share/contrib subdirectory of your Advanced Server home directory.

You must configure the database server to use SQL/Protect, and you must configure each database that you want SQL/Protect to monitor:

* The database server configuration file, postgresql.conf, must be modified by adding and enabling configuration parameters used by SQL/Protect.
* Database objects used by SQL/Protect must be installed in each database that you want SQL/Protect to monitor.

Step 1: Edit the following configuration parameters in the postgresql.conf file located in the data subdirectory of your Advanced Server home directory.

* shared\_preload\_libraries. Add $libdir/sqlprotect to the list of libraries.
* edb\_sql\_protect.enabled. Controls whether or not SQL/Protect is actively monitoring protected roles by analyzing SQL statements issued by those roles and reacting according to the setting of edb\_sql\_protect.level. When you are ready to begin monitoring with SQL/Protect set this parameter to on. If this parameter is omitted, the default is off.
* edb\_sql\_protect.level. Sets the action taken by SQL/Protect when a SQL statement is issued by a protected role. If this parameter is omitted, the default behavior is passive. Initially, set this parameter to learn. See Section 4.1.2.1.2 for further explanation of this parameter.
* edb\_sql\_protect.max\_protected\_roles. Sets the maximum number of roles that can be protected. If this parameter is omitted, the default setting is 64. See Section 2.1.3.12.8 for information on the maximum range of this parameter.
* edb\_sql\_protect.max\_protected\_relations. Sets the maximum number of relations that can be protected per role. If this parameter is omitted, the default setting is 1024.

Please note the total number of protected relations for the server will be the number of protected relations times the number of protected roles. Every protected relation consumes space in shared memory. The space for the maximum possible protected relations is reserved during database server startup.

See Section 2.1.3.12.7 for information about the maximum range of this parameter.

* edb\_sql\_protect.max\_queries\_to\_save. Sets the maximum number of offending queries to save in the edb\_sql\_protect\_queries view. If this parameter is omitted, the default setting is 5000. If the number of offending queries reaches the limit, additional queries are not saved in the view, but are accessible in the database server log file. Note: The minimum valid value for this parameter is 100. If a value less than 100 is specified, the database server starts using the default setting of 5000. A warning message is recorded in the database server log file. See Section 2.1.3.12.9 for information on the maximum range of this parameter.

The following example shows the settings of these parameters in the postgresql.conf file:

shared\_preload\_libraries = '$libdir/dbms\_pipe,$libdir/edb\_gen,$libdir/sqlprotect'

# (change requires restart)

.

.

.

edb\_sql\_protect.enabled = off

edb\_sql\_protect.level = learn

edb\_sql\_protect.max\_protected\_roles = 64

edb\_sql\_protect.max\_protected\_relations = 1024

edb\_sql\_protect.max\_queries\_to\_save = 5000

Step 2: Restart the database server after you have modified the postgresql.conf file.

For Linux only: Run the /etc/init.d/ppas-9.5 script with the restart option as shown by the following:

$ su root

Password:

$ /etc/init.d/ppas-9.5 restart

Restarting Postgres Plus Advanced Server 9.5:

waiting for server to shut down.... done

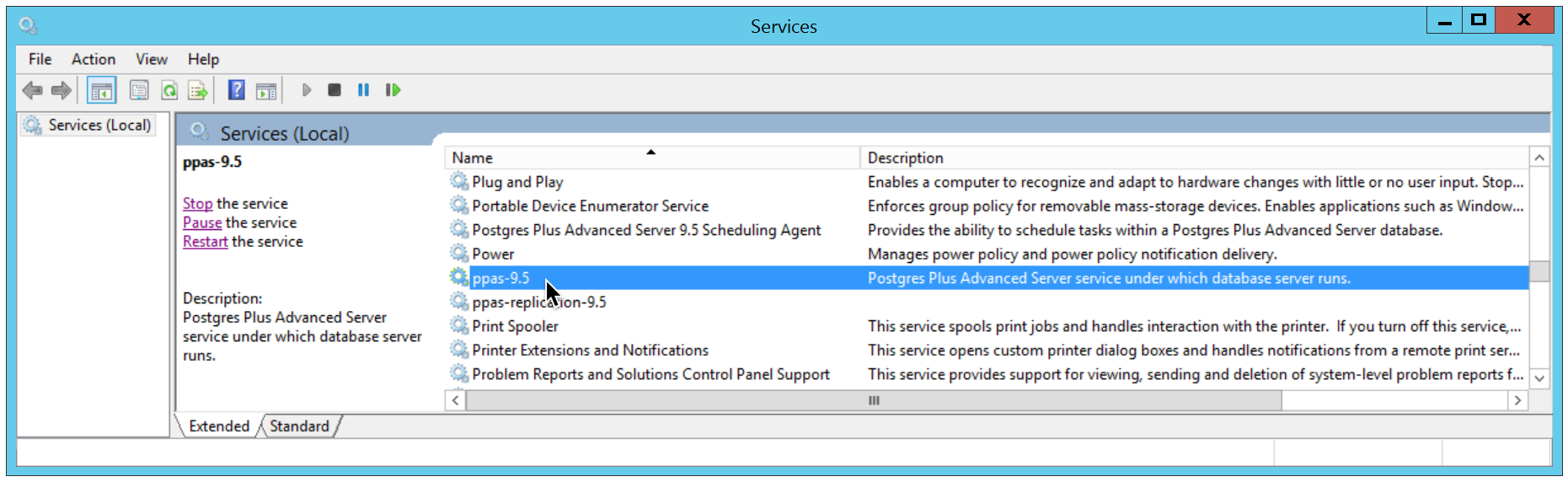
server stopped

waiting for server to start.... done

server started

Postgres Plus Advanced Server 9.5 restarted successfully

For Windows only: Open Control Panel, Administrative Tools, and then Services. Restart the service named ppas-9.5.



*Figure 4.2 - Starting the ppas-9.5 service.*

Step 3: For each database that you want to protect from SQL injection attacks, connect to the database as a superuser (either enterprisedb or postgres, depending upon your installation options) and run the script sqlprotect.sql located in the share/contrib subdirectory of your Advanced Server home directory. The script creates the SQL/Protect database objects in a schema named sqlprotect.

The following example shows this process to set up protection for a database named edb:

$ /opt/PostgresPlus/9.5AS/bin/psql -d edb -U enterprisedb

Password for user enterprisedb:

psql.bin (9.5.0.0)

Type "help" for help.

edb=# \i /opt/PostgresPlus/9.5AS/share/contrib/sqlprotect.sql

CREATE SCHEMA

GRANT

SET

CREATE TABLE

GRANT

CREATE TABLE

GRANT

CREATE FUNCTION

CREATE FUNCTION

CREATE FUNCTION

CREATE FUNCTION

CREATE FUNCTION

CREATE FUNCTION

CREATE FUNCTION

DO

CREATE FUNCTION

CREATE FUNCTION

DO

CREATE VIEW

GRANT

DO

CREATE VIEW

GRANT

CREATE VIEW

GRANT

CREATE FUNCTION

CREATE FUNCTION

SET

#### Selecting Roles to Protect

After the SQL/Protect database objects have been created in a database, you select the roles for which SQL queries are to be monitored for protection, and the level of protection.

##### Setting the Protected Roles List

For each database that you want to protect, you must determine the roles you want to monitor and then add those roles to the protected roles list of that database.

Step 1: Connect as a superuser to a database that you wish to protect using either psql or Postgres Enterprise Manager Client.

$ /opt/PostgresPlus/9.5AS/bin/psql -d edb -U enterprisedb

Password for user enterprisedb:

psql.bin (9.5.0.0)

Type "help" for help.

edb=#

Step 2: Since the SQL/Protect tables, functions, and views are built under the sqlprotect schema, use the SET search\_path command to include the sqlprotect schema in your search path. This eliminates the need to schema-qualify any operation or query involving SQL/Protect database objects.

edb=# SET search\_path TO sqlprotect;

SET

Step 3: Each role that you wish to protect must be added to the protected roles list. This list is maintained in the table edb\_sql\_protect.

To add a role, use the function protect\_role('rolename').

The following example protects a role named appuser.

edb=# SELECT protect\_role('appuser');

protect\_role

--------------

(1 row)

You can list the roles that have been added to the protected roles list by issuing the following query:

edb=# SELECT \* FROM edb\_sql\_protect;

dbid | roleid | protect\_relations | allow\_utility\_cmds | allow\_tautology | allow\_empty\_dml

-------+--------+-------------------+--------------------+-----------------+-----------------

13917 | 16671 | t | f | f | f

(1 row)

A view is also provided that gives the same information using the object names instead of the Object Identification numbers (OIDs).

edb=# \x

Expanded display is on.

edb=# SELECT \* FROM list\_protected\_users;

-[ RECORD 1 ]------+--------

dbname | edb

username | appuser

protect\_relations | t

allow\_utility\_cmds | f

allow\_tautology | f

allow\_empty\_dml | f

##### Setting the Protection Level

Configuration parameter edb\_sql\_protect.level sets the protection level, which defines the behavior of SQL/Protect when a protected role issues a SQL statement. The defined behavior applies to all roles in the protected roles lists of all databases configured with SQL/Protect in the database server.

In the postgresql.conf file the edb\_sql\_protect.level configuration parameter can be set to one of the following values to use either learn mode, passive mode, or active mode:

* learn. Tracks the activities of protected roles and records the relations used by the roles. This is used when initially configuring SQL/Protect so the expected behaviors of the protected applications are learned.
* passive. Issues warnings if protected roles are breaking the defined rules, but does not stop any SQL statements from executing. This is the next step after SQL/Protect has learned the expected behavior of the protected roles. This essentially behaves in intrusion detection mode and can be run in production when properly monitored.
* active. Stops all invalid statements for a protected role. This behaves as a SQL firewall preventing dangerous queries from running. This is particularly effective against early penetration testing when the attacker is trying to determine the vulnerability point and the type of database behind the application. Not only does SQL/Protect close those vulnerability points, but it tracks the blocked queries allowing administrators to be alerted before the attacker finds an alternate method of penetrating the system.

If the edb\_sql\_protect.level parameter is not set or is omitted from the configuration file, the default behavior of SQL/Protect is passive.

If you are using SQL/Protect for the first time, set edb\_sql\_protect.level to learn.

#### Monitoring Protected Roles

Once you have configured SQL/Protect in a database, added roles to the protected roles list, and set the desired protection level, you can then activate SQL/Protect in one of learn mode, passive mode, or active mode. You can then start running your applications.

With a new SQL/Protect installation, the first step is to determine the relations that protected roles should be permitted to access during normal operation. Learn mode allows a role to run applications during which time SQL/Protect is recording the relations that are accessed. These are added to the role’s protected relations list stored in table edb\_sql\_protect\_rel.

Monitoring for protection against attack begins when SQL/Protect is run in passive or active mode. In passive and active modes, the role is permitted to access the relations in its protected relations list as these were determined to be the relations the role should be able to access during typical usage.

However, if a role attempts to access a relation that is not in its protected relations list, a WARNING or ERROR severity level message is returned by SQL/Protect. The role’s attempted action on the relation may or may not be carried out depending upon whether the mode is passive or active.

##### Learn Mode

Step 1: To activate SQL/Protect in learn mode, set the following parameters in the postgresql.conf file as shown below:

edb\_sql\_protect.enabled = on

edb\_sql\_protect.level = learn

Step 2: Reload the postgresql.conf file.

Choose Expert Configuration, then Reload Configuration from the Advanced Server application menu.

Note: For an alternative method of reloading the configuration file, use the pg\_reload\_conf function. Be sure you are connected to a database as a superuser and execute function pg\_reload\_conf as shown by the following example:

edb=# SELECT pg\_reload\_conf();

pg\_reload\_conf

----------------

t

(1 row)

Step 3: Allow the protected roles to run their applications.

As an example the following queries are issued in the psql application by protected role appuser:

edb=> SELECT \* FROM dept;

NOTICE: SQLPROTECT: Learned relation: 16384

deptno | dname | loc

--------+------------+----------

10 | ACCOUNTING | NEW YORK

20 | RESEARCH | DALLAS

30 | SALES | CHICAGO

40 | OPERATIONS | BOSTON

(4 rows)

edb=> SELECT empno, ename, job FROM emp WHERE deptno = 10;

NOTICE: SQLPROTECT: Learned relation: 16391

empno | ename | job

-------+--------+-----------

7782 | CLARK | MANAGER

7839 | KING | PRESIDENT

7934 | MILLER | CLERK

(3 rows)

SQL/Protect generates a NOTICE severity level message indicating the relation has been added to the role’s protected relations list.

In SQL/Protect learn mode, SQL statements that are cause for suspicion are not prevented from executing, but a message is issued to alert the user to potentially dangerous statements as shown by the following example:

edb=> CREATE TABLE appuser\_tab (f1 INTEGER);

NOTICE: SQLPROTECT: This command type is illegal for this user

CREATE TABLE

edb=> DELETE FROM appuser\_tab;

NOTICE: SQLPROTECT: Learned relation: 16672

NOTICE: SQLPROTECT: Illegal Query: empty DML

DELETE 0

Step 4: As a protected role runs applications, the SQL/Protect tables can be queried to observe the addition of relations to the role’s protected relations list.

Connect as a superuser to the database you are monitoring and set the search path to include the sqlprotect schema.

edb=# SET search\_path TO sqlprotect;

SET

Query the edb\_sql\_protect\_rel table to see the relations added to the protected relations list:

edb=# SELECT \* FROM edb\_sql\_protect\_rel;

dbid | roleid | relid

-------+--------+-------

13917 | 16671 | 16384

13917 | 16671 | 16391

13917 | 16671 | 16672

(3 rows)

The view list\_protected\_rels is provided that gives more comprehensive information along with the object names instead of the OIDs.

edb=# SELECT \* FROM list\_protected\_rels;

Database | Protected User | Schema | Name | Type | Owner

----------+----------------+--------+-------------+-------+--------------

edb | appuser | public | dept | Table | enterprisedb

edb | appuser | public | emp | Table | enterprisedb

edb | appuser | public | appuser\_tab | Table | appuser

(3 rows)

##### Passive Mode

Once you have determined that a role’s applications have accessed all relations they will need, you can now change the protection level so that SQL/Protect can actively monitor the incoming SQL queries and protect against SQL injection attacks.

Passive mode is the less restrictive of the two protection modes, passive and active.

Step 1: To activate SQL/Protect in passive mode, set the following parameters in the postgresql.conf file as shown below:

edb\_sql\_protect.enabled = on

edb\_sql\_protect.level = passive

Step 2: Reload the configuration file as shown in Step 2 of Section 4.1.2.2.1.

Now SQL/Protect is in passive mode. For relations that have been learned such as the dept and emp tables of the prior examples, SQL statements are permitted with no special notification to the client by SQL/Protect as shown by the following queries run by user appuser:

edb=> SELECT \* FROM dept;

deptno | dname | loc

--------+------------+----------

10 | ACCOUNTING | NEW YORK

20 | RESEARCH | DALLAS

30 | SALES | CHICAGO

40 | OPERATIONS | BOSTON

(4 rows)

edb=> SELECT empno, ename, job FROM emp WHERE deptno = 10;

empno | ename | job

-------+--------+-----------

7782 | CLARK | MANAGER

7839 | KING | PRESIDENT

7934 | MILLER | CLERK

(3 rows)

SQL/Protect does not prevent any SQL statement from executing, but issues a message of WARNING severity level for SQL statements executed against relations that were not learned, or for SQL statements that contain a prohibited signature as shown in the following example:

edb=> CREATE TABLE appuser\_tab\_2 (f1 INTEGER);

WARNING: SQLPROTECT: This command type is illegal for this user

CREATE TABLE

edb=> INSERT INTO appuser\_tab\_2 VALUES (1);

WARNING: SQLPROTECT: Illegal Query: relations

INSERT 0 1

edb=> INSERT INTO appuser\_tab\_2 VALUES (2);

WARNING: SQLPROTECT: Illegal Query: relations

INSERT 0 1

edb=> SELECT \* FROM appuser\_tab\_2 WHERE 'x' = 'x';

WARNING: SQLPROTECT: Illegal Query: relations

WARNING: SQLPROTECT: Illegal Query: tautology

f1

----

1

2

(2 rows)

Step 3: Monitor the statistics for suspicious activity.

By querying the view edb\_sql\_protect\_stats, you can see the number of times SQL statements were executed that referenced relations that were not in a role’s protected relations list, or contained SQL injection attack signatures. See Section 4.1.1.2.2 for more information on view edb\_sql\_protect\_stats.

The following is a query on edb\_sql\_protect\_stats:

edb=# SET search\_path TO sqlprotect;

SET

edb=# SELECT \* FROM edb\_sql\_protect\_stats;

username | superusers | relations | commands | tautology | dml

----------+------------+-----------+----------+-----------+-----

appuser | 0 | 3 | 1 | 1 | 0

(1 row)

Step 4: View information on specific attacks.

By querying the view edb\_sql\_protect\_queries, you can see the SQL statements that were executed that referenced relations that were not in a role’s protected relations list, or contained SQL injection attack signatures. See Section 4.1.1.2.3 for more information on view edb\_sql\_protect\_queries.

The following is a query on edb\_sql\_protect\_queries:

edb=# SELECT \* FROM edb\_sql\_protect\_queries;

-[ RECORD 1 ]+---------------------------------------------

username | appuser

ip\_address |

port |

machine\_name |

date\_time | 20-JUN-14 13:21:00 -04:00

query | INSERT INTO appuser\_tab\_2 VALUES (1);

-[ RECORD 2 ]+---------------------------------------------

username | appuser

ip\_address |

port |

machine\_name |

date\_time | 20-JUN-14 13:21:00 -04:00

query | CREATE TABLE appuser\_tab\_2 (f1 INTEGER);

-[ RECORD 3 ]+---------------------------------------------

username | appuser

ip\_address |

port |

machine\_name |

date\_time | 20-JUN-14 13:22:00 -04:00

query | INSERT INTO appuser\_tab\_2 VALUES (2);

-[ RECORD 4 ]+---------------------------------------------

username | appuser

ip\_address |

port |

machine\_name |

date\_time | 20-JUN-14 13:22:00 -04:00

query | SELECT \* FROM appuser\_tab\_2 WHERE 'x' = 'x';

Note: The ip\_address and port columns do not return any information if the attack originated on the same host as the database server using the Unix-domain socket (that is, pg\_hba.conf connection type local).

##### Active Mode

In active mode, disallowed SQL statements are prevented from executing. Also, the message issued by SQL/Protect has a higher severity level of ERROR instead of WARNING.

Step 1: To activate SQL/Protect in active mode, set the following parameters in the postgresql.conf file as shown below:

edb\_sql\_protect.enabled = on

edb\_sql\_protect.level = active

Step 2: Reload the configuration file as shown in Step 2 of Section 4.1.2.2.1.

The following example illustrates SQL statements similar to those given in the examples of Step 2 in Section 4.1.2.2.2, but executed by user appuser when edb\_sql\_protect.level is set to active:

edb=> CREATE TABLE appuser\_tab\_3 (f1 INTEGER);

ERROR: SQLPROTECT: This command type is illegal for this user

edb=> INSERT INTO appuser\_tab\_2 VALUES (1);

ERROR: SQLPROTECT: Illegal Query: relations

edb=> SELECT \* FROM appuser\_tab\_2 WHERE 'x' = 'x';

ERROR: SQLPROTECT: Illegal Query: relations

The following shows the resulting statistics:

edb=# SELECT \* FROM sqlprotect.edb\_sql\_protect\_stats;

username | superusers | relations | commands | tautology | dml

----------+------------+-----------+----------+-----------+-----

appuser | 0 | 5 | 2 | 1 | 0

(1 row)

The following is a query on edb\_sql\_protect\_queries:

edb=# SELECT \* FROM sqlprotect.edb\_sql\_protect\_queries;

-[ RECORD 1 ]+---------------------------------------------

username | appuser

ip\_address |

port |

machine\_name |

date\_time | 20-JUN-14 13:21:00 -04:00

query | CREATE TABLE appuser\_tab\_2 (f1 INTEGER);

-[ RECORD 2 ]+---------------------------------------------

username | appuser

ip\_address |

port |

machine\_name |

date\_time | 20-JUN-14 13:22:00 -04:00

query | INSERT INTO appuser\_tab\_2 VALUES (2);

-[ RECORD 3 ]+---------------------------------------------

username | appuser

ip\_address | 192.168.2.6

port | 50098

machine\_name |

date\_time | 20-JUN-14 13:39:00 -04:00

query | CREATE TABLE appuser\_tab\_3 (f1 INTEGER);

-[ RECORD 4 ]+---------------------------------------------

username | appuser

ip\_address | 192.168.2.6

port | 50098

machine\_name |

date\_time | 20-JUN-14 13:39:00 -04:00

query | INSERT INTO appuser\_tab\_2 VALUES (1);

-[ RECORD 5 ]+---------------------------------------------

username | appuser

ip\_address | 192.168.2.6

port | 50098

machine\_name |

date\_time | 20-JUN-14 13:39:00 -04:00

query | SELECT \* FROM appuser\_tab\_2 WHERE 'x' = 'x';

### Common Maintenance Operations

The following describes how to perform other common operations.

You must be connected as a superuser to perform these operations and have included schema sqlprotect in your search path.

#### Adding a Role to the Protected Roles List

To add a role to the protected roles list run protect\_role('rolename').

protect\_role('rolename')

This is shown by the following example:

edb=# SELECT protect\_role('newuser');

protect\_role

--------------

(1 row)

#### Removing a Role From the Protected Roles List

To remove a role from the protected roles list use either of the following functions:

unprotect\_role('rolename')

unprotect\_role(roleoid)

Note: The variation of the function using the OID is useful if you remove the role using the DROP ROLE or DROP USER SQL statement before removing the role from the protected roles list. If a query on a SQL/Protect relation returns a value such as unknown (OID=16458) for the user name, use the unprotect\_role(roleoid) form of the function to remove the entry for the deleted role from the protected roles list.

Removing a role using these functions also removes the role’s protected relations list.

The statistics for a role that has been removed are not deleted until you use the drop\_stats function as described in Section 4.1.3.5.

The offending queries for a role that has been removed are not deleted until you use the drop\_queries function as described in Section 4.1.3.6.

The following is an example of the unprotect\_role function:

edb=# SELECT unprotect\_role('newuser');

unprotect\_role

----------------

(1 row)

Alternatively, the role could be removed by giving its OID of 16693:

edb=# SELECT unprotect\_role(16693);

unprotect\_role

----------------

(1 row)

#### Setting the Types of Protection for a Role

You can change whether or not a role is protected from a certain type of SQL injection attack.

Change the Boolean value for the column in edb\_sql\_protect corresponding to the type of SQL injection attack for which protection of a role is to be disabled or enabled.

Be sure to qualify the following columns in your WHERE clause of the statement that updates edb\_sql\_protect:

* dbid. OID of the database for which you are making the change
* roleid. OID of the role for which you are changing the Boolean settings

For example, to allow a given role to issue utility commands, update the allow\_utility\_cmds column as follows:

UPDATE edb\_sql\_protect SET allow\_utility\_cmds = TRUE WHERE dbid = 13917 AND roleid = 16671;

You can verify the change was made by querying edb\_sql\_protect or list\_protected\_users. In the following query note that column allow\_utility\_cmds now contains t.

edb=# SELECT dbid, roleid, allow\_utility\_cmds FROM edb\_sql\_protect;

dbid | roleid | allow\_utility\_cmds

-------+--------+--------------------

13917 | 16671 | t

(1 row)

The updated rules take effect on new sessions started by the role since the change was made.

#### Removing a Relation From the Protected Relations List

If SQL/Protect has learned that a given relation is accessible for a given role, you can subsequently remove that relation from the role’s protected relations list.

Delete its entry from the edb\_sql\_protect\_rel table using any of the following functions:

unprotect\_rel('rolename', 'relname')

unprotect\_rel('rolename', 'schema', 'relname')

unprotect\_rel(roleoid, reloid)

If the relation given by relname is not in your current search path, specify the relation’s schema using the second function format.

The third function format allows you to specify the OIDs of the role and relation, respectively, instead of their text names.

The following example illustrates the removal of the public.emp relation from the protected relations list of the role appuser.

edb=# SELECT unprotect\_rel('appuser', 'public', 'emp');

unprotect\_rel

---------------

(1 row)

The following query shows there is no longer an entry for the emp relation.

edb=# SELECT \* FROM list\_protected\_rels;

Database | Protected User | Schema | Name | Type | Owner

----------+----------------+--------+-------------+-------+--------------

edb | appuser | public | dept | Table | enterprisedb

edb | appuser | public | appuser\_tab | Table | appuser

(2 rows)

SQL/Protect will now issue a warning or completely block access (depending upon the setting of edb\_sql\_protect.level) whenever the role attempts to utilize that relation.

#### Deleting Statistics

You can delete statistics from view edb\_sql\_protect\_stats using either of the two following functions:

drop\_stats('rolename')

drop\_stats(roleoid)

Note: The variation of the function using the OID is useful if you remove the role using the DROP ROLE or DROP USER SQL statement before deleting the role’s statistics using drop\_stats('rolename'). If a query on edb\_sql\_protect\_stats returns a value such as unknown (OID=16458) for the user name, use the drop\_stats(roleoid) form of the function to remove the deleted role’s statistics from edb\_sql\_protect\_stats.

The following is an example of the drop\_stats function:

edb=# SELECT drop\_stats('appuser');

drop\_stats

------------

(1 row)

edb=# SELECT \* FROM edb\_sql\_protect\_stats;

username | superusers | relations | commands | tautology | dml

----------+------------+-----------+----------+-----------+-----

(0 rows)

The following is an example of using the drop\_stats(roleoid) form of the function when a role is dropped before deleting its statistics:

edb=# SELECT \* FROM edb\_sql\_protect\_stats;

username | superusers | relations | commands | tautology | dml

---------------------+------------+-----------+----------+-----------+-----

unknown (OID=16693) | 0 | 5 | 3 | 1 | 0

appuser | 0 | 5 | 2 | 1 | 0

(2 rows)

edb=# SELECT drop\_stats(16693);

drop\_stats

------------

(1 row)

edb=# SELECT \* FROM edb\_sql\_protect\_stats;

username | superusers | relations | commands | tautology | dml

----------+------------+-----------+----------+-----------+-----

appuser | 0 | 5 | 2 | 1 | 0

(1 row)

#### Deleting Offending Queries

You can delete offending queries from view edb\_sql\_protect\_queries using either of the two following functions:

drop\_queries('rolename')

drop\_queries(roleoid)

Note: The variation of the function using the OID is useful if you remove the role using the DROP ROLE or DROP USER SQL statement before deleting the role’s offending queries using drop\_queries('rolename'). If a query on edb\_sql\_protect\_queries returns a value such as unknown (OID=16454) for the user name, use the drop\_queries(roleoid) form of the function to remove the deleted role’s offending queries from edb\_sql\_protect\_queries.

The following is an example of the drop\_queries function:

edb=# SELECT drop\_queries('appuser');

drop\_queries

--------------

5

(1 row)

edb=# SELECT \* FROM edb\_sql\_protect\_queries;

username | ip\_address | port | machine\_name | date\_time | query

----------+------------+------+--------------+-----------+-------

(0 rows)

The following is an example of using the drop\_queries(roleoid) form of the function when a role is dropped before deleting its queries:

edb=# SELECT username, query FROM edb\_sql\_protect\_queries;

username | query

---------------------+----------------------------------------------

unknown (OID=16454) | CREATE TABLE appuser\_tab\_2 (f1 INTEGER);

unknown (OID=16454) | INSERT INTO appuser\_tab\_2 VALUES (2);

unknown (OID=16454) | CREATE TABLE appuser\_tab\_3 (f1 INTEGER);

unknown (OID=16454) | INSERT INTO appuser\_tab\_2 VALUES (1);

unknown (OID=16454) | SELECT \* FROM appuser\_tab\_2 WHERE 'x' = 'x';

(5 rows)

edb=# SELECT drop\_queries(16454);

drop\_queries

--------------

5

(1 row)

edb=# SELECT \* FROM edb\_sql\_protect\_queries;

username | ip\_address | port | machine\_name | date\_time | query

----------+------------+------+--------------+-----------+-------

(0 rows)

#### Disabling and Enabling Monitoring

If you wish to turn off SQL/Protect monitoring once you have enabled it, perform the following steps:

Step 1: Set the configuration parameter edb\_sql\_protect.enabled to off in the postgresql.conf file.

The entry for edb\_sql\_protect.enabled should look like the following:

edb\_sql\_protect.enabled = off

Step 2: Reload the configuration file as shown in Step 2 of Section 4.1.2.2.1.

To re-enable SQL/Protect monitoring perform the following steps:

Step 1: Set the configuration parameter edb\_sql\_protect.enabled to on in the postgresql.conf file.

The entry for edb\_sql\_protect.enabled should look like the following:

edb\_sql\_protect.enabled = on

Step 2: Reload the configuration file as shown in Step 2 of Section 4.1.2.2.1.

### Backing Up and Restoring a SQL/Protect Database

Backing up a database that is configured with SQL/Protect, and then restoring the backup file to a new database require additional considerations to what is normally associated with backup and restore procedures. This is primarily due to the use of Object Identification numbers (OIDs) in the SQL/Protect tables as explained in this section.

Note: This section is applicable if your backup and restore procedures result in the re-creation of database objects in the new database with new OIDs such as is the case when using the pg\_dump backup program.

If you are backing up your Advanced Server database server by simply using the operating system’s copy utility to create a binary image of the Advanced Server data files (file system backup method), then this section does not apply.

#### Object Identification Numbers in SQL/Protect Tables

SQL/Protect uses two tables, edb\_sql\_protect and edb\_sql\_protect\_rel, to store information on database objects such as databases, roles, and relations. References to these database objects in these tables are done using the objects’ OIDs, and not the objects’ text names. The OID is a numeric data type used by Advanced Server to uniquely identify each database object.

When a database object is created, Advanced Server assigns an OID to the object, which is then used whenever a reference is needed to the object in the database catalogs. If you create the same database object in two databases, such as a table with the same CREATE TABLE statement, each table is assigned a different OID in each database.

In a backup and restore operation that results in the re-creation of the backed up database objects, the restored objects end up with different OIDs in the new database than what they were assigned in the original database. As a result, the OIDs referencing databases, roles, and relations stored in the edb\_sql\_protect and edb\_sql\_protect\_rel tables are no longer valid when these tables are simply dumped to a backup file and then restored to a new database.

The following sections describe two functions, export\_sqlprotect and import\_sqlprotect, that are used specifically for backing up and restoring SQL/Protect tables in order to ensure the OIDs in the SQL/Protect tables reference the correct database objects after the SQL/Protect tables are restored.

#### Backing Up the Database

The following are the steps to back up a database that has been configured with SQL/Protect.

Step 1: Create a backup file using pg\_dump.

The following example shows a plain-text backup file named /tmp/edb.dmp created from database edb using the pg\_dump utility program:

$ cd /opt/PostgresPlus/9.5AS/bin

$ ./pg\_dump -U enterprisedb -Fp -f /tmp/edb.dmp edb

Password:

$

Step 2: Connect to the database as a superuser and export the SQL/Protect data using the export\_sqlprotect('sqlprotect\_file') function where sqlprotect\_file is the fully qualified path to a file where the SQL/Protect data is to be saved.

The enterprisedb operating system account (postgres if you installed Advanced Server in PostgreSQL compatibility mode) must have read and write access to the directory specified in sqlprotect\_file.

edb=# SELECT sqlprotect.export\_sqlprotect('/tmp/sqlprotect.dmp');

export\_sqlprotect

-------------------

(1 row)

The files /tmp/edb.dmp and /tmp/sqlprotect.dmp comprise your total database backup.

#### Restoring From the Backup Files

Step 1: Restore the backup file to the new database.

The following example uses the psql utility program to restore the plain-text backup file /tmp/edb.dmp to a newly created database named newdb:

$ /opt/PostgresPlus/9.5AS/bin/psql -d newdb -U enterprisedb -f /tmp/edb.dmp

Password for user enterprisedb:

SET

SET

SET

SET

SET

COMMENT

CREATE SCHEMA

.

.

.

Step 2: Connect to the new database as a superuser and delete all rows from the edb\_sql\_protect\_rel table.

This step removes any existing rows in the edb\_sql\_protect\_rel table that were backed up from the original database. These rows do not contain the correct OIDs relative to the database where the backup file has been restored.

$ /opt/PostgresPlus/9.5AS/bin/psql -d newdb -U enterprisedb

Password for user enterprisedb:

psql.bin (9.5.0.0)

Type "help" for help.

newdb=# DELETE FROM sqlprotect.edb\_sql\_protect\_rel;

DELETE 2

Step 3: Delete all rows from the edb\_sql\_protect table.

This step removes any existing rows in the edb\_sql\_protect table that were backed up from the original database. These rows do not contain the correct OIDs relative to the database where the backup file has been restored.

newdb=# DELETE FROM sqlprotect.edb\_sql\_protect;

DELETE 1

Step 4: Delete any statistics that may exist for the database.

This step removes any existing statistics that may exist for the database to which you are restoring the backup. The following query displays any existing statistics:

newdb=# SELECT \* FROM sqlprotect.edb\_sql\_protect\_stats;

username | superusers | relations | commands | tautology | dml

----------+------------+-----------+----------+-----------+-----

(0 rows)

For each row that appears in the preceding query, use the drop\_stats function specifying the role name of the entry.

For example, if a row appeared with appuser in the username column, issue the following command to remove it:

newdb=# SELECT sqlprotect.drop\_stats('appuser');

drop\_stats

------------

(1 row)

Step 5: Delete any offending queries that may exist for the database.

This step removes any existing queries that may exist for the database to which you are restoring the backup. The following query displays any existing queries:

edb=# SELECT \* FROM sqlprotect.edb\_sql\_protect\_queries;

username | ip\_address | port | machine\_name | date\_time | query

----------+------------+------+--------------+-----------+-------

(0 rows)

For each row that appears in the preceding query, use the drop\_queries function specifying the role name of the entry.

For example, if a row appeared with appuser in the username column, issue the following command to remove it:

edb=# SELECT sqlprotect.drop\_queries('appuser');

drop\_queries

--------------

(1 row)

Step 6: Make sure the role names that were protected by SQL/Protect in the original database exist in the database server where the new database resides.

If the original and new databases reside in the same database server, then nothing needs to be done assuming you have not deleted any of these roles from the database server.

Step 7: Run the function import\_sqlprotect('sqlprotect\_file') where sqlprotect\_file is the fully qualified path to the file you created in Step 2 of Section 4.1.4.2.

newdb=# SELECT sqlprotect.import\_sqlprotect('/tmp/sqlprotect.dmp');

import\_sqlprotect

-------------------

(1 row)

Tables edb\_sql\_protect and edb\_sql\_protect\_rel are now populated with entries containing the OIDs of the database objects as assigned in the new database. The statistics view edb\_sql\_protect\_stats also now displays the statistics imported from the original database.

The SQL/Protect tables and statistics are now properly restored for this database. This is verified by the following queries on the Advanced Server system catalogs:

newdb=# SELECT datname, oid FROM pg\_database;

datname | oid

-----------+-------

template1 | 1

template0 | 13909

edb | 13917

newdb | 16679

(4 rows)

newdb=# SELECT rolname, oid FROM pg\_roles;

rolname | oid

--------------+-------

enterprisedb | 10

appuser | 16671

newuser | 16678

(3 rows)

newdb=# SELECT relname, oid FROM pg\_class WHERE relname IN ('dept','emp','appuser\_tab');

relname | oid

-------------+-------

appuser\_tab | 16803

dept | 16809

emp | 16812

(3 rows)

newdb=# SELECT \* FROM sqlprotect.edb\_sql\_protect;

dbid | roleid | protect\_relations | allow\_utility\_cmds | allow\_tautology | allow\_empty\_dml

-------+--------+-------------------+--------------------+-----------------+-----------------

16679 | 16671 | t | t | f | f

(1 row)

newdb=# SELECT \* FROM sqlprotect.edb\_sql\_protect\_rel;

dbid | roleid | relid

-------+--------+-------

16679 | 16671 | 16809

16679 | 16671 | 16803

(2 rows)

newdb=# SELECT \* FROM sqlprotect.edb\_sql\_protect\_stats;

username | superusers | relations | commands | tautology | dml

----------+------------+-----------+----------+-----------+-----

appuser | 0 | 5 | 2 | 1 | 0

(1 row)

newedb=# \x

Expanded display is on.

nwedb=# SELECT \* FROM sqlprotect.edb\_sql\_protect\_queries;

-[ RECORD 1 ]+---------------------------------------------

username | appuser

ip\_address |

port |

machine\_name |

date\_time | 20-JUN-14 13:21:00 -04:00

query | CREATE TABLE appuser\_tab\_2 (f1 INTEGER);

-[ RECORD 2 ]+---------------------------------------------

username | appuser

ip\_address |

port |

machine\_name |

date\_time | 20-JUN-14 13:22:00 -04:00

query | INSERT INTO appuser\_tab\_2 VALUES (2);

-[ RECORD 3 ]+---------------------------------------------

username | appuser

ip\_address | 192.168.2.6

port | 50098

machine\_name |

date\_time | 20-JUN-14 13:39:00 -04:00

query | CREATE TABLE appuser\_tab\_3 (f1 INTEGER);

-[ RECORD 4 ]+---------------------------------------------

username | appuser

ip\_address | 192.168.2.6

port | 50098

machine\_name |

date\_time | 20-JUN-14 13:39:00 -04:00

query | INSERT INTO appuser\_tab\_2 VALUES (1);

-[ RECORD 5 ]+---------------------------------------------

username | appuser

ip\_address | 192.168.2.6

port | 50098

machine\_name |

date\_time | 20-JUN-14 13:39:00 -04:00

query | SELECT \* FROM appuser\_tab\_2 WHERE 'x' = 'x';

Note the following about the columns in tables edb\_sql\_protect and edb\_sql\_protect\_rel:

* dbid. Matches the value in the oid column from pg\_database for newdb
* roleid. Matches the value in the oid column from pg\_roles for appuser

Also note that in table edb\_sql\_protect\_rel, the values in the relid column match the values in the oid column of pg\_class for relations dept and appuser\_tab.

Step 8: Verify that the SQL/Protect configuration parameters are set as desired in the postgresql.conf file for the database server running the new database as described throughout sections 0, 4.1.2.1, and 4.1.2.2. Restart the database server or reload the configuration file as appropriate.

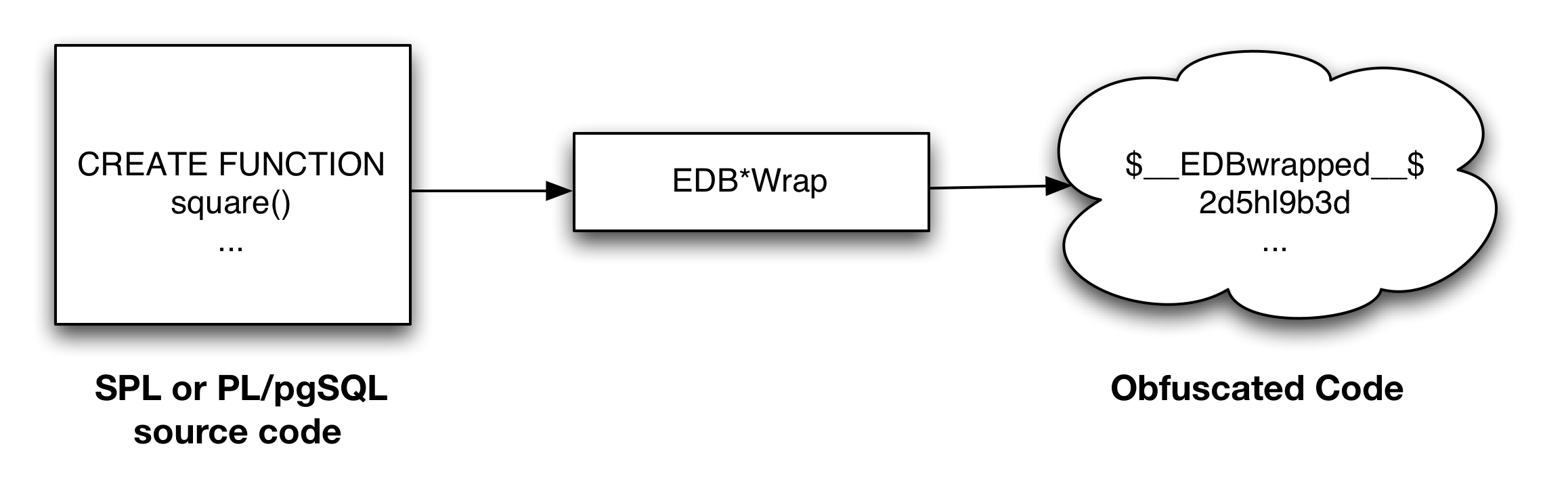
You can now monitor the database using SQL/Protect.

## EDB\*Wrap

The EDB\*Wrap utility protects proprietary source code and programs (functions, stored procedures, triggers, and packages) from unauthorized scrutiny. The EDB\*Wrap program translates a file that contains SPL or PL/pgSQL source code (the plaintext) into a file that contains the same code in a form that is nearly impossible to read. Once you have the obfuscated form of the code, you can send that code to Advanced Server and it will store those programs in obfuscated form. While EDB\*Wrap does obscure code, table definitions are still exposed.

Everything you wrap is stored in obfuscated form.  If you wrap an entire package, the package body source, as well as the prototypes contained in the package header and the functions and procedures contained in the package body are stored in obfuscated form.

If you wrap a CREATE PACKAGE statement, you hide the package API from other developers.  You may want to wrap the package body, but not the package header so users can see the package prototypes and other public variables that are defined in the package body.  To allow users to see what prototypes the package contains, use EDB\*Wrap to obfuscate only  the 'CREATE PACKAGE BODY' statement in the edbwrap input file, omitting the 'CREATE PACKAGE' statement. The package header source will be stored plaintext, while the package body source and package functions and procedures will be stored obfuscated.



Once wrapped, source code and programs cannot be unwrapped or debugged. Reverse engineering is possible, but would be very difficult.

The entire source file is wrapped into one unit. Any psql meta-commands included in the wrapped file will not be recognized when the file is executed; executing an obfuscated file that contains a psql meta-command will cause a syntax error. edbwrap does not validate SQL source code - if the plaintext form contains a syntax error, edbwrap will not complain. Instead, the server will report an error and abort the entire file when you try to execute the obfuscated form.

### Using EDB\*Wrap to Obfuscate Source Code

EDB\*Wrap is a command line utility; it accepts a single input source file, obfuscates the contents and returns a single output file. When you invoke the edbwrap utility, you must provide the name of the file that contains the source code to obfuscate. You may also specify the name of the file where edbwrap will write the obfuscated form of the code. edbwrap offers three different command-line styles. The first style is shown by the following:

edbwrap iname=*input\_file* [oname=*output\_file*]

The iname=input\_file argument specifies the name of the input file; if input\_file does not contain an extension, edbwrap will search for a file named *input\_file*.sql

The oname=*output\_file* argument (which is optional) specifies the name of the output file; if *output\_file* does not contain an extension, edbwrap will append .plb to the name.

If you do not specify an output file name, edbwrap writes to a file whose name is derived from the input file name: edbwrap strips the suffix (typically .sql) from the input file name and adds .plb.

edbwrap offers two other command-line styles that may feel more familiar:

edbwrap --iname *input\_file* [--oname *output\_file*]

edbwrap -i *input\_file* [-o *output\_file*]

You may mix command-line styles; the rules for deriving input and output file names are identical regardless of which style you use.

Once edbwrap has produced a file that contains obfuscated code, you typically feed that file into Advanced Server using a client application such as edb-psql. The server executes the obfuscated code line by line and stores the source code for SPL and PL/pgSQL programs in wrapped form.

In summary, to obfuscate code with EDB\*Wrap, you:

* Create the source code file.
* Invoke EDB\*Wrap to obfuscate the code.
* Import the file as if it were in plaintext form.

The following sequence demonstrates edbwrap functionality.

First, create the source code for the list\_emp function (in plaintext form):

$ cat listemp.sql

CREATE OR REPLACE FUNCTION list\_emp() RETURNS VOID

AS $$

DECLARE

v\_empno NUMERIC(4);

v\_ename VARCHAR(10);

emp\_cur CURSOR FOR

SELECT empno, ename FROM emp ORDER BY empno;

BEGIN

OPEN emp\_cur;

RAISE INFO 'EMPNO ENAME';

RAISE INFO '----- -------';

LOOP

FETCH emp\_cur INTO v\_empno, v\_ename;

EXIT WHEN NOT FOUND;

RAISE INFO '% %', v\_empno, v\_ename;

END LOOP;

CLOSE emp\_cur;

RETURN;

END;

$$ LANGUAGE 'plpgsql';

You can import the list\_emp function with a client application such as psql:

$ psql -d edb -U enterprisedb

Password for user enterprisedb:

psql.bin (9.5.0.0)

Type "help" for help.

edb=# \i listemp.sql

CREATE FUNCTION

You can view the plaintext source code (stored in the server) by examining the pg\_function system table:

edb=# SELECT funsrc FROM pg\_function WHERE funname = 'list\_emp';

funsrc

------------------------------------------------------

+

DECLARE +

v\_empno NUMERIC(4); +

v\_ename VARCHAR(10); +

emp\_cur CURSOR FOR +

SELECT empno, ename FROM emp ORDER BY empno;+

BEGIN +

OPEN emp\_cur; +

RAISE INFO 'EMPNO ENAME'; +

RAISE INFO '----- -------'; +

LOOP +

FETCH emp\_cur INTO v\_empno, v\_ename; +

EXIT WHEN NOT FOUND; +

RAISE INFO '% %', v\_empno, v\_ename; +

END LOOP; +

CLOSE emp\_cur; +

RETURN; +

END; +

(1 row)

Next, obfuscate the plaintext file with EDB\*Wrap:

$ edbwrap -i listemp.sql

EDB\*Wrap Utility: Release 9.5.0.0

Copyright (c) 2004-2016, EnterpriseDB Corporation. All Rights Reserved.

Using encoding UTF8 for input

Processing listemp.sql to listemp.plb

$

$ cat listemp.plb

$\_\_EDBwrapped\_\_$

UTF8

d6UEwTa69kFnCNAFVOgJqNyQMH7HwCn8dPPFJlkMFSb6YB4meTCGpIIoBnhYpcnxtAU+ZJMAu0Xe

WOTKG5iU9jpjqlwuioYVNa4EHFrf5JtNRTSL8tWhbi78li8ET5SWdU9eSGZiOfSVGi43b21ZWuGc

F8a342iMTy0bozbdl0r1dYku/f2kHnMYoBCi6EukmHik3j/iO1mJp06GHH71FG7BCOgCSW6L4B4x

BDje0MMVbBJveYyHWxBH12Bi8p4KDGy1HDLC8MK9S9EbfKPJbwKPZK37J8Ci9fhWBorfrTtz1k2f

vO1UKaTZGkYH0MIFvcZw6BG24dFL1kH5E2Rk5x4RzRsV2Hm+2LwTuDexs8hgleA3sPB/oZF9umb2

hZYkT5v1Ja7cKBnowdJrJNj/DOFoJcI1pFgG3DgJ

$\_\_EDBwrapped\_\_$

You may notice that the second line of the wrapped file contains an encoding name (in this case, the encoding is UTF8). When you obfuscate a file, edbwrap infers the encoding of the input file by examining the locale. For example, if you are running edbwrap while your locale is set to en\_US.utf8, edbwrap assumes that the input file is encoded in UTF8. Be sure to examine the output file after running edbwrap; if the locale contained in the wrapped file does not match the encoding of the input file, you should change your locale and rewrap the input file.

You can import the obfuscated code into Advanced Server using the same tools that work with plaintext code:

$ psql -d edb -U enterprisedb

Password for user enterprisedb:

psql.bin (9.5.0.0)

Type "help" for help.

edb=# \i listemp.plb

CREATE FUNCTION

edb=# SELECT funsrc FROM pg\_function WHERE funname = 'list\_emp';

funsrc

------------------------------------------------------------------------------

+

$\_\_EDBwrapped\_\_$ +

UTF8 +

d6UNH3OTrROsTCLF6NKWq5gWsZxi5giSpg6SmNgWDqHutT8OqqpJZnL5wNtaBxs4B6+inA6qeWCA+

QsTKvmcDNHk3yFneWI33Jeo/DsdVqkIEMrlUsu2ogymEJedHcM1YQFARyx+l0mWBI+yqixE4BNZw+

jSeqiVKAhAckek8JzL9pf0QLFT8TTzzTG61KN7iFQQii0B6C4/GpDlZCmC5oDXt94PR15YcZ5fJq+

p+UThN/uahwIaDu+FQ2AhSxNCxJH1aqjJEnwE9S7jsRvQXQ/yRt4zc7WbfeQMhhLA0E9w+hOy3aS+

CKb6bHF3pVVQLiG6tWpjdWwgTZ7neG+T1EounZC8bKwn +

$\_\_EDBwrapped\_\_$

(1 row)

Invoke the obfuscated code in the same way that you would invoke the plaintext form:

edb=# SELECT list\_emp();

INFO: EMPNO ENAME

INFO: ----- -------

INFO: 7369 SMITH

INFO: 7499 ALLEN

INFO: 7521 WARD

INFO: 7566 JONES

INFO: 7654 MARTIN

INFO: 7698 BLAKE

INFO: 7782 CLARK

INFO: 7788 SCOTT

INFO: 7839 KING

INFO: 7844 TURNER

INFO: 7876 ADAMS

INFO: 7900 JAMES

INFO: 7902 FORD

INFO: 7934 MILLER

list\_emp

----------

(1 row)

When you use pg\_dump to back up a database, wrapped programs remain obfuscated in the archive file.

Be aware that audit logs produced by Advanced Server will show wrapped programs in plaintext form. Source code is also displayed in plaintext in SQL error messages generated during the execution of a program.

Note: At this time, the bodies of the objects created by the following statements will not be stored in obfuscated form:

CREATE [OR REPLACE] TYPE *type\_name* AS OBJECT

CREATE [OR REPLACE] TYPE *type\_name* UNDER *type\_name*

CREATE [OR REPLACE] TYPE BODY *type\_name*

## Virtual Private Database

Virtual Private Database is a type of fine-grained access control using security policies. Fine-grained access control in Virtual Private Database means that access to data can be controlled down to specific rows as defined by the security policy.

The rules that encode a security policy are defined in a policy function, which is an SPL function with certain input parameters and return value. The security policy is the named association of the policy function to a particular database object, typically a table.

Note: In Advanced Server, the policy function can be written in any language supported by Advanced Server such as SQL and PL/pgSQL in addition to SPL.

Note: The database objects currently supported by Advanced Server Virtual Private Database are tables. Policies cannot be applied to views or synonyms.

The advantages of using Virtual Private Database are the following:

* Provides a fine-grained level of security. Database object level privileges given by the GRANT command determine access privileges to the entire instance of a database object, while Virtual Private Database provides access control for the individual rows of a database object instance.
* A different security policy can be applied depending upon the type of SQL command (INSERT, UPDATE, DELETE, or SELECT).
* The security policy can vary dynamically for each applicable SQL command affecting the database object depending upon factors such as the session user of the application accessing the database object.
* Invocation of the security policy is transparent to all applications that access the database object and thus, individual applications do not have to be modified to apply the security policy.
* Once a security policy is enabled, it is not possible for any application (including new applications) to circumvent the security policy except by the system privilege noted by the following.
* Even superusers cannot circumvent the security policy except by the system privilege noted by the following.

Note: The only way security policies can be circumvented is if the EXEMPT ACCESS POLICY system privilege has been granted to a user. The EXEMPT ACCESS POLICY privilege should be granted with extreme care as a user with this privilege is exempted from all policies in the database.

The DBMS\_RLS package provides procedures to create policies, remove policies, enable policies, and disable policies. See Section 9.11 for details on using the DBMS\_RLS package.

# EDB Resource Manager

EDB Resource Manager is an Advanced Server feature that provides the capability to control the usage of operating system resources used by Advanced Server processes.

This capability allows you to protect the system from processes that may uncontrollably overuse and monopolize certain system resources.

The following are some key points about using EDB Resource Manager.

* The basic component of EDB Resource Manager is a resource group. A resource group is a named, global group, available to all databases in an Advanced Server instance, on which various resource usage limits can be defined. Advanced Server processes that are assigned as members of a given resource group are then controlled by EDB Resource Manager so that the aggregate resource usage of all processes in the group is kept near the limits defined on the group.
* Data definition language commands are used to create, alter, and drop resource groups. These commands can only be used by a database user with superuser privileges.
* The desired, aggregate consumption level of all processes belonging to a resource group is defined by resource type parameters. There are different resource type parameters for the different types of system resources currently supported by EDB Resource Manager.
* Multiple resource groups can be created, each with different settings for its resource type parameters, thus defining different consumption levels for each resource group.
* EDB Resource Manager throttles processes in a resource group to keep resource consumption near the limits defined by the resource type parameters. If there are multiple resource type parameters with defined settings in a resource group, the actual resource consumption may be significantly lower for certain resource types than their defined resource type parameter settings. This is because EDB Resource Manager throttles processes attempting to keep all resources with defined resource type settings within their defined limits.
* The definition of available resource groups and their resource type settings are stored in a shared global system catalog. Thus, resource groups can be utilized by all databases in a given Advanced Server instance.
* The edb\_max\_resource\_groups configuration parameter sets the maximum number of resource groups that can be active simultaneously with running processes. The default setting is 16 resource groups. Changes to this parameter take effect on database server restart.
* Use the SET edb\_resource\_group TO group\_name command to assign the current process to a specified resource group. Use the RESET edb\_resource\_group command or SET edb\_resource\_group TO DEFAULT to remove the current process from a resource group.
* A default resource group can be assigned to a role using the ALTER ROLE ... SET command, or to a database by the ALTER DATABASE ... SET command. The entire database server instance can be assigned a default resource group by setting the parameter in the postgresql.conf file.
* In order to include resource groups in a backup file of the database server instance, use the pg\_dumpall backup utility with default settings (That is, do not specify any of the --globals-only, --roles-only, or --tablespaces-only options.)

## Creating and Managing Resource Groups

The data definition language commands described in this section provide for the creation and management of resource groups.

### CREATE RESOURCE GROUP

Use the CREATE RESOURCE GROUP command to create a new resource group.

CREATE RESOURCE GROUP *group*\_*name*;

**Description**

The CREATE RESOURCE GROUP command creates a resource group with the specified name. Resource limits can then be defined on the group with the ALTER RESOURCE GROUP command. The resource group is accessible from all databases in the Advanced Server instance.

To use the CREATE RESOURCE GROUP command you must have superuser privileges.

**Parameters**

group\_name

The name of the resource group.

**Example**

The following example results in the creation of three resource groups named resgrp\_a, resgrp\_b, and resgrp\_c.

edb=# CREATE RESOURCE GROUP resgrp\_a;

CREATE RESOURCE GROUP

edb=# CREATE RESOURCE GROUP resgrp\_b;

CREATE RESOURCE GROUP

edb=# CREATE RESOURCE GROUP resgrp\_c;

CREATE RESOURCE GROUP

The following query shows the entries for the resource groups in the edb\_resource\_group catalog.

edb=# SELECT \* FROM edb\_resource\_group;

rgrpname | rgrpcpuratelimit | rgrpdirtyratelimit

----------+------------------+--------------------

resgrp\_a | 0 | 0

resgrp\_b | 0 | 0

resgrp\_c | 0 | 0

(3 rows)

### ALTER RESOURCE GROUP

Use the ALTER RESOURCE GROUP command to change the attributes of an existing resource group. The command syntax comes in three forms.

The first form renames the resource group:

ALTER RESOURCE GROUP *group\_name* RENAME TO *new*\_*name*;

The second form assigns a resource type to the resource group:

ALTER RESOURCE GROUP *group\_name* SET

*resource\_type* { TO | = } { *value* | DEFAULT };

The third form resets the assignment of a resource type to its default within the group:

ALTER RESOURCE GROUP *group*\_*name* RESET *resource\_type*;

**Description**

The ALTER RESOURCE GROUP command changes certain attributes of an existing resource group.

The first form with the RENAME TO clause assigns a new name to an existing resource group.

The second form with the SET resource\_type TO clause either assigns the specified literal value to a resource type, or resets the resource type when DEFAULT is specified. Resetting or setting a resource type to DEFAULT means that the resource group has no defined limit on that resource type.

The third form with the RESET resource\_type clause resets the resource type for the group as described previously.

To use the ALTER RESOURCE GROUP command you must have superuser privileges.

**Parameters**

group\_name

The name of the resource group to be altered.

new\_name

The new name to be assigned to the resource group.

resource\_type

The resource type parameter specifying the type of resource to which a usage value is to be set.

value | DEFAULT

When value is specified, the literal value to be assigned to resource\_type. When DEFAULT is specified, the assignment of resource\_type is reset for the resource group.

**Example**

The following are examples of the ALTER RESOURCE GROUP command.

edb=# ALTER RESOURCE GROUP resgrp\_a RENAME TO newgrp;

ALTER RESOURCE GROUP

edb=# ALTER RESOURCE GROUP resgrp\_b SET cpu\_rate\_limit = .5;

ALTER RESOURCE GROUP

edb=# ALTER RESOURCE GROUP resgrp\_b SET dirty\_rate\_limit = 6144;

ALTER RESOURCE GROUP

edb=# ALTER RESOURCE GROUP resgrp\_c RESET cpu\_rate\_limit;

ALTER RESOURCE GROUP

The following query shows the results of the ALTER RESOURCE GROUP commands to the entries in the edb\_resource\_group catalog.

edb=# SELECT \* FROM edb\_resource\_group;

rgrpname | rgrpcpuratelimit | rgrpdirtyratelimit

----------+------------------+--------------------

newgrp | 0 | 0

resgrp\_b | 0.5 | 6144

resgrp\_c | 0 | 0

(3 rows)

### DROP RESOURCE GROUP

Use the DROP RESOURCE GROUP command to remove a resource group.

DROP RESOURCE GROUP [ IF EXISTS ] *group*\_*name*;

**Description**

The DROP RESOURCE GROUP command removes a resource group with the specified name.

To use the DROP RESOURCE GROUP command you must have superuser privileges.

**Parameters**

group\_name

The name of the resource group to be removed.

IF EXISTS

Do not throw an error if the resource group does not exist. A notice is issued in this case.

**Example**

The following example removes resource group newgrp.

edb=# DROP RESOURCE GROUP newgrp;

DROP RESOURCE GROUP

### Assigning a Process to a Resource Group

Use the SET edb\_resource\_group TO group\_name command to assign the current process to a specified resource group as shown by the following.

edb=# SET edb\_resource\_group TO resgrp\_b;

SET

edb=# SHOW edb\_resource\_group;

edb\_resource\_group

--------------------

resgrp\_b

(1 row)

The resource type settings of the group immediately take effect on the current process. If the command is used to change the resource group assigned to the current process, the resource type settings of the newly assigned group immediately take effect.

Processes can be included by default in a resource group by assigning a default resource group to roles, databases, or an entire database server instance.

A default resource group can be assigned to a role using the ALTER ROLE ... SET command. For more information about the ALTER ROLE command, please refer to the PostgreSQL core documentation available at:

<http://www.postgresql.org/docs/9.5/static/sql-alterrole.html>

A default resource group can be assigned to a database by the ALTER DATABASE ... SET command. For more information about the ALTER DATABASE command, please refer to the PostgreSQL core documentation available at:

<http://www.postgresql.org/docs/9.5/static/sql-alterdatabase.html>

The entire database server instance can be assigned a default resource group by setting the edb\_resource\_group configuration parameter in the postgresql.conf file as shown by the following.

# - EDB Resource Manager -

#edb\_max\_resource\_groups = 16 # 0-65536 (change requires restart)

edb\_resource\_group = 'resgrp\_b'

A change to edb\_resource\_group in the postgresql.conf file requires a configuration file reload before it takes effect on the database server instance.

### Removing a Process from a Resource Group

Set edb\_resource\_group to DEFAULT or use RESET edb\_resource\_group to remove the current process from a resource group as shown by the following.

edb=# SET edb\_resource\_group TO DEFAULT;

SET

edb=# SHOW edb\_resource\_group;

edb\_resource\_group

--------------------

(1 row)

For removing a default resource group from a role, use the ALTER ROLE ... RESET form of the ALTER ROLE command.

For removing a default resource group from a database, use the ALTER DATABASE ... RESET form of the ALTER DATABASE command.

For removing a default resource group from the database server instance, set the edb\_resource\_group configuration parameter to an empty string in the postgresql.conf file and reload the configuration file.

### Monitoring Processes in Resource Groups

After resource groups have been created, the number of processes actively using these resource groups can be obtained from the view edb\_all\_resource\_groups.

The columns in edb\_all\_resource\_groups are the following:

* group\_name. Name of the resource group.
* active\_processes. Number of active processes in the resource group.
* cpu\_rate\_limit. The value of the CPU rate limit resource type assigned to the resource group.
* per\_process\_cpu\_rate\_limit. The CPU rate limit applicable to an individual, active process in the resource group.
* dirty\_rate\_limit. The value of the dirty rate limit resource type assigned to the resource group.
* per\_process\_dirty\_rate\_limit. The dirty rate limit applicable to an individual, active process in the resource group.

Note: Columns per\_process\_cpu\_rate\_limit and per\_process\_dirty\_rate\_limit do not show the actual resource consumption used by the processes, but indicate how EDB Resource Manager sets the resource limit for an individual process based upon the number of active processes in the resource group.

The following shows edb\_all\_resource\_groups when resource group resgrp\_a contains no active processes, resource group resgrp\_b contains two active processes, and resource group resgrp\_c contains one active process.

edb=# SELECT \* FROM edb\_all\_resource\_groups ORDER BY group\_name;

-[ RECORD 1 ]----------------+------------------

group\_name | resgrp\_a

active\_processes | 0

cpu\_rate\_limit | 0.5

per\_process\_cpu\_rate\_limit |

dirty\_rate\_limit | 12288

per\_process\_dirty\_rate\_limit |

-[ RECORD 2 ]----------------+------------------

group\_name | resgrp\_b

active\_processes | 2

cpu\_rate\_limit | 0.4

per\_process\_cpu\_rate\_limit | 0.195694289022895

dirty\_rate\_limit | 6144

per\_process\_dirty\_rate\_limit | 3785.92924684337

-[ RECORD 3 ]----------------+------------------

group\_name | resgrp\_c

active\_processes | 1

cpu\_rate\_limit | 0.3

per\_process\_cpu\_rate\_limit | 0.292342129631091

dirty\_rate\_limit | 3072

per\_process\_dirty\_rate\_limit | 3072

The CPU rate limit and dirty rate limit settings that are assigned to these resource groups are as follows.

edb=# SELECT \* FROM edb\_resource\_group;

rgrpname | rgrpcpuratelimit | rgrpdirtyratelimit

----------+------------------+--------------------

resgrp\_a | 0.5 | 12288

resgrp\_b | 0.4 | 6144

resgrp\_c | 0.3 | 3072

(3 rows)

In the edb\_all\_resource\_groups view, note that the per\_process\_cpu\_rate\_limit and per\_process\_dirty\_rate\_limit values are roughly the corresponding CPU rate limit and dirty rate limit divided by the number of active processes.

## CPU Usage Throttling

CPU usage of a resource group is controlled by setting the cpu\_rate\_limit resource type parameter.

Set the cpu\_rate\_limit parameter to the fraction of CPU time over wall-clock time to which the combined, simultaneous CPU usage of all processes in the group should not exceed. Thus, the value assigned to cpu\_rate\_limit should typically be less than or equal to 1.

The valid range of the cpu\_rate\_limit parameter is 0 to 1.67772e+07. A setting of 0 means no CPU rate limit has been set for the resource group.

When multiplied by 100, the cpu\_rate\_limit can also be interpreted as the CPU usage percentage for a resource group.

EDB Resource Manager utilizes CPU throttling to keep the aggregate CPU usage of all processes in the group within the limit specified by the cpu\_rate\_limit parameter. A process in the group may be interrupted and put into sleep mode for a short interval of time to maintain the defined limit. When and how such interruptions occur is defined by a proprietary algorithm used by EDB Resource Manager.

### Setting the CPU Rate Limit for a Resource Group

The ALTER RESOURCE GROUP command with the SET cpu\_rate\_limit clause is used to set the CPU rate limit for a resource group.

In the following example the CPU usage limit is set to 50% for resgrp\_a, 40% for resgrp\_b and 30% for resgrp\_c. This means that the combined CPU usage of all processes assigned to resgrp\_a is maintained at approximately 50%. Similarly, for all processes in resgrp\_b, the combined CPU usage is kept to approximately 40%, etc.

edb=# ALTER RESOURCE GROUP resgrp\_a SET cpu\_rate\_limit TO .5;

ALTER RESOURCE GROUP

edb=# ALTER RESOURCE GROUP resgrp\_b SET cpu\_rate\_limit TO .4;

ALTER RESOURCE GROUP

edb=# ALTER RESOURCE GROUP resgrp\_c SET cpu\_rate\_limit TO .3;

ALTER RESOURCE GROUP

The following query shows the settings of cpu\_rate\_limit in the catalog.

edb=# SELECT rgrpname, rgrpcpuratelimit FROM edb\_resource\_group;

rgrpname | rgrpcpuratelimit

----------+------------------

resgrp\_a | 0.5

resgrp\_b | 0.4

resgrp\_c | 0.3

(3 rows)

Changing the cpu\_rate\_limit of a resource group not only affects new processes that are assigned to the group, but any currently running processes that are members of the group are immediately affected by the change. That is, if the cpu\_rate\_limit is changed from .5 to .3, currently running processes in the group would be throttled downward so that the aggregate group CPU usage would be near 30% instead of 50%.

To illustrate the effect of setting the CPU rate limit for resource groups, the following examples use a CPU-intensive calculation of 20000 factorial (multiplication of 20000 \* 19999 \* 19998, etc.) performed by the query SELECT 20000!; run in the psql command line utility.

The resource groups with the CPU rate limit settings shown in the previous query are used in these examples.

### Example – Single Process in a Single Group

The following shows that the current process is set to use resource group resgrp\_b. The factorial calculation is then started.

edb=# SET edb\_resource\_group TO resgrp\_b;

SET

edb=# SHOW edb\_resource\_group;

edb\_resource\_group

--------------------

resgrp\_b

(1 row)

edb=# SELECT 20000!;

In a second session, the Linux top command is used to display the CPU usage as shown under the %CPU column. The following is a snapshot at an arbitrary point in time as the top command output periodically changes.

$ top

top - 16:37:03 up 4:15, 7 users, load average: 0.49, 0.20, 0.38

Tasks: 202 total, 1 running, 201 sleeping, 0 stopped, 0 zombie

Cpu(s): 42.7%us, 2.3%sy, 0.0%ni, 55.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0

Mem: 1025624k total, 791160k used, 234464k free, 23400k buffers

Swap: 103420k total, 13404k used, 90016k free, 373504k cached

PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND

28915 enterpri 20 0 195m 5900 4212 S 39.9 0.6 3:36.98 edb-postgres

1033 root 20 0 171m 77m 2960 S 1.0 7.8 3:43.96 Xorg

3040 user 20 0 278m 22m 14m S 1.0 2.2 3:41.72 knotify4

.

.

.

The psql session performing the factorial calculation is shown by the row where edb-postgres appears under the COMMAND column. The CPU usage of the session shown under the %CPU column shows 39.9, which is close to the 40% CPU limit set for resource group resgrp\_b.

By contrast, if the psql session is removed from the resource group and the factorial calculation is performed again, the CPU usage is much higher.

edb=# SET edb\_resource\_group TO DEFAULT;

SET

edb=# SHOW edb\_resource\_group;

edb\_resource\_group

--------------------

(1 row)

edb=# SELECT 20000!;

Under the %CPU column for edb-postgres, the CPU usage is now 93.6, which is significantly higher than the 39.9 when the process was part of the resource group.

$ top

top - 16:43:03 up 4:21, 7 users, load average: 0.66, 0.33, 0.37

Tasks: 202 total, 5 running, 197 sleeping, 0 stopped, 0 zombie

Cpu(s): 96.7%us, 3.3%sy, 0.0%ni, 0.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0

Mem: 1025624k total, 791228k used, 234396k free, 23560k buffers

Swap: 103420k total, 13404k used, 90016k free, 373508k cached

PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND

28915 enterpri 20 0 195m 5900 4212 R 93.6 0.6 5:01.56 edb-postgres

1033 root 20 0 171m 77m 2960 S 1.0 7.8 3:48.15 Xorg

2907 user 20 0 98.7m 11m 9100 S 0.3 1.2 0:46.51 vmware-user-lo

.

.

.

### Example – Multiple Processes in a Single Group

As stated previously, the CPU rate limit applies to the aggregate of all processes in the resource group. This concept is illustrated in the following example.

The factorial calculation is performed simultaneously in two separate psql sessions, each of which has been added to resource group resgrp\_b that has cpu\_rate\_limit set to .4 (CPU usage of 40%).

**Session 1:**

edb=# SET edb\_resource\_group TO resgrp\_b;

SET

edb=# SHOW edb\_resource\_group;

edb\_resource\_group

--------------------

resgrp\_b

(1 row)

edb=# SELECT 20000!;

**Session 2:**

edb=# SET edb\_resource\_group TO resgrp\_b;

SET

edb=# SHOW edb\_resource\_group;

edb\_resource\_group

--------------------

resgrp\_b

(1 row)

edb=# SELECT 20000!;

A third session monitors the CPU usage.

$ top

top - 16:53:03 up 4:31, 7 users, load average: 0.31, 0.19, 0.27

Tasks: 202 total, 1 running, 201 sleeping, 0 stopped, 0 zombie

Cpu(s): 41.2%us, 3.0%sy, 0.0%ni, 55.8%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0

Mem: 1025624k total, 792020k used, 233604k free, 23844k buffers

Swap: 103420k total, 13404k used, 90016k free, 373508k cached

PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND

29857 enterpri 20 0 195m 4708 3312 S 19.9 0.5 0:57.35 edb-postgres

28915 enterpri 20 0 195m 5900 4212 S 19.6 0.6 5:35.49 edb-postgres

3040 user 20 0 278m 22m 14m S 1.0 2.2 3:54.99 knotify4

1033 root 20 0 171m 78m 2960 S 0.3 7.8 3:55.71 Xorg

.

.

.

There are now two processes named edb-postgres with %CPU values of 19.9 and 19.6, whose sum is close to the 40% CPU usage set for resource group resgrp\_b.

The following command sequence displays the sum of all edb-postgres processes sampled over half second time intervals. This shows how the total CPU usage of the processes in the resource group changes over time as EDB Resource Manager throttles the processes to keep the total resource group CPU usage near 40%.

$ while [[ 1 -eq 1 ]]; do top -d0.5 -b -n2 | grep edb-postgres | awk '{ SUM += $9} END { print SUM / 2 }'; done

37.2

39.1

38.9

38.3

44.7

39.2

42.5

39.1

39.2

39.2

41

42.85

46.1

.

.

.

### Example – Multiple Processes in Multiple Groups

In this example, two additional psql sessions are used along with the previous two sessions. The third and fourth sessions perform the same factorial calculation within resource group resgrp\_c with a cpu\_rate\_limit of .3 (30% CPU usage).

**Session 3:**

edb=# SET edb\_resource\_group TO resgrp\_c;

SET

edb=# SHOW edb\_resource\_group;

edb\_resource\_group

--------------------

resgrp\_c

(1 row)

edb=# SELECT 20000!;

**Session 4:**

edb=# SET edb\_resource\_group TO resgrp\_c;

SET

edb=# SHOW edb\_resource\_group;

edb\_resource\_group

--------------------

resgrp\_c

(1 row)

edb=# SELECT 20000!;

The top command displays the following output.

$ top

top - 17:45:09 up 5:23, 8 users, load average: 0.47, 0.17, 0.26

Tasks: 203 total, 4 running, 199 sleeping, 0 stopped, 0 zombie

Cpu(s): 70.2%us, 0.0%sy, 0.0%ni, 29.8%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0

Mem: 1025624k total, 806140k used, 219484k free, 25296k buffers

Swap: 103420k total, 13404k used, 90016k free, 374092k cached

PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND

29857 enterpri 20 0 195m 4820 3324 S 19.9 0.5 4:25.02 edb-postgres

28915 enterpri 20 0 195m 5900 4212 R 19.6 0.6 9:07.50 edb-postgres

29023 enterpri 20 0 195m 4744 3248 R 16.3 0.5 4:01.73 edb-postgres

11019 enterpri 20 0 195m 4120 2764 R 15.3 0.4 0:04.92 edb-postgres

2907 user 20 0 98.7m 12m 9112 S 1.3 1.2 0:56.54 vmware-user-lo

3040 user 20 0 278m 22m 14m S 1.3 2.2 4:38.73 knotify4

The two resource groups in use have CPU usage limits of 40% and 30%. The sum of the %CPU column for the first two edb-postgres processes is 39.5 (approximately 40%, which is the limit for resgrp\_b) and the sum of the %CPU column for the third and fourth edb-postgres processes is 31.6 (approximately 30%, which is the limit for resgrp\_c).

The sum of the CPU usage limits of the two resource groups to which these processes belong is 70%. The following output shows that the sum of the four processes borders around 70%.

$ while [[ 1 -eq 1 ]]; do top -d0.5 -b -n2 | grep edb-postgres | awk '{ SUM += $9} END { print SUM / 2 }'; done

61.8

76.4

72.6

69.55

64.55

79.95

68.55

71.25

74.85

62

74.85

76.9

72.4

65.9

74.9

68.25

By contrast, if three sessions are processing where two sessions remain in resgrp\_b, but the third session does not belong to any resource group, the top command shows the following output.

$ top

top - 17:24:55 up 5:03, 7 users, load average: 1.00, 0.41, 0.38

Tasks: 199 total, 3 running, 196 sleeping, 0 stopped, 0 zombie

Cpu(s): 99.7%us, 0.3%sy, 0.0%ni, 0.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0

Mem: 1025624k total, 797692k used, 227932k free, 24724k buffers

Swap: 103420k total, 13404k used, 90016k free, 374068k cached

PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND

29023 enterpri 20 0 195m 4744 3248 R 58.6 0.5 2:53.75 edb-postgres

28915 enterpri 20 0 195m 5900 4212 S 18.9 0.6 7:58.45 edb-postgres

29857 enterpri 20 0 195m 4820 3324 S 18.9 0.5 3:14.85 edb-postgres

1033 root 20 0 174m 81m 2960 S 1.7 8.2 4:26.50 Xorg

3040 user 20 0 278m 22m 14m S 1.0 2.2 4:21.20 knotify4

The second and third edb-postgres processes belonging to the resource group where the CPU usage is limited to 40%, have a total CPU usage of 37.8. However, the first edb-postgres process has a 58.6% CPU usage as it is not within a resource group, and basically utilizes the remaining, available CPU resources on the system.

Likewise, the following output shows the sum of all three sessions is around 95% since one of the sessions has no set limit on its CPU usage.

$ while [[ 1 -eq 1 ]]; do top -d0.5 -b -n2 | grep edb-postgres | awk '{ SUM += $9} END { print SUM / 2 }'; done

96

90.35

92.55

96.4

94.1

90.7

95.7

95.45

93.65

87.95

96.75

94.25

95.45

97.35

92.9

96.05

96.25

94.95

.

.

.

## Dirty Buffer Throttling

Writing to shared buffers is controlled by setting the dirty\_rate\_limit resource type parameter.

Set the dirty\_rate\_limit parameter to the number of kilobytes per second for the combined rate at which all the processes in the group should write to or “dirty” the shared buffers. An example setting would be 3072 kilobytes per seconds.

The valid range of the dirty\_rate\_limit parameter is 0 to 1.67772e+07. A setting of 0 means no dirty rate limit has been set for the resource group.

EDB Resource Manager utilizes dirty buffer throttling to keep the aggregate, shared buffer writing rate of all processes in the group near the limit specified by the dirty\_rate\_limit parameter. A process in the group may be interrupted and put into sleep mode for a short interval of time to maintain the defined limit. When and how such interruptions occur is defined by a proprietary algorithm used by EDB Resource Manager.

### Setting the Dirty Rate Limit for a Resource Group

The ALTER RESOURCE GROUP command with the SET dirty\_rate\_limit clause is used to set the dirty rate limit for a resource group.

In the following example the dirty rate limit is set to 12288 kilobytes per second for resgrp\_a, 6144 kilobytes per second for resgrp\_b and 3072 kilobytes per second for resgrp\_c. This means that the combined writing rate to the shared buffer of all processes assigned to resgrp\_a is maintained at approximately 12288 kilobytes per second. Similarly, for all processes in resgrp\_b, the combined writing rate to the shared buffer is kept to approximately 6144 kilobytes per second, etc.

edb=# ALTER RESOURCE GROUP resgrp\_a SET dirty\_rate\_limit TO 12288;

ALTER RESOURCE GROUP

edb=# ALTER RESOURCE GROUP resgrp\_b SET dirty\_rate\_limit TO 6144;

ALTER RESOURCE GROUP

edb=# ALTER RESOURCE GROUP resgrp\_c SET dirty\_rate\_limit TO 3072;

ALTER RESOURCE GROUP

The following query shows the settings of dirty\_rate\_limit in the catalog.

edb=# SELECT rgrpname, rgrpdirtyratelimit FROM edb\_resource\_group;

rgrpname | rgrpdirtyratelimit

----------+--------------------

resgrp\_a | 12288

resgrp\_b | 6144

resgrp\_c | 3072

(3 rows)

Changing the dirty\_rate\_limit of a resource group not only affects new processes that are assigned to the group, but any currently running processes that are members of the group are immediately affected by the change. That is, if the dirty\_rate\_limit is changed from 12288 to 3072, currently running processes in the group would be throttled downward so that the aggregate group dirty rate would be near 3072 kilobytes per second instead of 12288 kilobytes per second.

To illustrate the effect of setting the dirty rate limit for resource groups, the following examples use the following table for intensive I/O operations.

CREATE TABLE t1 (c1 INTEGER, c2 CHARACTER(500)) WITH (FILLFACTOR = 10);

The FILLFACTOR = 10 clause results in INSERT commands packing rows up to only 10% per page. This results in a larger sampling of dirty shared blocks for the purpose of these examples.

The pg\_stat\_statements module is used to display the number of shared buffer blocks that are dirtied by a SQL command and the amount of time the command took to execute. This provides the information to calculate the actual kilobytes per second writing rate for the SQL command, and thus compare it to the dirty rate limit set for a resource group.

In order to use the pg\_stat\_statements module, perform the following steps.

Step 1: In the postgresql.conf file, add $libdir/pg\_stat\_statements to the shared\_preload\_libraries configuration parameter as shown by the following.

shared\_preload\_libraries = '$libdir/dbms\_pipe,$libdir/edb\_gen,$libdir/pg\_stat\_statements'

Step 2: Restart the database server.

Step 3: Use the CREATE EXTENSION command to complete the creation of the pg\_stat\_statements module.

edb=# CREATE EXTENSION pg\_stat\_statements SCHEMA public;

CREATE EXTENSION

The pg\_stat\_statements\_reset() function is used to clear out the pg\_stat\_statements view for clarity of each example.

The resource groups with the dirty rate limit settings shown in the previous query are used in these examples.

### Example – Single Process in a Single Group

The following sequence of commands shows the creation of table t1. The current process is set to use resource group resgrp\_b. The pg\_stat\_statements view is cleared out by running the pg\_stat\_statements\_reset() function.

Finally, the INSERT command generates a series of integers from 1 to 10,000 to populate the table, and dirty approximately 10,000 blocks.

edb=# CREATE TABLE t1 (c1 INTEGER, c2 CHARACTER(500)) WITH (FILLFACTOR = 10);

CREATE TABLE

edb=# SET edb\_resource\_group TO resgrp\_b;

SET

edb=# SHOW edb\_resource\_group;

edb\_resource\_group

--------------------

resgrp\_b

(1 row)

edb=# SELECT pg\_stat\_statements\_reset();

pg\_stat\_statements\_reset

--------------------------

(1 row)

edb=# INSERT INTO t1 VALUES (generate\_series (1,10000), 'aaa');

INSERT 0 10000

The following shows the results from the INSERT command.

edb=# SELECT query, rows, total\_time, shared\_blks\_dirtied FROM pg\_stat\_statements;

-[ RECORD 1 ]-------+--------------------------------------------------

query | INSERT INTO t1 VALUES (generate\_series (?,?), ?);

rows | 10000

total\_time | 13496.184

shared\_blks\_dirtied | 10003

The actual dirty rate is calculated as follows.

* The number of blocks dirtied per millisecond (ms) is 10003 blocks / 13496.184 ms, which yields 0.74117247 blocks per millisecond.
* Multiply the result by 1000 to give the number of shared blocks dirtied per second (1 second = 1000 ms), which yields 741.17247 blocks per second.
* Multiply the result by 8.192 to give the number of kilobytes dirtied per second (1 block = 8.192 kilobytes), which yields approximately 6072 kilobytes per second.

Note that the actual dirty rate of 6072 kilobytes per second is close to the dirty rate limit for the resource group, which is 6144 kilobytes per second.

By contrast, if the steps are repeated again without the process belonging to any resource group, the dirty buffer rate is much higher.

edb=# CREATE TABLE t1 (c1 INTEGER, c2 CHARACTER(500)) WITH (FILLFACTOR = 10);

CREATE TABLE

edb=# SHOW edb\_resource\_group;

edb\_resource\_group

--------------------

(1 row)

edb=# SELECT pg\_stat\_statements\_reset();

pg\_stat\_statements\_reset

--------------------------

(1 row)

edb=# INSERT INTO t1 VALUES (generate\_series (1,10000), 'aaa');

INSERT 0 10000

The following shows the results from the INSERT command without the usage of a resource group.

edb=# SELECT query, rows, total\_time, shared\_blks\_dirtied FROM pg\_stat\_statements;

-[ RECORD 1 ]-------+--------------------------------------------------

query | INSERT INTO t1 VALUES (generate\_series (?,?), ?);

rows | 10000

total\_time | 2432.165

shared\_blks\_dirtied | 10003

First, note the total time was only 2432.165 milliseconds as compared to 13496.184 milliseconds when a resource group with a dirty rate limit set to 6144 kilobytes per second was used.

The actual dirty rate without the use of a resource group is calculated as follows.

* The number of blocks dirtied per millisecond (ms) is 10003 blocks / 2432.165 ms, which yields 4.112797 blocks per millisecond.
* Multiply the result by 1000 to give the number of shared blocks dirtied per second (1 second = 1000 ms), which yields 4112.797 blocks per second.
* Multiply the result by 8.192 to give the number of kilobytes dirtied per second (1 block = 8.192 kilobytes), which yields approximately 33692 kilobytes per second.

Note that the actual dirty rate of 33692 kilobytes per second is significantly higher than when the resource group with a dirty rate limit of 6144 kilobytes per second was used.

### Example – Multiple Processes in a Single Group

As stated previously, the dirty rate limit applies to the aggregate of all processes in the resource group. This concept is illustrated in the following example.

For this example the inserts are performed simultaneously on two different tables in two separate psql sessions, each of which has been added to resource group resgrp\_b that has a dirty\_rate\_limit set to 6144 kilobytes per second.

**Session 1:**

edb=# CREATE TABLE t1 (c1 INTEGER, c2 CHARACTER(500)) WITH (FILLFACTOR = 10);

CREATE TABLE

edb=# SET edb\_resource\_group TO resgrp\_b;

SET

edb=# SHOW edb\_resource\_group;

edb\_resource\_group

--------------------

resgrp\_b

(1 row)

edb=# INSERT INTO t1 VALUES (generate\_series (1,10000), 'aaa');

INSERT 0 10000

**Session 2:**

edb=# CREATE TABLE t2 (c1 INTEGER, c2 CHARACTER(500)) WITH (FILLFACTOR = 10);

CREATE TABLE

edb=# SET edb\_resource\_group TO resgrp\_b;

SET

edb=# SHOW edb\_resource\_group;

edb\_resource\_group

--------------------

resgrp\_b

(1 row)

edb=# SELECT pg\_stat\_statements\_reset();

pg\_stat\_statements\_reset

--------------------------

(1 row)

edb=# INSERT INTO t2 VALUES (generate\_series (1,10000), 'aaa');

INSERT 0 10000

**Note:** The INSERT commands in session 1 and session 2 were started after the SELECT pg\_stat\_statements\_reset() command in session 2 was run.

The following shows the results from the INSERT commands in the two sessions. RECORD 3 shows the results from session 1. RECORD 2 shows the results from session 2.

edb=# SELECT query, rows, total\_time, shared\_blks\_dirtied FROM pg\_stat\_statements;

-[ RECORD 1 ]-------+--------------------------------------------------

query | SELECT pg\_stat\_statements\_reset();

rows | 1

total\_time | 0.43

shared\_blks\_dirtied | 0

-[ RECORD 2 ]-------+--------------------------------------------------

query | INSERT INTO t2 VALUES (generate\_series (?,?), ?);

rows | 10000

total\_time | 30591.551

shared\_blks\_dirtied | 10003

-[ RECORD 3 ]-------+--------------------------------------------------

query | INSERT INTO t1 VALUES (generate\_series (?,?), ?);

rows | 10000

total\_time | 33215.334

shared\_blks\_dirtied | 10003

First, note the total time was 33215.334 milliseconds for session 1 and 30591.551 milliseconds for session 2. When only one session was active in the same resource group as shown in the first example, the time was 13496.184 milliseconds. Thus more active processes in the resource group result in a slower dirty rate for each active process in the group. This is shown in the following calculations.

The actual dirty rate for session 1 is calculated as follows.

* The number of blocks dirtied per millisecond (ms) is 10003 blocks / 33215.334 ms, which yields 0.30115609 blocks per millisecond.
* Multiply the result by 1000 to give the number of shared blocks dirtied per second (1 second = 1000 ms), which yields 301.15609 blocks per second.
* Multiply the result by 8.192 to give the number of kilobytes dirtied per second (1 block = 8.192 kilobytes), which yields approximately 2467 kilobytes per second.

The actual dirty rate for session 2 is calculated as follows.

* The number of blocks dirtied per millisecond (ms) is 10003 blocks / 30591.551 ms, which yields 0.32698571 blocks per millisecond.
* Multiply the result by 1000 to give the number of shared blocks dirtied per second (1 second = 1000 ms), which yields 326.98571 blocks per second.
* Multiply the result by 8.192 to give the number of kilobytes dirtied per second (1 block = 8.192 kilobytes), which yields approximately 2679 kilobytes per second.

The combined dirty rate from session 1 (2467 kilobytes per second) and from session 2 (2679 kilobytes per second) yields 5146 kilobytes per second, which is below the set dirty rate limit of the resource group (6144 kilobytes per seconds).

### Example – Multiple Processes in Multiple Groups

In this example, two additional psql sessions are used along with the previous two sessions. The third and fourth sessions perform the same INSERT command in resource group resgrp\_c with a dirty\_rate\_limit of 3072 kilobytes per second.

Sessions 1 and 2 are repeated as illustrated in the prior example using resource group resgrp\_b. with a dirty\_rate\_limit of 6144 kilobytes per second.

**Session 3:**

edb=# CREATE TABLE t3 (c1 INTEGER, c2 CHARACTER(500)) WITH (FILLFACTOR = 10);

CREATE TABLE

edb=# SET edb\_resource\_group TO resgrp\_c;

SET

edb=# SHOW edb\_resource\_group;

edb\_resource\_group

--------------------

resgrp\_c

(1 row)

edb=# INSERT INTO t3 VALUES (generate\_series (1,10000), 'aaa');

INSERT 0 10000

**Session 4:**

edb=# CREATE TABLE t4 (c1 INTEGER, c2 CHARACTER(500)) WITH (FILLFACTOR = 10);

CREATE TABLE

edb=# SET edb\_resource\_group TO resgrp\_c;

SET

edb=# SHOW edb\_resource\_group;

edb\_resource\_group

--------------------

resgrp\_c

(1 row)

edb=# SELECT pg\_stat\_statements\_reset();

pg\_stat\_statements\_reset

--------------------------

(1 row)

edb=# INSERT INTO t4 VALUES (generate\_series (1,10000), 'aaa');

INSERT 0 10000

**Note:** The INSERT commands in all four sessions were started after the SELECT pg\_stat\_statements\_reset() command in session 4 was run.

The following shows the results from the INSERT commands in the four sessions. RECORD 3 shows the results from session 1. RECORD 2 shows the results from session 2. RECORD 4 shows the results from session 3. RECORD 5 shows the results from session 4.

edb=# SELECT query, rows, total\_time, shared\_blks\_dirtied FROM pg\_stat\_statements;

-[ RECORD 1 ]-------+--------------------------------------------------

query | SELECT pg\_stat\_statements\_reset();

rows | 1

total\_time | 0.467

shared\_blks\_dirtied | 0

-[ RECORD 2 ]-------+--------------------------------------------------

query | INSERT INTO t2 VALUES (generate\_series (?,?), ?);

rows | 10000

total\_time | 31343.458

shared\_blks\_dirtied | 10003

-[ RECORD 3 ]-------+--------------------------------------------------

query | INSERT INTO t1 VALUES (generate\_series (?,?), ?);

rows | 10000

total\_time | 28407.435

shared\_blks\_dirtied | 10003

-[ RECORD 4 ]-------+--------------------------------------------------

query | INSERT INTO t3 VALUES (generate\_series (?,?), ?);

rows | 10000

total\_time | 52727.846

shared\_blks\_dirtied | 10003

-[ RECORD 5 ]-------+--------------------------------------------------

query | INSERT INTO t4 VALUES (generate\_series (?,?), ?);

rows | 10000

total\_time | 56063.697

shared\_blks\_dirtied | 10003

First note that the times of session 1 (28407.435) and session 2 (31343.458) are close to each other as they are both in the same resource group with dirty\_rate\_limit set to 6144, as compared to the times of session 3 (52727.846) and session 4 (56063.697), which are in the resource group with dirty\_rate\_limit set to 3072. The latter group has a slower dirty rate limit so the expected processing time is longer as is the case for sessions 3 and 4.

The actual dirty rate for session 1 is calculated as follows.

* The number of blocks dirtied per millisecond (ms) is 10003 blocks / 28407.435 ms, which yields 0.35212612 blocks per millisecond.
* Multiply the result by 1000 to give the number of shared blocks dirtied per second (1 second = 1000 ms), which yields 352.12612 blocks per second.
* Multiply the result by 8.192 to give the number of kilobytes dirtied per second (1 block = 8.192 kilobytes), which yields approximately 2885 kilobytes per second.

The actual dirty rate for session 2 is calculated as follows.

* The number of blocks dirtied per millisecond (ms) is 10003 blocks / 31343.458 ms, which yields 0.31914156 blocks per millisecond.
* Multiply the result by 1000 to give the number of shared blocks dirtied per second (1 second = 1000 ms), which yields 319.14156 blocks per second.
* Multiply the result by 8.192 to give the number of kilobytes dirtied per second (1 block = 8.192 kilobytes), which yields approximately 2614 kilobytes per second.

The combined dirty rate from session 1 (2885 kilobytes per second) and from session 2 (2614 kilobytes per second) yields 5499 kilobytes per second, which is near the set dirty rate limit of the resource group (6144 kilobytes per seconds).

The actual dirty rate for session 3 is calculated as follows.

* The number of blocks dirtied per millisecond (ms) is 10003 blocks / 52727.846 ms, which yields 0.18971001 blocks per millisecond.
* Multiply the result by 1000 to give the number of shared blocks dirtied per second (1 second = 1000 ms), which yields 189.71001 blocks per second.
* Multiply the result by 8.192 to give the number of kilobytes dirtied per second (1 block = 8.192 kilobytes), which yields approximately 1554 kilobytes per second.

The actual dirty rate for session 4 is calculated as follows.

* The number of blocks dirtied per millisecond (ms) is 10003 blocks / 56063.697 ms, which yields 0.17842205 blocks per millisecond.
* Multiply the result by 1000 to give the number of shared blocks dirtied per second (1 second = 1000 ms), which yields 178.42205 blocks per second.
* Multiply the result by 8.192 to give the number of kilobytes dirtied per second (1 block = 8.192 kilobytes), which yields approximately 1462 kilobytes per second.

The combined dirty rate from session 3 (1554 kilobytes per second) and from session 4 (1462 kilobytes per second) yields 3016 kilobytes per second, which is near the set dirty rate limit of the resource group (3072 kilobytes per seconds).

Thus, this demonstrates how EDB Resource Manager keeps the aggregate dirty rate of the active processes in its groups close to the dirty rate limit set for each group.

## System Catalogs

This section describes the system catalogs that store the resource group information used by EDB Resource Manager.

### edb\_all\_resource\_groups

The following table lists the information available in the edb\_all\_resource\_groups catalog:

| Column | Type | Description |
| --- | --- | --- |
| group\_name | name | The name of the resource group. |
| active\_processes | integer | Number of currently active processes in the resource group. |
| cpu\_rate\_limit | float8 | Maximum CPU rate limit for the resource group. 0 means no limit. |
| per\_process\_cpu\_rate\_limit | float8 | Maximum CPU rate limit per currently active process in the resource group. |
| dirty\_rate\_limit | float8 | Maximum dirty rate limit for a resource group. 0 means no limit. |
| per\_process\_dirty\_rate\_limit | float8 | Maximum dirty rate limit per currently active process in the resource group. |

### edb\_resource\_group

The following table lists the information available in the edb\_resource\_group catalog:

| Column | Type | Description |
| --- | --- | --- |
| rgrpname | name | The name of the resource group. |
| rgrpcpuratelimit | float8 | Maximum CPU rate limit for a resource group. 0 means no limit. |
| rgrpdirtyratelimit | float8 | Maximum dirty rate limit for a resource group. 0 means no limit. |

# Database Utilities

This chapter describes various database utilities that provide many usage benefits with Advanced Server.

## EDB\*Loader

EDB\*Loader is a high-performance bulk data loader that provides an interface compatible with Oracle databases for Advanced Server. The EDB\*Loader command line utility loads data from an input source, typically a file, into one or more tables using a subset of the parameters offered by Oracle SQL\*Loader.

EDB\*Loader features include:

* Support for the Oracle SQL\*Loader data loading methods - conventional path load, direct path load, and parallel direct path load
* Oracle SQL\*Loader compatible syntax for control file directives
* Input data with delimiter-separated or fixed-width fields
* Bad file for collecting rejected records
* Loading of multiple target tables
* Discard file for collecting records that do not meet the selection criteria of any target table
* Log file for recording the EDB\*Loader session and any error messages
* Data loading from standard input and remote loading, particularly useful for large data sources on remote hosts

These features are explained in detail in the following sections.

Note: The following are important version compatibility restrictions between the EDB\*Loader client and the database server.

* Invoking EDB\*Loader is done using a client program called edbldr, which is used to pass parameters and directive information to the database server. It is strongly recommended that the 9.5 EDB\*Loader client (that is, the edbldr program supplied with EDB Postgres Advanced Server 9.5) be used to load data only into version 9.5 of the database server. In general, the EDB\*Loader client and database server should be the same version.
* It is possible to use a 9.5 EDB\*Loader client to load data into a 9.5 database server, but the new 9.5 EDB\*Loader features may not be available under those circumstances.
* Use of a 9.5, 9.4 or 9.3 EDB\*Loader client is not supported for database servers version 9.2 or earlier.

### Data Loading Methods

As with Oracle SQL\*Loader, EDB\*Loader supports three data loading methods:

* Conventional path load
* Direct path load
* Parallel direct path load

Conventional path load is the default method used by EDB\*Loader. Basic insert processing is used to add rows to the table.

The advantage of a conventional path load over the other methods is that table constraints and database objects defined on the table such as primary keys, not null constraints, check constraints, unique indexes, foreign key constraints, and triggers are enforced during a conventional path load.

One exception is that Advanced Server rules defined on the table are not enforced. EDB\*Loader can load tables on which rules are defined, but the rules are not executed. As a consequence, partitioned tables implemented using rules cannot be loaded using EDB\*Loader.

Note: Advanced Server rules are created by the CREATE RULE command. Advanced Server rules are not the same database objects as rules and rule sets used in Oracle.

EDB\*Loader also supports direct path loads. A direct path load is faster than a conventional path load, but requires the removal of most types of constraints and triggers from the table. See Section 6.1.5 for information on direct path loads.

Finally, EDB\*Loader supports parallel direct path loads. A parallel direct path load provides even greater performance improvement by permitting multiple EDB\*Loader sessions to run simultaneously to load a single table. See Section 6.1.6 for information on parallel direct path loads.

### General Usage

EDB\*Loader can load data files with either delimiter-separated or fixed-width fields, in single-byte or multi-byte character sets. The delimiter can be a string consisting of one or more single-byte or multi-byte characters. Data file encoding and the database encoding may be different. Character set conversion of the data file to the database encoding is supported.

Each EDB\*Loader session runs as a single, independent transaction. If an error should occur during the EDB\*Loader session that aborts the transaction, all changes made during the session are rolled back.

Generally, formatting errors in the data file do not result in an aborted transaction. Instead, the badly formatted records are written to a text file called the bad file. The reason for the error is recorded in the log file.

Records causing database integrity errors do result in an aborted transaction and rollback. As with formatting errors, the record causing the error is written to the bad file and the reason is recorded in the log file.

Note: EDB\*Loader differs from Oracle SQL\*Loader in that a database integrity error results in a rollback in EDB\*Loader. In Oracle SQL\*Loader, only the record causing the error is rejected. Records that were previously inserted into the table are retained and loading continues after the rejected record.

The following are examples of types of formatting errors that do not abort the transaction:

* Attempt to load non-numeric value into a numeric column
* Numeric value is too large for a numeric column
* Character value is too long for the maximum length of a character column
* Attempt to load improperly formatted date value into a date column

The following are examples of types of database errors that abort the transaction and result in the rollback of all changes made in the EDB\*Loader session:

* Violation of a unique constraint such as a primary key or unique index
* Violation of a referential integrity constraint
* Violation of a check constraint
* Error thrown by a trigger fired as a result of inserting rows

### Building the EDB\*Loader Control File

When you invoke EDB\*Loader, the list of arguments provided must include the name of a control file. The control file includes the instructions that EDB\*Loader uses to load the table (or tables) from the input data file. The control file includes information such as:

* The name of the input data file containing the data to be loaded.
* The name of the table or tables to be loaded from the data file.
* Names of the columns within the table or tables and their corresponding field placement in the data file.
* Specification of whether the data file uses a delimiter string to separate the fields, or if the fields occupy fixed column positions.
* Optional selection criteria to choose which records from the data file to load into a given table.
* The name of the file that will collect illegally formatted records.
* The name of the discard file that will collect records that do not meet the selection criteria of any table.

The syntax for the EDB\*Loader control file is as follows:

[ OPTIONS (param=value [, param=value ] ...) ]

LOAD DATA

[ CHARACTERSET charset ]

[ INFILE '{ data\_file | stdin }' ]

[ BADFILE 'bad\_file' ]

[ DISCARDFILE 'discard\_file' ]

[ { DISCARDMAX | DISCARDS } max\_discard\_recs ]

[ INSERT | APPEND | REPLACE | TRUNCATE ]

[ PRESERVE BLANKS ]

{ INTO TABLE target\_table

[ WHEN field\_condition [ AND field\_condition ] ...]

[ FIELDS TERMINATED BY 'termstring'

[ OPTIONALLY ENCLOSED BY 'enclstring' ] ]

[ TRAILING NULLCOLS ]

(field\_def [, field\_def ] ...)

} ...

where field\_def defines afield in the specified data\_file that describes the location, data format, or value of the data to be inserted into column\_name of the target\_table. The syntax of field\_def is the following:

column\_name {

CONSTANT val |

FILLER [ POSITION (start:end) ] [ fieldtype ] |

[ POSITION (start:end) ] [ fieldtype ]

[ PRESERVE BLANKS ] [ "expr" ]

}

where fieldtype is one of:

CHAR | DATE [ "datemask" ] | INTEGER EXTERNAL |   
FLOAT EXTERNAL | DECIMAL EXTERNAL | ZONED EXTERNAL |   
ZONED [(precision[,scale])]

Description

The specification of data\_file, bad\_file, and discard\_file may include the full directory path or a relative directory path to the file name. If the file name is specified alone or with a relative directory path, the file is then assumed to exist (in the case of data\_file), or is created (in the case of bad\_file or discard\_file), relative to the current working directory from which edbldr is invoked.

You can include references to environment variables within the EDB\*Loader control file when referring to a directory path and/or file name. Environment variable references are formatted differently on Windows systems than on Linux systems:

* On Linux, the format is $ENV\_VARIABLE or ${ENV\_VARIABLE}
* On Windows, the format is %ENV\_VARIABLE%

Where ENV\_VARIABLE is the environment variable that is set to the directory path and/or file name.

The EDBLDR\_ENV\_STYLE environment variable instructs Advanced Server to interpret environment variable references as Windows-styled references or Linux-styled references irregardless of the operating system on which EDB\*Loader resides. You can use this environment variable to create portable control files for EDB\*Loader.

* On a Windows system, set EDBLDR\_ENV\_STYLE to linux or unix to instruct Advanced Server to recognize Linux-style references within the control file.
* On a Linux system, set EDBLDR\_ENV\_STYLE to windows to instruct Advanced Server to recognize Windows-style references within the control file.

The operating system account enterprisedb must have read permission on the directory and file specified by data\_file.

The operating system account enterprisedb must have write permission on the directories where bad\_file and discard\_file are to be written.

Note: It is suggested that the file names for data\_file, bad\_file, and discard\_file include extensions of .dat, .bad, and .dsc, respectively. If the provided file name does not contain an extension, EDB\*Loader assumes the actual file name includes the appropriate aforementioned extension.

If an EDB\*Loader session results in data format errors and the BADFILE clause is not specified, nor is the BAD parameter given on the command line when edbldr is invoked, a bad file is created with the name control\_file\_base.bad in the current working directory from which edbldr is invoked. control\_file\_base is the base name of the control file (that is, the file name without any extension) used in the edbldr session.

If all of the following conditions are true, the discard file is not created even if the EDB\*Loader session results in discarded records:

* The DISCARDFILE clause for specifying the discard file is not included in the control file.
* The DISCARD parameter for specifying the discard file is not included on the command line.
* The DISCARDMAX clause for specifying the maximum number of discarded records is not included in the control file.
* The DISCARDS clause for specifying the maximum number of discarded records is not included in the control file.
* The DISCARDMAX parameter for specifying the maximum number of discarded records is not included on the command line.

If neither the DISCARDFILE clause nor the DISCARD parameter for explicitly specifying the discard file name are specified, but DISCARDMAX or DISCARDS is specified, then the EDB\*Loader session creates a discard file using the data file name with an extension of .dsc.

Note: There is a distinction between keywords DISCARD and DISCARDS. DISCARD is an EDB\*Loader command line parameter used to specify the discard file name (see Section 6.1.4). DISCARDS is a clause of the LOAD DATA directive that may only appear in the control file. Keywords DISCARDS and DISCARDMAX provide the same functionality of specifying the maximum number of discarded records allowed before terminating the EDB\*Loader session. Records loaded into the database before termination of the EDB\*Loader session due to exceeding the DISCARDS or DISCARDMAX settings are kept in the database and are not rolled back.

If one of INSERT, APPEND, REPLACE, or TRUNCATE is specified, it establishes the default action of how rows are to be added to target tables. If omitted, the default action is as if INSERT had been specified.

If the FIELDS TERMINATED BY clause is specified, then the POSITION (start:end) clause may not be specified for any field\_def. Alternatively if the FIELDS TERMINATED BY clause is not specified, then every field\_def must contain the POSITION (start:end) clause, excluding those with the CONSTANT clause.

Parameters

OPTIONS param=value

Use the OPTIONS clause to specify param=value pairs that represent an EDB\*Loader directive. If a parameter is specified in both the OPTIONS clause and on the command line when edbldr is invoked, the command line setting is used.

Specify one or more of the following parameter/value pairs:

DIRECT= { FALSE | TRUE }

If DIRECT is set to TRUE EDB\*Loader performs a direct path load instead of a conventional path load. The default value of DIRECT is FALSE.

See Section 6.1.5 for information on direct path loads.

ERRORS=error\_count

*error\_count* specifies the number of errors permitted before aborting the EDB\*Loader session. The default is 50.

FREEZE= { FALSE | TRUE }

Set FREEZE to TRUE to indicate that the data should be copied with the rows *frozen*. A tuple guaranteed to be visible to all current and future transactions is marked as frozen to prevent transaction ID wrap-around. For more information about frozen tuples, please refer to the PostgreSQL core documentation at:

<http://www.postgresql.org/docs/9.5/static/routine-vacuuming.html>

You must specify a data-loading type of TRUNCATE in the control file when using the FREEZE option. FREEZE is not supported for direct loading.

By default, FREEZE is FALSE.

PARALLEL= { FALSE | TRUE }

Set PARALLEL to TRUE to indicate that this EDB\*Loader session is one of a number of concurrent EDB\*Loader sessions participating in a parallel direct path load. The default value of PARALLEL is FALSE.

When PARALLEL is TRUE, the DIRECT parameter must also be set to TRUE . See Section 6.1.6 for more information about parallel direct path loads.

ROWS=*n*

*n* specifies the number of rows that EDB\*Loader will commit before loading the next set of *n* rows.

If EDB\*Loader encounters an invalid row during a load (in which the ROWS parameter is specified), those rows committed prior to encountering the error will remain in the destination table.

SKIP=*skip*\_*count*

skip\_count specifies the number of records at the beginning of the input data file that should be skipped before loading begins. The default is 0.

SKIP\_INDEX\_MAINTENANCE={ FALSE | TRUE }

If SKIP\_INDEX\_MAINTENANCE is TRUE, index maintenance is not performed as part of a direct path load, and indexes on the loaded table are marked as invalid. The default value of SKIP\_INDEX\_MAINTENANCE is FALSE.

Please note: During a parallel direct path load, target table indexes are not updated, and are marked as invalid after the load is complete.

You can use the REINDEX command to rebuild an index. For more information about the REINDEX command, please refer to the PostgreSQL core documentation available at:

<http://www.postgresql.org/docs/9.5/static/sql-reindex.html>

charset

Use the CHARACTERSET clause to identify the character set encoding of data\_file where charset is the character set name. This clause is required if the data file encoding differs from the control file encoding. (The control file encoding must always be in the encoding of the client where edbldr is invoked.)

Examples of charset settings are UTF8, SQL\_ASCII, and SJIS.

For more information about client to database character set conversion, please refer to the PostgreSQL core documentation available at:

<http://www.postgresql.org/docs/9.5/static/multibyte.html>

data\_file

File containing the data to be loaded into target\_table. Each record in the data file corresponds to a row to be inserted into target\_table.

If an extension is not provided in the file name, EDB\*Loader assumes the file has an extension of .dat, for example, mydatafile.dat.

Note: If the DATA parameter is specified on the command line when edbldr is invoked, the file given by the command line DATA parameter is used instead.

If the INFILE clause is omitted as well as the command line DATA parameter, then the data file name is assumed to be identical to the control file name, but with an extension of .dat.

stdin

Specify stdin (all lowercase letters) if you want to use standard input to pipe the data to be loaded directly to EDB\*Loader. This is useful for data sources generating a large number of records to be loaded.

bad\_file

File that receives data\_file records that cannot be loaded due to errors.

If an extension is not provided in the file name, EDB\*Loader assumes the file has an extension of .bad, for example, mybadfile.bad.

Note: If the BAD parameter is specified on the command line when edbldr is invoked, the file given by the command line BAD parameter is used instead.

discard\_file

File that receives input data records that are not loaded into any table because none of the selection criteria are met for tables with the WHEN clause, and there are no tables without a WHEN clause. (All records meet the selection criteria of a table without a WHEN clause.)

If an extension is not provided in the file name, EDB\*Loader assumes the file has an extension of .dsc, for example, mydiscardfile.dsc.

Note: If the DISCARD parameter is specified on the command line when edbldr is invoked, the file given by the command line DISCARD parameter is used instead.

{ DISCARDMAX | DISCARDS } max\_discard\_recs

Maximum number of discarded records that may be encountered from the input data records before terminating the EDB\*Loader session. (A discarded record is described in the preceding description of the discard\_file parameter.) Either keyword DISCARDMAX or DISCARDS may be used preceding the integer value specified by max\_discard\_recs.

For example, if max\_discard\_recs is 0, then the EDB\*Loader session is terminated if and when a first discarded record is encountered. If max\_discard\_recs is 1, then the EDB\*Loader session is terminated if and when a second discarded record is encountered.

When the EDB\*Loader session is terminated due to exceeding max\_discard\_recs, prior input data records that have been loaded into the database are retained. They are not rolled back.

INSERT | APPEND | REPLACE | TRUNCATE

Specifies how data is to be loaded into the target tables. If one of INSERT, APPEND, REPLACE, or TRUNCATE is specified, it establishes the default action for all tables, overriding the default of INSERT.

INSERT

Data is to be loaded into an empty table. EDB\*Loader throws an exception and does not load any data if the table is not initially empty.

Note: If the table contains rows, the TRUNCATE command must be used to empty the table prior to invoking EDB\*Loader. EDB\*Loader throws an exception if the DELETE command is used to empty the table instead of the TRUNCATE command. Oracle SQL\*Loader allows the table to be emptied by using either the DELETE or TRUNCATE command.

APPEND

Data is to be added to any existing rows in the table. The table may be initially empty as well.

REPLACE

The REPLACE keyword and TRUNCATE keywords are functionally identical. The table is truncated by EDB\*Loader prior to loading the new data.

Note: Delete triggers on the table are not fired as a result of the REPLACE operation.

TRUNCATE

The table is truncated by EDB\*Loader prior to loading the new data. Delete triggers on the table are not fired as a result of the truncate operation.

PRESERVE BLANKS

For all target tables, retains leading white space when the optional enclosure delimiters are not present and leaves trailing white space intact when fields are specified with a predetermined size. When omitted, the default behavior is to trim leading and trailing white space.

target\_table

Name of the table into which data is to be loaded. The table name may be schema-qualified (for example, enterprisedb.emp). The specified target must not be a view.

field\_condition

Conditional clause taking the following form:

[ ( ] (start:end) { = | != | <> } 'val' [ ) ]

start and end are positive integers specifying the column positions in data\_file that mark the beginning and end of a field that is to be compared with the constant val. The first character in each record begins with a start value of 1.

In the WHEN field\_condition [ AND field\_condition ] clause, if all such conditions evaluate to true for a given record, then EDB\*Loader attempts to insert that record into target\_table. If the insert operation fails, the record is written to bad\_file.

All characters used in the field\_condition text (particularly in the val string) must be valid in the database encoding. (For performing data conversion, EDB\*Loader first converts the characters in val string to the database encoding and then to the data file encoding.)

If for a given record, none of the WHEN clauses evaluate to true for all INTO TABLE clauses, the record is written to discard\_file, if a discard file was specified for the EDB\*Loader session.

termstring

String of one or more characters that separates each field in data\_file. The characters may be single-byte or multi-byte as long as they are valid in the database encoding. Two consecutive appearances of termstring with no intervening character results in the corresponding column set to null.

enclstring

String of one or more characters used to enclose a field value in data\_file. The characters may be single-byte or multi-byte as long as they are valid in the database encoding. Use enclstring on fields where termstring appears as part of the data.

TRAILING NULLCOLS

If TRAILING NULLCOLS is specified, then the columns in the column list for which there is no data in data\_file for a given record, are set to null when the row is inserted. This applies only to one or more consecutive columns at the end of the column list.

If fields are omitted at the end of a record and TRAILING NULLCOLS is not specified, EDB\*Loader assumes the record contains formatting errors and writes it to the bad file.

column\_name

Name of a column in target\_table into which a field value defined by field\_def is to be inserted.

CONSTANT val

Specifies a constant that is type-compatible with the column data type to which it is assigned in a field definition. Single or double quotes may enclose val. If val contains white space, then enclosing quotation marks must be used.

The use of the CONSTANT clause completely determines the value to be assigned to a column in each inserted row. No other clause may appear in the same field definition.

If the TERMINATED BY clause is used to delimit the fields in data\_file, there must be no delimited field in data\_file corresponding to any field definition with a CONSTANT clause. In other words, EDB\*Loader assumes there is no field in data\_file for any field definition with a CONSTANT clause.

FILLER

Specifies that the data in the field defined by the field definition is not to be loaded into the associated column. The column is set to null.

A column name defined with the FILLER clause must not be referenced in a SQL expression. See the discussion of the expr parameter.

POSITION (start:end)

Defines the location of the field in a record in a fixed-width field data file. start and end are positive integers. The first character in the record has a start value of 1.

CHAR | DATE [ "datemask" ] | INTEGER EXTERNAL |   
FLOAT EXTERNAL | DECIMAL EXTERNAL | ZONED EXTERNAL |   
ZONED [(precision[,scale])]

Field type that describes the format of the data field in data\_file.

Note: Specification of a field type is optional (for descriptive purposes only) and has no effect on whether or not EDB\*Loader successfully inserts the data in the field into the table column. Successful loading depends upon the compatibility of the column data type and the field value. For example, a column with data type NUMBER(7,2) successfully accepts a field containing 2600, but if the field contains a value such as 26XX, the insertion fails and the record is written to bad\_file.

Please note that ZONED data is not human-readable; ZONED data is stored in an internal format where each digit is encoded in a separate nibble/nybble/4-bit field. In each ZONED value, the last byte contains a single digit (in the high-order 4 bits) and the sign (in the low-order 4 bits).

precision

Use precision to specify the length of the ZONED value.

If the precision value specified for ZONED conflicts with the length calculated by the server based on information provided with the POSITION clause, EDB\*Loader will use the value specified for precision.

scale

*scale* specifies the number of digits to the right of the decimal point in a ZONED value.

datemask

Specifies the ordering and abbreviation of the day, month, and year components of a date field.

Note: If the DATE field type is specified along with a SQL expression for the column, then datemask must be specified after DATE and before the SQL expression. See the following discussion of the expr parameter.

PRESERVE BLANKS

For the column on which this option appears, retains leading white space when the optional enclosure delimiters are not present and leaves trailing white space intact when fields are specified with a predetermined size. When omitted, the default behavior is to trim leading and trailing white space.

expr

A SQL expression returning a scalar value that is type-compatible with the column data type to which it is assigned in a field definition. Double quotes must enclose expr. expr may contain a reference to any column in the field list (except for fields with the FILLER clause) by prefixing the column name by a colon character (:).

**Examples**

The following are some examples of control files and their corresponding data files.

The following control file uses a delimiter-separated data file that appends rows to the emp table:

LOAD DATA

INFILE 'emp.dat'

BADFILE 'emp.bad'

APPEND

INTO TABLE emp

FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"'

TRAILING NULLCOLS

(

empno,

ename,

job,

mgr,

hiredate,

sal,

deptno,

comm

)

In the preceding control file, the APPEND clause is used to allow the insertion of additional rows into the emp table.

The following is the corresponding delimiter-separated data file:

9101,ROGERS,CLERK,7902,17-DEC-10,1980.00,20

9102,PETERSON,SALESMAN,7698,20-DEC-10,2600.00,30,2300.00

9103,WARREN,SALESMAN,7698,22-DEC-10,5250.00,30,2500.00

9104,"JONES, JR.",MANAGER,7839,02-APR-09,7975.00,20

The use of the TRAILING NULLCOLS clause allows the last field supplying the comm column to be omitted from the first and last records. The comm column is set to null for the rows inserted from these records.

The double quotation mark enclosure character surrounds the value JONES, JR. in the last record since the comma delimiter character is part of the field value.

The following query displays the rows added to the table after the EDB\*Loader session:

SELECT \* FROM emp WHERE empno > 9100;

empno | ename | job | mgr | hiredate | sal | comm | deptno

-------+------------+----------+------+--------------------+---------+---------+--------

9101 | ROGERS | CLERK | 7902 | 17-DEC-10 00:00:00 | 1980.00 | | 20

9102 | PETERSON | SALESMAN | 7698 | 20-DEC-10 00:00:00 | 2600.00 | 2300.00 | 30

9103 | WARREN | SALESMAN | 7698 | 22-DEC-10 00:00:00 | 5250.00 | 2500.00 | 30

9104 | JONES, JR. | MANAGER | 7839 | 02-APR-09 00:00:00 | 7975.00 | | 20

(4 rows)

The following example is a control file that loads the same rows into the emp table, but uses a data file containing fixed-width fields:

LOAD DATA

INFILE 'emp\_fixed.dat'

BADFILE 'emp\_fixed.bad'

APPEND

INTO TABLE emp

TRAILING NULLCOLS

(

empno POSITION (1:4),

ename POSITION (5:14),

job POSITION (15:23),

mgr POSITION (24:27),

hiredate POSITION (28:38),

sal POSITION (39:46),

deptno POSITION (47:48),

comm POSITION (49:56)

)

In the preceding control file, the FIELDS TERMINATED BY and OPTIONALLY ENCLOSED BY clauses are absent. Instead, each field now includes the POSITION clause.

The following is the corresponding data file containing fixed-width fields:

9101ROGERS CLERK 790217-DEC-10 1980.0020

9102PETERSON SALESMAN 769820-DEC-10 2600.0030 2300.00

9103WARREN SALESMAN 769822-DEC-10 5250.0030 2500.00

9104JONES, JR.MANAGER 783902-APR-09 7975.0020

The following control file illustrates the use of the FILLER clause in the data fields for the sal and comm columns. EDB\*Loader ignores the values in these fields and sets the corresponding columns to null.

LOAD DATA

INFILE 'emp\_fixed.dat'

BADFILE 'emp\_fixed.bad'

APPEND

INTO TABLE emp

TRAILING NULLCOLS

(

empno POSITION (1:4),

ename POSITION (5:14),

job POSITION (15:23),

mgr POSITION (24:27),

hiredate POSITION (28:38),

sal FILLER POSITION (39:46),

deptno POSITION (47:48),

comm FILLER POSITION (49:56)

)

Using the same fixed-width data file as in the prior example, the resulting rows in the table appear as follows:

SELECT \* FROM emp WHERE empno > 9100;

empno | ename | job | mgr | hiredate | sal | comm | deptno

-------+------------------+----------+------+--------------------+-----+------+--------

9101 | ROGERS | CLERK | 7902 | 17-DEC-10 00:00:00 | | | 20

9102 | PETERSON | SALESMAN | 7698 | 20-DEC-10 00:00:00 | | | 30

9103 | WARREN | SALESMAN | 7698 | 22-DEC-10 00:00:00 | | | 30

9104 | JONES, JR. | MANAGER | 7839 | 02-APR-09 00:00:00 | | | 20

(4 rows)

The following example illustrates the use of multiple INTO TABLE clauses. For this example, two empty tables are created with the same data definition as the emp table. The following CREATE TABLE commands create these two empty tables, while inserting no rows from the original emp table:

CREATE TABLE emp\_research AS SELECT \* FROM emp WHERE deptno = 99;

CREATE TABLE emp\_sales AS SELECT \* FROM emp WHERE deptno = 99;

The following control file contains two INTO TABLE clauses. Also note that there is no APPEND clause so the default operation of INSERT is used, which requires that tables emp\_research and emp\_sales be empty.

LOAD DATA

INFILE 'emp\_multitbl.dat'

BADFILE 'emp\_multitbl.bad'

DISCARDFILE 'emp\_multitbl.dsc'

INTO TABLE emp\_research

WHEN (47:48) = '20'

TRAILING NULLCOLS

(

empno POSITION (1:4),

ename POSITION (5:14),

job POSITION (15:23),

mgr POSITION (24:27),

hiredate POSITION (28:38),

sal POSITION (39:46),

deptno CONSTANT '20',

comm POSITION (49:56)

)

INTO TABLE emp\_sales

WHEN (47:48) = '30'

TRAILING NULLCOLS

(

empno POSITION (1:4),

ename POSITION (5:14),

job POSITION (15:23),

mgr POSITION (24:27),

hiredate POSITION (28:38),

sal POSITION (39:46),

deptno CONSTANT '30',

comm POSITION (49:56) "ROUND(:comm + (:sal \* .25), 0)"

)

The WHEN clauses specify that when the field designated by columns 47 thru 48 contains 20, the record is inserted into the emp\_research table and when that same field contains 30, the record is inserted into the emp\_sales table. If neither condition is true, the record is written to the discard file named emp\_multitbl.dsc.

The CONSTANT clause is given for column deptno so the specified constant value is inserted into deptno for each record. When the CONSTANT clause is used, it must be the only clause in the field definition other than the column name to which the constant value is assigned.

Finally, column comm of the emp\_sales table is assigned a SQL expression. Column names may be referenced in the expression by prefixing the column name with a colon character (:).

The following is the corresponding data file:

9101ROGERS CLERK 790217-DEC-10 1980.0020

9102PETERSON SALESMAN 769820-DEC-10 2600.0030 2300.00

9103WARREN SALESMAN 769822-DEC-10 5250.0030 2500.00

9104JONES, JR.MANAGER 783902-APR-09 7975.0020

9105ARNOLDS CLERK 778213-SEP-10 3750.0010

9106JACKSON ANALYST 756603-JAN-11 4500.0040

Since the records for employees ARNOLDS and JACKSON contain 10 and 40 in columns 47 thru 48, which do not satisfy any of the WHEN clauses, EDB\*Loader writes these two records to the discard file, emp\_multitbl.dsc, whose content is shown by the following:

9105ARNOLDS CLERK 778213-SEP-10 3750.0010

9106JACKSON ANALYST 756603-JAN-11 4500.0040

The following are the rows loaded into the emp\_research and emp\_sales tables:

SELECT \* FROM emp\_research;

empno | ename | job | mgr | hiredate | sal | comm | deptno

-------+------------+---------+------+--------------------+---------+------+--------

9101 | ROGERS | CLERK | 7902 | 17-DEC-10 00:00:00 | 1980.00 | | 20.00

9104 | JONES, JR. | MANAGER | 7839 | 02-APR-09 00:00:00 | 7975.00 | | 20.00

(2 rows)

SELECT \* FROM emp\_sales;

empno | ename | job | mgr | hiredate | sal | comm | deptno

-------+----------+----------+------+--------------------+---------+---------+--------

9102 | PETERSON | SALESMAN | 7698 | 20-DEC-10 00:00:00 | 2600.00 | 2950.00 | 30.00

9103 | WARREN | SALESMAN | 7698 | 22-DEC-10 00:00:00 | 5250.00 | 3813.00 | 30.00

(2 rows)

### Invoking EDB\*Loader

You must have superuser privileges to run EDB\*Loader. Use the following command to invoke EDB\*Loader from the command line:

edbldr [ -d dbname ] [ -p port ] [ -h host ]

[ USERID={ username/password | username/ | username | / } ]

CONTROL=control\_file

[ DATA=data\_file ]

[ BAD=bad\_file ]

[ DISCARD=discard\_file ]

[ DISCARDMAX=max\_discard\_recs ]

[ LOG=log\_file ]

[ PARFILE=param\_file ]

[ DIRECT={ FALSE | TRUE } ]

[ FREEZE={ FALSE | TRUE } ]

[ ERRORS=error\_count ]

[ PARALLEL={ FALSE | TRUE } ]

[ ROWS=*n* ]

[ SKIP=skip\_count ]

[ SKIP\_INDEX\_MAINTENANCE={ FALSE | TRUE } ]

[ edb\_resource\_group=group\_name ]

Description

If the -d option, the -p option, or the -h option are omitted, the defaults for the database, port, and host are determined according to the same rules as other Advanced Server utility programs such as edb-psql, for example.

Any parameter listed in the preceding syntax diagram except for the -d option, -p option, -h option, and the PARFILE parameter may be specified in a parameter file. The parameter file is specified on the command line when edbldr is invoked using PARFILE=param\_file. Some parameters may be specified in the OPTIONS clause in the control file. See the description of the control file in Section 6.1.3.

The specification of control\_file, data\_file, bad\_file, discard\_file, log\_file, and param\_file may include the full directory path or a relative directory path to the file name. If the file name is specified alone or with a relative directory path, the file is assumed to exist (in the case of control\_file, data\_file, or param\_file), or to be created (in the case of bad\_file, discard\_file, or log\_file) relative to the current working directory from which edbldr is invoked.

Note: The control file must exist in the character set encoding of the client where edbldr is invoked. If the client is in a different encoding than the database encoding, then the PGCLIENTENCODING environment variable must be set on the client to the client’s encoding prior to invoking edbldr. This must be done to ensure character set conversion is properly done between the client and the database server.

The operating system account used to invoke edbldr must have read permission on the directories and files specified by control\_file, data\_file, and param\_file.

The operating system account enterprisedb must have write permission on the directories where bad\_file, discard\_file, and log\_file are to be written.

Note: It is suggested that the file names for control\_file, data\_file, bad\_file, discard\_file, and log\_file include extensions of .ctl, .dat, .bad, .dsc, and .log, respectively. If the provided file name does not contain an extension, EDB\*Loader assumes the actual file name includes the appropriate aforementioned extension.

Parameters

dbname

Name of the database containing the tables to be loaded.

port

Port number on which the database server is accepting connections.

host

IP address of the host on which the database server is running.

USERID={ username/password | username/ | username | / }

EDB\*Loader connects to the database with username. username must be a superuser. password is the password for username.

If the USERID parameter is omitted, EDB\*Loader prompts for username and password. If USERID=username/ is specified, then EDB\*Loader 1) uses the password file specified by environment variable PGPASSFILE if PGPASSFILE is set, or 2) uses the .pgpass password file (pgpass.conf on Windows systems) if PGPASSFILE is not set. If USERID=username is specified, then EDB\*Loader prompts for password. If USERID=/ is specified, the connection is attempted using the operating system account as the user name.

Note: The Advanced Server connection environment variables PGUSER and PGPASSWORD are ignored by EDB\*Loader. Please refer to the PostgreSQL core documentation for information on the PGPASSFILE environment variable and the password file.

CONTROL=control\_file

*control\_file* specifies the name of the control file containing EDB\*Loader directives. If a file extension is not specified, an extension of .ctl is assumed. See Section 6.1.3 for a description of the control file.

DATA=data\_file

*data\_file* specifies the name of the file containing the data to be loaded into the target table. If a file extension is not specified, an extension of .dat is assumed. See Section 6.1.3 for a description of the data\_file.

Note: Specifying a data\_file on the command line overrides the INFILE clause specified in the control file.

BAD=bad\_file

*bad\_file* specifies the name of a file that receives input data records that cannot be loaded due to errors. See Section 6.1.3 for a description of the *bad\_file*.

Note: Specifying a *bad\_file* on the command line overrides any BADFILE clause specified in the control file.

DISCARD=discard\_file

*discard\_file* is the name of the file that receives input data records that do not meet any table’s selection criteria. See the description of discard\_file in Section 6.1.3.

Note: Specifying a discard\_file using the command line DISCARD parameter overrides the DISCARDFILE clause in the control file.

DISCARDMAX=max\_discard\_recs

*max\_discard\_recs* is the maximum number of discarded records that may be encountered from the input data records before terminating the EDB\*Loader session. See the description of max\_discard\_recs in Section 6.1.3.

Note: Specifying max\_discard\_recs using the command line DISCARDMAX parameter overrides the DISCARDMAX or DISCARDS clause in the control file.

LOG=log\_file

*log\_file* specifies the name of the file in which EDB\*Loader records the results of the EDB\*Loader session.

If the LOG parameter is omitted, EDB\*Loader creates a log file with the name control\_file\_base.log in the directory from which edbldr is invoked. control\_file\_base is the base name of the control file used in the EDB\*Loader session. The operating system account enterprisedb must have write permission on the directory where the log file is to be written.

PARFILE=param\_file

*param\_file* specifies the name of the file that contains command line parameters for the EDB\*Loader session. Any command line parameter listed in this section except for the -d, -p, and -h options, and the PARFILE parameter itself, can be specified in param\_file instead of on the command line.

Any parameter given in param\_file overrides the same parameter supplied on the command line before the PARFILE option. Any parameter given on the command line that appears after the PARFILE option overrides the same parameter given in param\_file.

Note: Unlike other EDB\*Loader files, there is no default file name or extension assumed for param\_file, though by Oracle SQL\*Loader convention, .par is typically used, but not required, as an extension.

DIRECT= { FALSE | TRUE }

If DIRECT is set to TRUE EDB\*Loader performs a direct path load instead of a conventional path load. The default value of DIRECT is FALSE.

See Section 6.1.5 for information on direct path loads.

FREEZE= { FALSE | TRUE }

Set FREEZE to TRUE to indicate that the data should be copied with the rows *frozen*. A tuple guaranteed to be visible to all current and future transactions is marked as frozen to prevent transaction ID wrap-around. For more information about frozen tuples, please refer to the PostgreSQL core documentation at:

<http://www.postgresql.org/docs/9.5/static/routine-vacuuming.html>

You must specify a data-loading type of TRUNCATE in the control file when using the FREEZE option. FREEZE is not supported for direct loading.

By default, FREEZE is FALSE.

ERRORS=error\_count

*error\_count* specifies the number of errors permitted before aborting the EDB\*Loader session. The default is 50.

PARALLEL= { FALSE | TRUE }

Set PARALLEL to TRUE to indicate that this EDB\*Loader session is one of a number of concurrent EDB\*Loader sessions participating in a parallel direct path load. The default value of PARALLEL is FALSE.

When PARALLEL is TRUE, the DIRECT parameter must also be set to TRUE . See Section 6.1.6 for more information about parallel direct path loads.

ROWS=*n*

*n* specifies the number of rows that EDB\*Loader will commit before loading the next set of *n* rows.

SKIP=skip\_count

Number of records at the beginning of the input data file that should be skipped before loading begins. The default is 0.

SKIP\_INDEX\_MAINTENANCE= { FALSE | TRUE }

If set to TRUE, index maintenance is not performed as part of a direct path load, and indexes on the loaded table are marked as invalid. The default value of SKIP\_INDEX\_MAINTENANCE is FALSE.

Please note: During a parallel direct path load, target table indexes are not updated, and are marked as invalid after the load is complete.

You can use the REINDEX command to rebuild an index. For more information about the REINDEX command, please refer to the PostgreSQL core documentation available at:

<http://www.postgresql.org/docs/9.5/static/sql-reindex.html>

edb\_resource\_group=group\_name

group\_name specifies the name of an EDB Resource Manager resource group to which the EDB\*Loader session is to be assigned.

Any default resource group that may have been assigned to the session (for example, a database user running the EDB\*Loader session who had been assigned a default resource group with the ALTER ROLE ... SET edb\_resource\_group command) is overridden by the resource group given by the edb\_resource\_group parameter specified on the edbldr command line.

For information about the EDB Resource Manager, see Chapter 5, *EDB Resource Manager.*

Examples

In the following example EDB\*Loader is invoked using a control file named emp.ctl located in the current working directory to load a table in database edb:

$ /opt/PostgresPlus/9.5AS/bin/edbldr -d edb USERID=enterprisedb/password CONTROL=emp.ctl

EDB\*Loader: Copyright (c) 2007-2014, EnterpriseDB Corporation.

Successfully loaded (4) records

In the following example, EDB\*Loader prompts for the user name and password since they are omitted from the command line. In addition, the files for the bad file and log file are specified with the BAD and LOG command line parameters.

$ /opt/PostgresPlus/9.5AS/bin/edbldr -d edb CONTROL=emp.ctl BAD=/tmp/emp.bad LOG=/tmp/emp.log

Enter the user name : enterprisedb

Enter the password :

EDB\*Loader: Copyright (c) 2007-2014, EnterpriseDB Corporation.

Successfully loaded (4) records

The following example runs EDB\*Loader with the same parameters as shown in the preceding example, but using a parameter file located in the current working directory. The SKIP and ERRORS parameters are altered from their defaults in the parameter file as well.

The parameter file, emp.par, contains the following:

CONTROL=emp.ctl

BAD=/tmp/emp.bad

LOG=/tmp/emp.log

SKIP=1

ERRORS=10

EDB\*Loader is invoked with the parameter file as shown by the following:

$ /opt/PostgresPlus/9.5AS/bin/edbldr -d edb PARFILE=emp.par

Enter the user name : enterprisedb

Enter the password :

EDB\*Loader: Copyright (c) 2007-2014, EnterpriseDB Corporation.

Successfully loaded (3) records

#### Exit Codes

When EDB\*Loader exits, it will return one of the following codes:

|  |  |
| --- | --- |
| Exit Code | Description |
| 0 | Indicates that all rows loaded successfully. |
| 1 | Indicates that EDB\*Loader encountered command line or syntax errors, or aborted the load operation due to an unrecoverable error. |
| 2 | Indicates that the load completed, but some (or all) rows were rejected or discarded. |
| 3 | Indicates that EDB\*Loader encountered fatal errors (such as OS errors). This class of errors is equivalent to the FATAL or PANIC severity levels of PostgreSQL errors. |

### Direct Path Load

During a direct path load, EDB\*Loader writes the data directly to the database pages, which is then synchronized to disk. The insert processing associated with a conventional path load is bypassed, thereby resulting in a performance improvement.

Bypassing insert processing reduces the types of constraints that may exist on the target table. The following types of constraints are permitted on the target table of a direct path load:

* Primary key
* Not null constraints
* Indexes (unique or non-unique)

The restrictions on the target table of a direct path load are the following:

* Triggers are not permitted
* Check constraints are not permitted
* Foreign key constraints on the target table referencing another table are not permitted
* Foreign key constraints on other tables referencing the target table are not permitted
* The table must not be partitioned
* Rules may exist on the target table, but they are not executed

Note: Currently, a direct path load in EDB\*Loader is more restrictive than in Oracle SQL\*Loader. The preceding restrictions do not apply to Oracle SQL\*Loader in most cases. The following restrictions apply to a control file used in a direct path load:

* Multiple table loads are not supported. That is, only one INTO TABLE clause may be specified in the control file.
* SQL expressions may not be used in the data field definitions of the INTO TABLE clause.
* The FREEZE option is not supported for direct path loading.

To run a direct path load, add the DIRECT=TRUE option as shown by the following example:

$ /opt/PostgresPlus/9.5AS/bin/edbldr -d edb USERID=enterprisedb/password CONTROL=emp.ctl DIRECT=TRUE

EDB\*Loader: Copyright (c) 2007-2014, EnterpriseDB Corporation.

Successfully loaded (4) records

### Parallel Direct Path Load

The performance of a direct path load can be further improved by distributing the loading process over two or more sessions running concurrently. Each session runs a direct path load into the same table.

Since the same table is loaded from multiple sessions, the input records to be loaded into the table must be divided amongst several data files so that each EDB\*Loader session uses its own data file and the same record is not loaded more than once into the table.

The target table of a parallel direct path load is under the same restrictions as a direct path load run in a single session.

The restrictions on the target table of a direct path load are the following:

* Triggers are not permitted
* Check constraints are not permitted
* Foreign key constraints on the target table referencing another table are not permitted
* Foreign key constraints on other tables referencing the target table are not permitted
* The table must not be partitioned
* Rules may exist on the target table, but they are not executed

In addition, the APPEND clause must be specified in the control file used by each EDB\*Loader session.

To run a parallel direct path load, run EDB\*Loader in a separate session for each participant of the parallel direct path load. Invocation of each such EDB\*Loader session must include the DIRECT=TRUE and PARALLEL=TRUE parameters.

Each EDB\*Loader session runs as an independent transaction so if one of the parallel sessions aborts and rolls back its changes, the loading done by the other parallel sessions are not affected.

Note: In a parallel direct path load, each EDB\*Loader session reserves a fixed number of blocks in the target table in a round-robin fashion. Some of the blocks in the last allocated chunk may not be used, and those blocks remain uninitialized. A subsequent use of the VACUUM command on the target table may show warnings regarding these uninitialized blocks such as the following:

WARNING: relation "emp" page 98264 is uninitialized --- fixing

WARNING: relation "emp" page 98265 is uninitialized --- fixing

WARNING: relation "emp" page 98266 is uninitialized --- fixing

This is an expected behavior and does not indicate data corruption.

Indexes on the target table are not updated during a parallel direct path load and are therefore marked as invalid after the load is complete. You must use the REINDEX command to rebuild the indexes.

The following example shows the use of a parallel direct path load on the emp table.

Note: If you attempt a parallel direct path load on the sample emp table provided with Advanced Server, you must first remove the triggers and constraints referencing the emp table. In addition the primary key column, empno, was expanded from NUMBER(4) to NUMBER in this example to allow for the insertion of a larger number of rows.

The following is the control file used in the first session:

LOAD DATA

INFILE '/home/user/loader/emp\_parallel\_1.dat'

APPEND

INTO TABLE emp

FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"'

TRAILING NULLCOLS

(

empno,

ename,

job,

mgr,

hiredate,

sal,

deptno,

comm

)

The APPEND clause must be specified in the control file for a parallel direct path load.

The following shows the invocation of EDB\*Loader in the first session. The DIRECT=TRUE and PARALLEL=TRUE parameters must be specified.

$ /opt/PostgresPlus/9.5AS/bin/edbldr -d edb USERID=enterprisedb/password CONTROL=emp\_parallel\_1.ctl DIRECT=TRUE PARALLEL=TRUE

WARNING: index maintenance will be skipped with PARALLEL load

EDB\*Loader: Copyright (c) 2007-2014, EnterpriseDB Corporation.

The control file used for the second session appears as follows. Note that it is the same as the one used in the first session, but uses a different data file.

LOAD DATA

INFILE '/home/user/loader/emp\_parallel\_2.dat'

APPEND

INTO TABLE emp

FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"'

TRAILING NULLCOLS

(

empno,

ename,

job,

mgr,

hiredate,

sal,

deptno,

comm

)

The preceding control file is used in a second session as shown by the following:

$ /opt/PostgresPlus/9.5AS/bin/edbldr -d edb USERID=enterprisedb/password CONTROL=emp\_parallel\_2.ctl DIRECT=TRUE PARALLEL=TRUE

WARNING: index maintenance will be skipped with PARALLEL load

EDB\*Loader: Copyright (c) 2007-2014, EnterpriseDB Corporation.

EDB\*Loader displays the following message in each session when its respective load operation completes:

Successfully loaded (10000) records

The following query shows that the index on the emp table has been marked as INVALID:

SELECT index\_name, status FROM user\_indexes WHERE table\_name = 'EMP';

index\_name | status

------------+---------

EMP\_PK | INVALID

(1 row)

Note: user\_indexes is the view of indexes compatible with oracle databases owned by the current user.

Queries on the emp table will not utilize the index unless it is rebuilt using the REINDEX command as shown by the following:

REINDEX INDEX emp\_pk;

A subsequent query on user\_indexes shows that the index is now marked as VALID:

SELECT index\_name, status FROM user\_indexes WHERE table\_name = 'EMP';

index\_name | status

------------+--------

EMP\_PK | VALID

(1 row)

### Remote Loading

EDB\*Loader supports a feature called remote loading. In remote loading, the database containing the table to be loaded is running on a database server on a different host than from where EDB\*Loader is invoked with the input data source.

This feature is useful if you have a large amount of data to be loaded, and you do not want to create a large data file on the host running the database server.

In addition, you can use the standard input feature to pipe the data from the data source such as another program or script, directly to EDB\*Loader, which then loads the table in the remote database. This bypasses the process of having to create a data file on disk for EDB\*Loader.

Performing remote loading along with using standard input requires the following:

* The edbldr program must be installed on the client host on which it is to be invoked with the data source for the EDB\*Loader session.
* The control file must contain the clause INFILE 'stdin' so you can pipe the data directly into EDB\*Loader’s standard input. See Section 6.1.3 for information on the INFILE clause and the EDB\*Loader control file.
* All files used by EDB\*Loader such as the control file, bad file, discard file, and log file must reside on, or are created on, the client host on which edbldr is invoked.
* When invoking EDB\*Loader, use the -h option to specify the IP address of the remote database server. See Section 6.1.4 for information on invoking EDB\*Loader.
* Use the operating system pipe operator (|) or input redirection operator (<) to supply the input data to EDB\*Loader.

The following example loads a database running on a database server at 192.168.1.14 using data piped from a source named datasource.

datasource | ./edbldr -d edb -h 192.168.1.14 USERID=enterprisedb/password CONTROL=remote.ctl

The following is another example of how standard input can be used:

./edbldr -d edb -h 192.168.1.14 USERID=enterprisedb/password CONTROL=remote.ctl < datasource

### Updating a Table with a Conventional Path Load

You can use EDB\*Loader with a conventional path load to update the rows within a table, merging new data with the existing data. When you invoke EDB\*Loader to perform an update, the server searches the table for an existing row with a matching primary key:

* If the server locates a row with a matching key, it replaces the existing row with the new row.
* If the server does not locate a row with a matching key, it adds the new row to the table.

To use EDB\*Loader to update a table, the table must have a primary key. Please note that you cannot use EDB\*Loader to UPDATE a partitioned table.

To perform an UPDATE, use the same steps as when performing a conventional path load:

1. Create a data file that contains the rows you wish to UPDATE or INSERT.
2. Define a control file that uses the INFILE keyword to specify the name of the data file. For information about building the EDB\*Loader control file, see Section 6.1.3.
3. Invoke EDB\*Loader, specifying the database name, connection information, and the name of the control file. For information about invoking EDB\*Loader, see Section 6.1.4.

The following example uses the emp table that is distributed with the Advanced Server sample data. By default, the table contains:

edb=# select \* from emp;

empno|ename | job | mgr | hiredate | sal | comm | deptno

-----+------+---------+------+--------------------+---------+-------+--------

7369 |SMITH |CLERK | 7902 | 17-DEC-80 00:00:00 | 800.00 | | 20

7499 |ALLEN |SALESMAN | 7698 | 20-FEB-81 00:00:00 | 1600.00 |300.00 | 30

7521 |WARD |SALESMAN | 7698 | 22-FEB-81 00:00:00 | 1250.00 |500.00 | 30

7566 |JONES |MANAGER | 7839 | 02-APR-81 00:00:00 | 2975.00 | | 20

7654 |MARTIN|SALESMAN | 7698 | 28-SEP-81 00:00:00 | 1250.00 |1400.00| 30

7698 |BLAKE |MANAGER | 7839 | 01-MAY-81 00:00:00 | 2850.00 | | 30

7782 |CLARK |MANAGER | 7839 | 09-JUN-81 00:00:00 | 2450.00 | | 10

7788 |SCOTT |ANALYST | 7566 | 19-APR-87 00:00:00 | 3000.00 | | 20

7839 |KING |PRESIDENT| | 17-NOV-81 00:00:00 | 5000.00 | | 10

7844 |TURNER|SALESMAN | 7698 | 08-SEP-81 00:00:00 | 1500.00 | 0.00 | 30

7876 |ADAMS |CLERK | 7788 | 23-MAY-87 00:00:00 | 1100.00 | | 20

7900 |JAMES |CLERK | 7698 | 03-DEC-81 00:00:00 | 950.00 | | 30

7902 |FORD |ANALYST | 7566 | 03-DEC-81 00:00:00 | 3000.00 | | 20

7934 |MILLER|CLERK | 7782 | 23-JAN-82 00:00:00 | 1300.00 | | 10

(14 rows)

The following control file (emp\_update.ctl) specifies the fields in the table in a comma-delimited list. The control file performs an UPDATE on the emp table:

LOAD DATA

INFILE 'emp\_update.dat'

BADFILE 'emp\_update.bad'

DISCARDFILE 'emp\_update.dsc'

UPDATE INTO TABLE emp

FIELDS TERMINATED BY ","

(empno, ename, job, mgr, hiredate, sal, comm, deptno)

The data that is being updated or inserted is saved in the emp\_update.dat file. emp\_update.dat contains:

7521,WARD,MANAGER,7839,22-FEB-81 00:00:00,3000.00,0.00,30

7566,JONES,MANAGER,7839,02-APR-81 00:00:00,3500.00,0.00,20

7903,BAKER,SALESMAN,7521,10-JUN-13 00:00:00,1800.00,500.00,20

7904,MILLS,SALESMAN,7839,13-JUN-13 00:00:00,1800.00,500.00,20

7654,MARTIN,SALESMAN,7698,28-SEP-81 00:00:00,1500.00,400.00,30

Invoke EDB\*Loader, specifying the name of the database (edb), the name of a database superuser (and their associated password) and the name of the control file (emp\_update.ctl):

edbldr -d edb userid=*user\_name*/*password* control=emp\_update.ctl

After performing the update, the emp table contains:

edb=# select \* from emp;

empno|ename | job | mgr | hiredate | sal | comm | deptno

-----+------+---------+------+--------------------+---------+-------+--------

7369 |SMITH |CLERK | 7902 | 17-DEC-80 00:00:00 | 800.00 | | 20

7499 |ALLEN |SALESMAN | 7698 | 20-FEB-81 00:00:00 | 1600.00 |300.00 | 30

7521 |WARD |MANAGER | 7839 | 22-FEB-81 00:00:00 | 3000.00 |0.00 | 30

7566 |JONES |MANAGER | 7839 | 02-APR-81 00:00:00 | 3500.00 |0.00 | 20

7654 |MARTIN|SALESMAN | 7698 | 28-SEP-81 00:00:00 | 1500.00 |400.00 | 30

7698 |BLAKE |MANAGER | 7839 | 01-MAY-81 00:00:00 | 2850.00 | | 30

7782 |CLARK |MANAGER | 7839 | 09-JUN-81 00:00:00 | 2450.00 | | 10

7788 |SCOTT |ANALYST | 7566 | 19-APR-87 00:00:00 | 3000.00 | | 20

7839 |KING |PRESIDENT| | 17-NOV-81 00:00:00 | 5000.00 | | 10

7844 |TURNER|SALESMAN | 7698 | 08-SEP-81 00:00:00 | 1500.00 | 0.00 | 30

7876 |ADAMS |CLERK | 7788 | 23-MAY-87 00:00:00 | 1100.00 | | 20

7900 |JAMES |CLERK | 7698 | 03-DEC-81 00:00:00 | 950.00 | | 30

7902 |FORD |ANALYST | 7566 | 03-DEC-81 00:00:00 | 3000.00 | | 20

7903 |BAKER |SALESMAN |7521 | 10-JUN-13 00:00:00 | 1800.00 |500.00 | 20

7904 |MILLS |SALESMAN |7839 |13-JUN-13 00:00:00 |1800.00 |500.00 | 20

7934 |MILLER|CLERK | 7782 | 23-JAN-82 00:00:00 | 1300.00 | | 10

(16 rows)

The rows containing information for the three employees that are currently in the emp table are updated, while rows are added for the new employees (BAKER and MILLS).

### Loading Empty Strings with EDB\*Loader

Advanced Server includes a configuration parameter that controls how EDB\*Loader handles a CSV (comma-separated value) file containing empty strings. An empty string within a CSV file may take the form of:

* an unquoted empty string. For example:

9001, , 40

* a single-quoted, comma-delimited value. For example:

9001, '', 40

* a double-quoted, comma-delimited value. For example:

9001, "", 40

You can use the edbldr.empty\_csv\_field parameter to specify how EDB\*Loader will treat an empty string. The valid values for the edbldr.empty\_csv\_field parameter are:

|  |  |
| --- | --- |
| Parameter Setting | EDB\*Loader Behavior |
| NULL | An empty field is treated as NULL. |
| empty\_string | An empty field is treated as a string of length zero. |
| pgsql | An empty field is treated as a NULL if it does not contain quotes and as an empty string if it contains quotes. |

You can set the edbldr.empty\_csv\_field parameter in any context (i.e. with a SET statement, or in the postgresql.conf file). You can also use the PGOPTIONS environment variable to set the value of edbldr.empty\_csv\_field for an EDB\*Loader session. For example, before invoking EDB\*Loader, enter the command:

$ export PGOPTIONS="-c edbldr.empty\_csv\_field=empty\_string"

Then, invoke EDB\*Loader, specifying command line options as required.

For more information about setting parameter values, please refer to the PostgreSQL core documentation available at:

<http://www.enterprisedb.com/products-services-training/products/documentation/enterpriseedition>

## EDB\*Plus

EDB\*Plus is a utility program that provides a command line user interface to the Advanced Server. EDB\*Plus accepts SQL commands, SPL anonymous blocks, and EDB\*Plus commands. EDB\*Plus provides various capabilities including:

* Querying certain database objects
* Executing stored procedures
* Formatting output from SQL commands
* Executing batch scripts
* Executing OS commands
* Recording output

The following section describes how to connect to an Advanced Server database using EDB\*Plus. The final section provides a summary of the EDB\*Plus commands.

### Starting EDB\*Plus

To open an EDB\*Plus command line, navigate through the Applications (or Start) menu to the Postgres Plus Advanced Server menu, to the Run SQL Command Line menu, and select the EDB\*Plus option. You can also invoke EDB\*Plus from the operating system command line with the following command:

edbplus [ -S[ILENT ] ] [ login | /NOLOG ] [ @scriptfile[.ext ] ]

-SILENT

If specified, the EDB\*Plus sign-on banner is suppressed along with all prompts.

login

Login information for connecting to the database server and database. login takes the following format. (There must be no white space within the login information.)

username[/password][@{connectstring | variable } ]

Where:

username is a database username with which to connect to the database.

password is the password associated with the specified username. If a password is not provided, but a password is required for authentication, EDB\*Plus will prompt for the password.

connectstring is the database connection string.

variable is a variable defined in the login.sql file that contains a database connection string. The login.sql file can be found in the edbplus subdirectory of the Advanced Server home directory.

host[:port][/dbname ] ]

host is the hostname on which the database server resides. If neither @connectstring nor @variable nor /NOLOG is specified, the default host is assumed to be the localhost. port is the port number receiving connections on the database server. If not specified, the default is 5444. dbname is the name of the database to connect to. If not specified the default is edb.

/NOLOG

Specify /NOLOG to start EDB\*Plus without establishing a database connection. SQL commands and EDB\*Plus commands that require a database connection cannot be used in this mode. The CONNECT command can be subsequently given to connect to a database after starting EDB\*Plus with the /NOLOG option.

scriptfile[.ext ]

scriptfile is the name of a file residing in the current working directory, containing SQL and/or EDB\*Plus commands that will be automatically executed after startup of EDB\*Plus. ext is the filename extension. If the filename extension is sql, then the .sql extension may be omitted when specifying scriptfile. When creating a script file, always name the file with an extension, otherwise it will not be accessible by EDB\*Plus. (EDB\*Plus will always assume a .sql extension on filenames that are specified with no extension.)

The following example shows user enterprisedb with password, password, connecting to database edb running on a database server on the localhost at port 5444.

C:\Program Files (x86)\PostgresPlus\9.5AS\edbplus>edbplus enterprisedb/password

Connected to EnterpriseDB 9.5.0.0 (localhost:5444/edb) AS enterprisedb

EDB\*Plus: Release 9.5

Copyright (c) 2008-2016, EnterpriseDB Corporation. All rights reserved.

SQL>

The following example shows user enterprisedb with password, password, connecting to database edb running on a database server on the localhost at port 5445.

C:\Program Files (x86)\PostgresPlus\9.5AS\edbplus>edbplus enterprisedb/password@localhost:5445/edb

Connected to EnterpriseDB 9.5.0.0 (localhost:5445/edb) AS enterprisedb

EDB\*Plus: Release 9.5

Copyright (c) 2008-2016, EnterpriseDB Corporation. All rights reserved.

SQL>

Using variable hr\_5445 in the login.sql file, the following illustrates how it is used to connect to database hr on localhost at port 5445.

C:\Program Files (x86)\PostgresPlus\9.5AS\edbplus>edbplus enterprisedb/password@hr\_5445

Connected to EnterpriseDB 9.5.0.0 (localhost:5445/hr) AS enterprisedb

EDB\*Plus: Release 9.5 (Build 28)

Copyright (c) 2008-2016, EnterpriseDB Corporation. All rights reserved.

SQL>

The following is the content of the login.sql file used in the previous example.

define edb="localhost:5445/edb"

define hr\_5445="localhost:5445/hr"

The following example executes a script file, dept\_query.sql after connecting to database edb on server localhost at port 5444.

C:\Program Files (x86)\PostgresPlus\9.5AS\edbplus>edbplus enterprisedb/password @dept\_query

Connected to EnterpriseDB 9.5.0.0 (localhost:5444/edb) AS enterprisedb

SQL> SELECT \* FROM dept;

DEPTNO DNAME LOC

------ -------------- -------------

10 ACCOUNTING NEW YORK

20 RESEARCH DALLAS

30 SALES CHICAGO

40 OPERATIONS BOSTON

SQL> EXIT

Disconnected from EnterpriseDB Database.

The following is the content of file dept\_query.sql used in the previous example.

SET PAGESIZE 9999

SET ECHO ON

SELECT \* FROM dept;

EXIT

### Command Summary

This section contains a summary of EDB\*Plus commands.

#### ACCEPT

The ACCEPT command displays a prompt and waits for the user’s keyboard input. The value input by the user is placed in the specified variable.

ACC[EPT ] variable

The following example creates a new variable named my\_name, accepts a value of John Smith, then displays the value using the DEFINE command.

SQL> ACCEPT my\_name

Enter value for my\_name: John Smith

SQL> DEFINE my\_name

DEFINE MY\_NAME = "John Smith"

#### APPEND

APPEND is a line editor command that appends the given text to the end of the current line in the SQL buffer.

A[PPEND ] text

In the following example, a SELECT command is built-in the SQL buffer using the APPEND command. Note that two spaces are placed between the APPEND command and the WHERE clause in order to separate dept and WHERE by one space in the SQL buffer.

SQL> APPEND SELECT \* FROM dept

SQL> LIST

1\* SELECT \* FROM dept

SQL> APPEND WHERE deptno = 10

SQL> LIST

1\* SELECT \* FROM dept WHERE deptno = 10

#### CHANGE

CHANGE is a line editor command performs a search-and-replace on the current line in the SQL buffer.

C[HANGE ] /from/[to/ ]

If to/ is specified, the first occurrence of text from in the current line is changed to text to. If to/ is omitted, the first occurrence of text from in the current line is deleted.

The following sequence of commands makes line 3 the current line, then changes the department number in the WHERE clause from 20 to 30.

SQL> LIST

1 SELECT empno, ename, job, sal, comm

2 FROM emp

3 WHERE deptno = 20

4\* ORDER BY empno

SQL> 3

3\* WHERE deptno = 20

SQL> CHANGE /20/30/

3\* WHERE deptno = 30

SQL> LIST

1 SELECT empno, ename, job, sal, comm

2 FROM emp

3 WHERE deptno = 30

4\* ORDER BY empno

#### CLEAR

The CLEAR command removes the contents of the SQL buffer, deletes all column definitions set with the COLUMN command, or clears the screen.

CL[EAR ] [ BUFF[ER ] | SQL | COL[UMNS ] | SCR[EEN ] ]

|  |  |
| --- | --- |
| BUFFER | SQL | Clears the SQL buffer. |
| COLUMNS | Removes column definitions. |
| SCREEN | Clears the screen. This is the default if no options are specified. |

#### COLUMN

The COLUMN command controls output formatting. The formatting attributes set by using the COLUMN command remain in effect only for the duration of the current session.

COL[UMN ]

[ column

{ CLE[AR ] |

{ FOR[MAT ] spec |

HEA[DING ] text |

{ OFF | ON }

} [...]

}

]

If the COLUMN command is specified with no subsequent options, formatting options for current columns in effect for the session are displayed.

If the COLUMN command is followed by a column name, then the column name may be followed by one of the following:

* No other options
* CLEAR
* Any combination of FORMAT, HEADING, and one of OFF or ON

column

Name of a column in a table to which subsequent column formatting options are to apply. If no other options follow column, then the current column formatting options if any, of column are displayed.

CLEAR

The CLEAR option reverts all formatting options back to their defaults for column. If the CLEAR option is specified, it must be the only option specified.

spec

Format specification to be applied to column. For character columns, spec takes the following format:

n

n is a positive integer that specifies the column width in characters within which to display the data. Data in excess of n will wrap around with the specified column width.

For numeric columns, spec is comprised of the following elements.

Table 10‑‑ Numeric Column Format Elements

| Element | Description |
| --- | --- |
| $ | Display a leading dollar sign. |
| , | Display a comma in the indicated position. |
| . | Marks the location of the decimal point. |
| 0 | Display leading zeros. |
| 9 | Number of significant digits to display. |

If loss of significant digits occurs due to overflow of the format, then all #’s are displayed.

text

Text to be used for the column heading of column.

OFF | ON

If OFF is specified, formatting options are reverted back to their defaults, but are still available within the session. If ON is specified, the formatting options specified by previous COLUMN commands for column within the session are re-activated.

The following example shows the effect of changing the display width of the job column.

SQL> SET PAGESIZE 9999

SQL> COLUMN job FORMAT A5

SQL> COLUMN job

COLUMN JOB ON

FORMAT A5

wrapped

SQL> SELECT empno, ename, job FROM emp;

EMPNO ENAME JOB

----- ---------- -----

7369 SMITH CLERK

7499 ALLEN SALES

MAN

7521 WARD SALES

MAN

7566 JONES MANAG

ER

7654 MARTIN SALES

MAN

7698 BLAKE MANAG

ER

7782 CLARK MANAG

ER

7788 SCOTT ANALY

ST

7839 KING PRESI

DENT

7844 TURNER SALES

MAN

7876 ADAMS CLERK

7900 JAMES CLERK

7902 FORD ANALY

ST

7934 MILLER CLERK

14 rows retrieved.

The following example applies a format to the sal column.

SQL> COLUMN sal FORMAT $99,999.00

SQL> COLUMN

COLUMN JOB ON

FORMAT A5

wrapped

COLUMN SAL ON

FORMAT $99,999.00

wrapped

SQL> SELECT empno, ename, job, sal FROM emp;

EMPNO ENAME JOB SAL

----- ---------- ----- -----------

7369 SMITH CLERK $800.00

7499 ALLEN SALES $1,600.00

MAN

7521 WARD SALES $1,250.00

MAN

7566 JONES MANAG $2,975.00

ER

7654 MARTIN SALES $1,250.00

MAN

7698 BLAKE MANAG $2,850.00

ER

7782 CLARK MANAG $2,450.00

ER

7788 SCOTT ANALY $3,000.00

ST

7839 KING PRESI $5,000.00

DENT

7844 TURNER SALES $1,500.00

MAN

7876 ADAMS CLERK $1,100.00

7900 JAMES CLERK $950.00

7902 FORD ANALY $3,000.00

ST

7934 MILLER CLERK $1,300.00

14 rows retrieved.

#### CONNECT

Change the database connection to a different user and/or connect to a different database. There must be no white space between any of the parameters following the CONNECT command.

CON[NECT] username[/password][@{connectstring | variable } ]

Where:

username is a database username with which to connect to the database.

password is the password associated with the specified username. If a password is not provided, but a password is required for authentication, EDB\*Plus will prompt for the password.

connectstring is the database connection string.

variable is a variable defined in the login.sql file that contains a database connection string. The login.sql file can be found in the edbplus subdirectory of the Advanced Server home directory.

In the following example, the database connection is changed to database edb on the localhost at port 5445 with username, smith.

SQL> CONNECT smith/mypassword@localhost:5445/edb

Disconnected from EnterpriseDB Database.

Connected to EnterpriseDB 9.5.0.0 (localhost:5445/edb) AS smith

From within the session shown above, the connection is changed to username enterprisedb. Also note that the host defaults to the localhost, the port defaults to 5444 (which is not the same as the port previously used), and the database defaults to edb.

SQL> CONNECT enterprisedb/password

Disconnected from EnterpriseDB Database.

Connected to EnterpriseDB 9.5.0.0 (localhost:5444/edb) AS enterprisedb

#### DEFINE

The DEFINE command creates or replaces the value of a user variable (also called a substitution variable).

DEF[INE ] [ variable [ = text ] ]

If the DEFINE command is given without any parameters, all current variables and their values are displayed.

If DEFINE variable is given, only variable is displayed with its value.

DEFINE variable = text assigns text to variable. text may be optionally enclosed within single or double quotation marks. Quotation marks must be used if text contains space characters.

The following example defines two variables, dept and name.

SQL> DEFINE dept = 20

SQL> DEFINE name = 'John Smith'

SQL> DEFINE

DEFINE EDB = "localhost:5445/edb"

DEFINE DEPT = "20"

DEFINE NAME = "John Smith"

Note: The variable EDB is read from the login.sql file located in the edbplus subdirectory of the Advanced Server home directory.

#### DEL

DEL is a line editor command that deletes one or more lines from the SQL buffer.

DEL [ n | n m | n \* | n L[AST ] | \* | \* n | \* L[AST ] |

L[AST ] ]

The parameters specify which lines are to be deleted from the SQL buffer. Two parameters specify the start and end of a range of lines to be deleted. If the DEL command is given with no parameters, the current line is deleted.

n

n is an integer representing the nth line

n m

n and m are integers where m is greater than n representing the nth through the mth lines

\*

Current line

LAST

Last line

In the following example, the fifth and sixth lines containing columns sal and comm, respectively, are deleted from the SELECT command in the SQL buffer.

SQL> LIST

1 SELECT

2 empno

3 ,ename

4 ,job

5 ,sal

6 ,comm

7 ,deptno

8\* FROM emp

SQL> DEL 5 6

SQL> LIST

1 SELECT

2 empno

3 ,ename

4 ,job

5 ,deptno

6\* FROM emp

#### DESCRIBE

The DESCRIBE command displays:

* A list of columns, column data types, and column lengths for a table or view
* A list of parameters for a procedure or function
* A list of procedures and functions and their respective parameters for a package.

The DESCRIBE command will also display the structure of the database object referred to by a synonym. The syntax is:

DESC[RIBE] [ schema.]object

schema

Name of the schema containing the object to be described.

object

Name of the table, view, procedure, function, or package to be displayed, or the synonym of an object.

#### DISCONNECT

The DISCONNECT command closes the current database connection, but does not terminate EDB\*Plus.

DISC[ONNECT ]

#### EDIT

The EDIT command invokes an external editor to edit the contents of an operating system file or the SQL buffer.

ED[IT ] [ filename[.ext ] ]

filename[.ext ]

filename is the name of the file to open with an external editor. ext is the filename extension. If the filename extension is sql, then the .sql extension may be omitted when specifying filename. EDIT always assumes a .sql extension on filenames that are specified with no extension. If the filename parameter is omitted from the EDIT command, the contents of the SQL buffer are brought into the editor.

#### EXECUTE

The EXECUTE command executes an SPL procedure from EDB\*Plus.

EXEC[UTE ] spl\_procedure [ ([ parameters ]) ]

spl\_procedure

The name of the SPL procedure to be executed.

parameters

Comma-delimited list of parameters. If there are no parameters, then a pair of empty parentheses may optionally be specified.

#### EXIT

The EXIT command terminates the EDB\*Plus session and returns control to the operating system. QUIT is a synonym for EXIT. Specifying no parameters is equivalent to EXIT SUCCESS COMMIT.

{ EXIT | QUIT }

[ SUCCESS | FAILURE | WARNING | value |variable ]

[ COMMIT | ROLLBACK ]SUCCESS | FAILURE |WARNING

Returns an operating system dependent return code indicating successful operation, failure, or warning for SUCCESS, FAILURE, and WARNING, respectively. The default is SUCCESS.

value

An integer value that is returned as the return code.

variable

A variable created with the DEFINE command whose value is returned as the return code.

COMMIT | ROLLBACK

If COMMIT is specified, uncommitted updates are committed upon exit. If ROLLBACK is specified, uncommitted updates are rolled back upon exit. The default is COMMIT.

#### GET

The GET command loads the contents of the given file to the SQL buffer.

GET filename[.ext ] [ LIS[T ] | NOL[IST ] ]

filename[.ext ]

filename is the name of the file to load into the SQL buffer. ext is the filename extension. If the filename extension is sql, then the .sql extension may be omitted when specifying filename. GET always assumes a .sql extension on filenames that are specified with no extension.

LIST | NOLIST

If LIST is specified, the content of the SQL buffer is displayed after the file is loaded. If NOLIST is specified, no listing is displayed. The default is LIST.

#### HELP

The HELP command obtains an index of topics or help on a specific topic. The question mark (?) is synonymous with specifying HELP.

{ HELP | ? } { INDEX | topic }

INDEX

Displays an index of available topics.

topic

The name of a specific topic – e.g., an EDB\*Plus command, for which help is desired.

#### HOST

The HOST command executes an operating system command from EDB\*Plus.

HO[ST ] [os\_command]

os\_command

The operating system command to be executed. If you do not provide an operating system command, EDB\*Plus pauses execution and opens a new shell prompt. When the shell exits, EDB\*Plus resumes execution.

#### INPUT

The INPUT line editor command adds a line of text to the SQL buffer after the current line.

I[NPUT ] text

The following sequence of INPUT commands constructs a SELECT command.

SQL> INPUT SELECT empno, ename, job, sal, comm

SQL> INPUT FROM emp

SQL> INPUT WHERE deptno = 20

SQL> INPUT ORDER BY empno

SQL> LIST

1 SELECT empno, ename, job, sal, comm

2 FROM emp

3 WHERE deptno = 20

4\* ORDER BY empno

#### LIST

LIST is a line editor command that displays the contents of the SQL buffer.

L[IST] [ n | n m | n \* | n L[AST] | \* | \* n | \* L[AST] |

L[AST] ]

The buffer does not include a history of the EDB\*Plus commands.

n

n represents the buffer line number.

n m

n m displays a list of lines between n and m.

n \*

n \* displays a list of lines that range between line n and the current line.

n L[AST]

n L[AST] displays a list of lines that range from line n through the last line in the buffer.

\*

\* displays the current line.

\* n

\* n displays a list of lines that range from the current line through line n.

\* L[AST]

\* L[AST] displays a list of lines that range from the current line through the last line.

L[AST]

L[AST] displays the last line.

#### PASSWORD

Use the PASSWORD command to change your database password.

PASSW[ORD] [user\_name]

You must have sufficient privileges to use the PASSWORD command to change another user's password. The following example demonstrates using the PASSWORD command to change the password for a user named acctg:

SQL> PASSWORD acctg

Changing password for acctg

New password:

New password again:

Password successfully changed.

#### PAUSE

The PAUSE command displays a message, and waits for the user to press ENTER.

PAU[SE] [optional\_text]

optional\_text specifies the text that will be displayed to the user. If the optional\_text is omitted, Advanced Server will display two blank lines. If you double quote the optional\_text string, the quotes will be included in the output.

#### PRINT

The PRINT command displays the value of a bind variable.

PRI[NT] [bind\_variable\_name]

bind\_variable\_name specifies the name of a bind variable. Omit bind\_variable\_name to generate a list that includes the values of all bind variables.

#### PROMPT

The PROMPT command displays a message to the user before continuing.

PRO[MPT] [message\_text]

message\_text specifies the text displayed to the user. Double quote the string to include quotes in the output.

#### QUIT

The QUIT command terminates the session and returns control to the operating system. QUIT is a synonym for EXIT.

QUIT  
 [SUCCESS | FAILURE | WARNING | value | sub\_variable]  
 [COMMIT | ROLLBACK]

The default value is QUIT SUCCESS COMMIT.

#### REMARK

Use REMARK to include comments in a script.

REM[ARK] [optional\_text]

You may also use the following convention to include a comment:

/\*

\* This is an example of a three line comment.

\*/

#### SAVE

Use the SAVE command to write the SQL Buffer to an operating system file.

SAV[E] file\_name  
 [CRE[ATE] | REP[LACE] | APP[END]]

file\_name

file\_name specifies the name of the file (including the path) where the buffer contents are written. If you do not provide a file extension, .sql is appended to the end of the file name.

CREATE

Include the CREATE keyword to create a new file. A new file is created only if a file with the specified name does not already exist. This is the default.

REPLACE

Include the REPLACE keyword to specify that Advanced Server should overwrite an existing file.

APPEND

Include the APPEND keyword to specify that Advanced Server should append the contents of the SQL buffer to the end of the specified file.

The following example saves the contents of the SQL buffer to a file named example.sql, located in the temp directory:

SQL> SAVE C:\example.sql CREATE

File "example.sql" written.

#### SET

Use the SET command to specify a value for a session level variable that controls EDB\*Plus behavior. The following forms of the SET command are valid:

SET AUTOCOMMIT

Use the SET AUTOCOMMIT command to specify COMMIT behavior for Advanced Server transactions.

SET AUTO[COMMIT]  
 {ON | OFF | IMMEDIATE | statement\_count}

Please note that EDB\*Plus always automatically commits DDL statements.

ON

Specify ON to turn AUTOCOMMIT behavior on.

OFF

Specify OFF to turn AUTOCOMMIT behavior off.

IMMEDIATE

IMMEDIATE has the same effect as ON.

statement\_count

Include a value for statement\_count to instruct EDB\*Plus to issue a commit after the specified count of successful SQL statements.

SET COLUMN SEPARATOR

Use the SET COLUMN SEPARATOR command to specify the text that Advanced Server displays between columns.

SET COLSEP column\_separator

The default value of column\_separator is a single space.

SET ECHO

Use the SET ECHO command to specify if SQL and EDB\*Plus script statements should be displayed onscreen as they are executed.

SET ECHO {ON | OFF}

The default value is OFF.

SET FEEDBACK

The SET FEEDBACK command controls the display of interactive information after a SQL statement executes.

SET FEED[BACK] {ON | OFF | row\_threshold}

row\_threshold

Specify an integer value for row\_threshold. Setting row\_threshold to 0 is same as setting FEEDBACK to OFF. Setting row\_threshold equal 1 effectively sets FEEDBACK to ON.

SET FLUSH

Use the SET FLUSH command to control display buffering.

SET FLU[SH] {ON | OFF}

Set FLUSH to OFF to enable display buffering. If you enable buffering, messages bound for the screen may not appear until the script completes. Please note that setting FLUSH to OFF will offer better performance.

Set FLUSH to ON to disable display buffering. If you disable buffering, messages bound for the screen appear immediately.

SET HEADING

Use the SET HEADING variable to specify if Advanced Server should display column headings for SELECT statements.

SET HEA[DING] {ON | OFF}

SET HEAD SEPARATOR

The SET HEADSEP command sets the new heading separator character used by the COLUMN HEADING command. The default is '|'.

SET HEADS[EP]

SET LINESIZE

Use the SET LINESIZE command to specify the width of a line in characters.

SET LIN[ESIZE] width\_of\_line

width\_of\_line

The default value of width\_of\_line is 132.

SET NEWPAGE

Use the SET NEWPAGE command to specify how many blank lines are printed after a page break.

SET NEWP[AGE] lines\_per\_page

lines\_per\_page

The default value of lines\_per\_page is 1.

SET NULL

Use the SET NULL command to specify a string that is displayed to the user when a NULL column value is displayed in the output buffer.

SET NULL null\_string

SET PAGESIZE

Use the SET PAGESIZE command to specify the number of printed lines that fit on a page.

SET PAGES[IZE] line\_count

Use the line\_count parameter to specify the number of lines per page.

SET SQLCASE

The SET SQLCASE command specifies if SQL statements transmitted to the server should be converted to upper or lower case.

SET SQLC[ASE] {MIX[ED] | UP[PER] | LO[WER]}

UPPER

Specify UPPER to convert the command text to uppercase.

LOWER

Specify LOWER to convert the command text to lowercase.

MIXED

Specify MIXED to leave the case of SQL commands unchanged. The default is MIXED.

SET PAUSE

The SET PAUSE command is most useful when included in a script; the command displays a prompt and waits for the user to press Return.

SET PAU[SE] {ON | OFF}

If SET PAUSE is ON, the message Hit ENTER to continue… will be displayed before each command is executed.

SET SPACE

Use the SET SPACE command to specify the number of spaces to display between columns:

SET SPACE number\_of\_spaces

SET SQLPROMPT

Use SET SQLPROMPT to set a value for a user-interactive prompt:

SET SQLP[ROMPT] "prompt"

By default, SQLPROMPT is set to "SQL> "

SQL TERMOUT

Use the SQL TERMOUT command to specify if command output should be displayed onscreen.

SET TERM[OUT] {ON | OFF}

SQL TIMING

The SQL TIMING command specifies if Advanced Server should display the execution time for each SQL statement after it is executed.

SET TIMI[NG] {ON | OFF}

SET VERIFY

Specifies if both the old and new values of a SQL statement are displayed when a substitution variable is encountered.

SET VER[IFY] { ON | OFF }

#### SHOW

Use the SHOW command to display current parameter values.

SHO[W] {ALL | parameter\_name}

Display the current parameter settings by including the ALL keyword:

SQL> SHOW ALL

autocommit OFF

colsep " "

define "&"

echo OFF

FEEDBACK ON for 6 row(s).

flush ON

heading ON

headsep "|"

linesize 78

newpage 1

null " "

pagesize 14

pause OFF

serveroutput OFF

spool OFF

sqlcase MIXED

sqlprompt "SQL> "

sqlterminator ";"

suffix ".sql"

termout ON

timing OFF

verify ON

USER is "enterprisedb"

HOST is "localhost"

PORT is "5444"

DATABASE is "edb"

VERSION is "9.5.0.0"

Or display a specific parameter setting by including the parameter\_name in the SHOW command:

SQL> SHOW VERSION

VERSION is "9.5.0.0"

#### SPOOL

The SPOOL command sends output from the display to a file.

SP[OOL] output\_file | OFF

Use the output\_file parameter to specify a path name for the output file.

#### START

Use the START command to run an EDB\*Plus script file; START is an alias for @ command.

STA[RT] script\_file

Specify the name of a script file in the script\_file parameter.

#### UNDEFINE

The UNDEFINE command erases a user variable created by the DEFINE command.

UNDEF[INE] variable\_name [ variable\_name...]

Use the variable\_name parameter to specify the name of a variable or variables.

#### WHENEVER SQLERROR

The WHENEVER SQLERROR command provides error handling for SQL errors or PL/SQL block errors. The syntax is:

WHENEVER SQLERROR   
 {CONTINUE [COMMIT|ROLLBACK|NONE]  
 |EXIT [SUCCESS|FAILURE|WARNING|n|sub\_variable]  
 [COMMIT|ROLLBACK]}

If Advanced Server encounters an error during the execution of a SQL command or PL/SQL block, EDB\*Plus performs the action specified in the WHENEVER SQLERROR command:

Include the CONTINUE clause to instruct EDB\*Plus to perform the specified action before continuing.

Include the COMMIT clause to instruct EDB\*Plus to COMMIT the current transaction before exiting or continuing.

Include the ROLLBACK clause to instruct EDB\*Plus to ROLLBACK the current transaction before exiting or continuing.

Include the NONE clause to instruct EDB\*Plus to continue without committing or rolling back the transaction.

Include the EXIT clause to instruct EDB\*Plus to perform the specified action and exit if it encounters an error.

Use the following options to specify a status code that EDB\*Plus will return before exiting:

[SUCCESS|FAILURE|WARNING|n|sub\_variable]

Please note that EDB\*Plus supports substitution variables, but does not support bind variables.

## libpq C Library

libpq is the C application programmer’s interface to Advanced Server. libpq is a set of library functions that allow client programs to pass queries to the Advanced Server and to receive the results of these queries.

libpq is also the underlying engine for several other EnterpriseDB application interfaces including those written for C++, Perl, Python, Tcl and ECPG. So some aspects of libpq’s behavior will be important to the user if one of those packages is used.

Client programs that use libpq must include the header file libpq-fe.h and must link with the libpq library.

### Using libpq with EnterpriseDB SPL

The EnterpriseDB SPL language can be used with the libpq interface library, providing support for:

* Procedures, functions, packages
* Prepared statements
* REFCURSORs
* Static cursors
* structs and typedefs
* Arrays
* DML and DDL operations
* IN/OUT/IN OUT parameters

### REFCURSOR Support

In earlier releases, Advanced Server provided support for REFCURSORs through the following libpq functions; these functions should now be considered deprecated:

* PQCursorResult()
* PQgetCursorResult()
* PQnCursor()

You may now use PQexec() and PQgetvalue() to retrieve a REFCURSOR returned by an SPL (or PL/pgSQL) function. A REFCURSOR is returned in the form of a null-terminated string indicating the name of the cursor. Once you have the name of the cursor, you can execute one or more FETCH statements to retrieve the values exposed through the cursor.

Please note that the samples that follow do not include error-handling code that would be required in a real-world client application.

**Returning a Single REFCURSOR**

The following example shows an SPL function that returns a value of type REFCURSOR:

CREATE OR REPLACE FUNCTION getEmployees(p\_deptno NUMERIC) RETURN REFCURSOR AS  
 result REFCURSOR;  
BEGIN  
 OPEN result FOR SELECT \* FROM emp WHERE deptno = p\_deptno;  
  
 RETURN result;  
END;

This function expects a single parameter, p\_deptno, and returns a REFCURSOR that holds the result set for the SELECT query shown in the OPEN statement. The OPEN statement executes the query and stores the result set in a cursor. The server constructs a name for that cursor and stores the name in a variable (named result). The function then returns the name of the cursor to the caller.

To call this function from a C client using libpq, you can use PQexec() and PQgetvalue():

#include <stdio.h>  
#include <stdlib.h>  
#include "libpq-fe.h"  
  
static void fetchAllRows(PGconn \*conn,   
 const char \*cursorName,   
 const char \*description);  
static void fail(PGconn \*conn, const char \*msg);  
  
int   
main(int argc, char \*argv[])  
{  
 PGconn \*conn = PQconnectdb(argv[1]);  
 PGresult \*result;  
  
 if (PQstatus(conn) != CONNECTION\_OK)  
 fail(conn, PQerrorMessage(conn));  
  
 result = PQexec(conn, "BEGIN TRANSACTION");  
  
 if (PQresultStatus(result) != PGRES\_COMMAND\_OK)  
 fail(conn, PQerrorMessage(conn));  
  
 PQclear(result);  
  
 result = PQexec(conn, "SELECT \* FROM getEmployees(10)");  
  
 if (PQresultStatus(result) != PGRES\_TUPLES\_OK)  
 fail(conn, PQerrorMessage(conn));  
  
 fetchAllRows(conn, PQgetvalue(result, 0, 0), "employees");  
  
 PQclear(result);  
  
 PQexec(conn, "COMMIT");  
  
 PQfinish(conn);  
  
 exit(0);  
}  
  
static void   
fetchAllRows(PGconn \*conn,   
 const char \*cursorName,   
 const char \*description)  
{  
 size\_t commandLength = strlen("FETCH ALL FROM ") +  
 strlen(cursorName) + 3;

char \*commandText = malloc(commandLength);  
 PGresult \*result;  
 int row;  
  
 sprintf(commandText, "FETCH ALL FROM \"%s\"", cursorName);  
  
 result = PQexec(conn, commandText);  
  
 if (PQresultStatus(result) != PGRES\_TUPLES\_OK)  
 fail(conn, PQerrorMessage(conn));  
  
 printf("-- %s --\n", description);  
  
 for (row = 0; row < PQntuples(result); row++)  
 {  
 const char \*delimiter = "\t";  
 int col;  
  
 for (col = 0; col < PQnfields(result); col++)  
 {  
 printf("%s%s", delimiter, PQgetvalue(result, row, col));  
 delimiter = ",";  
 }  
  
 printf("\n");  
 }  
  
 PQclear(result);  
 free(commandText);  
}  
  
static void   
fail(PGconn \*conn, const char \*msg)  
{  
 fprintf(stderr, "%s\n", msg);  
  
 if (conn != NULL)  
 PQfinish(conn);  
  
 exit(-1);  
}

The code sample contains a line of code that calls the getEmployees() function, and returns a result set that contains all of the employees in department 10:

result = PQexec(conn, "SELECT \* FROM getEmployees(10)");

The PQexec() function returns a result set handle to the C program. The result set will contain exactly one value; that value is the name of the cursor as returned by getEmployees().

Once you have the name of the cursor, you can use the SQL FETCH statement to retrieve the rows in that cursor. The function fetchAllRows() builds a FETCH ALL statement, executes that statement, and then prints the result set of the FETCH ALL statement.

The output of this program is shown below:

-- employees --  
 7782,CLARK,MANAGER,7839,09-JUN-81 00:00:00,2450.00,,10  
 7839,KING,PRESIDENT,,17-NOV-81 00:00:00,5000.00,,10  
 7934,MILLER,CLERK,7782,23-JAN-82 00:00:00,1300.00,,10

**Returning Multiple REFCURSORs**

The next example returns two REFCURSORs:

* The first REFCURSOR contains the name of a cursor (employees) that contains all employees who work in a department within the range specified by the caller.
* The second REFCURSOR contains the name of a cursor (departments) that contains all of the departments in the range specified by the caller.

In this example, instead of returning a single REFCURSOR, the function returns a SETOF REFCURSOR (which means 0 or more REFCURSORS). One other important difference is that the libpq program should not expect a single REFCURSOR in the result set, but should expect two rows, each of which will contain a single value (the first row contains the name of the employees cursor, and the second row contains the name of the departments cursor).

CREATE OR REPLACE FUNCTION getEmpsAndDepts(p\_min NUMERIC,  
 p\_max NUMERIC)  
RETURN SETOF REFCURSOR AS  
 employees REFCURSOR;  
 departments REFCURSOR;  
BEGIN  
 OPEN employees FOR   
 SELECT \* FROM emp WHERE deptno BETWEEN p\_min AND p\_max;  
 RETURN NEXT employees;  
  
 OPEN departments FOR   
 SELECT \* FROM dept WHERE deptno BETWEEN p\_min AND p\_max;  
 RETURN NEXT departments;  
END;

As in the previous example, you can use PQexec() and PQgetvalue() to call the SPL function:

#include <stdio.h>  
#include <stdlib.h>  
#include "libpq-fe.h"  
  
static void fetchAllRows(PGconn \*conn,   
 const char \*cursorName,   
 const char \*description);  
static void fail(PGconn \*conn, const char \*msg);  
  
int   
main(int argc, char \*argv[])  
{  
 PGconn \*conn = PQconnectdb(argv[1]);  
 PGresult \*result;  
  
 if (PQstatus(conn) != CONNECTION\_OK)  
 fail(conn, PQerrorMessage(conn));  
  
 result = PQexec(conn, "BEGIN TRANSACTION");  
  
 if (PQresultStatus(result) != PGRES\_COMMAND\_OK)  
 fail(conn, PQerrorMessage(conn));  
  
 PQclear(result);  
  
 result = PQexec(conn, "SELECT \* FROM getEmpsAndDepts(20, 30)");  
  
 if (PQresultStatus(result) != PGRES\_TUPLES\_OK)  
 fail(conn, PQerrorMessage(conn));  
  
 fetchAllRows(conn, PQgetvalue(result, 0, 0), "employees");  
 fetchAllRows(conn, PQgetvalue(result, 1, 0), "departments");  
  
 PQclear(result);  
  
 PQexec(conn, "COMMIT");  
  
 PQfinish(conn);  
  
 exit(0);  
}  
  
static void   
fetchAllRows(PGconn \*conn,   
 const char \*cursorName,   
 const char \*description)  
{  
 size\_t commandLength = strlen("FETCH ALL FROM ") +  
 strlen(cursorName) + 3;  
 char \*commandText = malloc(commandLength);  
 PGresult \*result;  
 int row;  
  
 sprintf(commandText, "FETCH ALL FROM \"%s\"", cursorName);  
  
 result = PQexec(conn, commandText);  
  
 if (PQresultStatus(result) != PGRES\_TUPLES\_OK)  
 fail(conn, PQerrorMessage(conn));  
  
 printf("-- %s --\n", description);  
  
 for (row = 0; row < PQntuples(result); row++)  
 {  
 const char \*delimiter = "\t";  
 int col;  
  
 for (col = 0; col < PQnfields(result); col++)  
 {  
 printf("%s%s", delimiter, PQgetvalue(result, row, col));  
 delimiter = ",";  
 }  
  
 printf("\n");  
 }  
  
 PQclear(result);  
 free(commandText);  
}  
  
static void   
fail(PGconn \*conn, const char \*msg)  
{  
 fprintf(stderr, "%s\n", msg);  
  
 if (conn != NULL)  
 PQfinish(conn);  
  
 exit(-1);  
}

If you call getEmpsAndDepts(20, 30), the server will return a cursor that contains all employees who work in department 20 or 30, and a second cursor containing the description of departments 20 and 30.

-- employees --  
 7369,SMITH,CLERK,7902,17-DEC-80 00:00:00,800.00,,20  
 7499,ALLEN,SALESMAN,7698,20-FEB-81 00:00:00,1600.00,300.00,30  
 7521,WARD,SALESMAN,7698,22-FEB-81 00:00:00,1250.00,500.00,30  
 7566,JONES,MANAGER,7839,02-APR-81 00:00:00,2975.00,,20  
 7654,MARTIN,SALESMAN,7698,28-SEP-81 00:00:00,1250.00,1400.00,30  
 7698,BLAKE,MANAGER,7839,01-MAY-81 00:00:00,2850.00,,30  
 7788,SCOTT,ANALYST,7566,19-APR-87 00:00:00,3000.00,,20  
 7844,TURNER,SALESMAN,7698,08-SEP-81 00:00:00,1500.00,0.00,30  
 7876,ADAMS,CLERK,7788,23-MAY-87 00:00:00,1100.00,,20  
 7900,JAMES,CLERK,7698,03-DEC-81 00:00:00,950.00,,30  
 7902,FORD,ANALYST,7566,03-DEC-81 00:00:00,3000.00,,20  
-- departments --  
 20,RESEARCH,DALLAS  
 30,SALES,CHICAGO

### Array Binding

Advanced Server's array binding functionality allows you to send an array of data across the network in a single round-trip. When the back end receives the bulk data, it can use the data to perform insert or update operations.

Perform bulk operations with a prepared statement; use the following function to prepare the statement:

PGresult \*PQprepare(PGconn \*conn,

const char \*stmtName,

const char \*query,

int nParams,

const Oid \*paramTypes);

Details of PQprepare() can be found in the prepared statement section.

The following functions can be used to perform bulk operations:

* PQBulkStart
* PQexecBulk
* PQBulkFinish
* PQexecBulkPrepared

#### PQBulkStart

PQBulkStart() initializes bulk operations on the server. You must call this function before sending bulk data to the server. PQBulkStart() initializes the prepared statement specified in stmtName to receive data in a format specified by paramFmts.

API Definition

PGresult \* PQBulkStart(PGconn \*conn,

const char \* Stmt\_Name,

unsigned int nCol,

const int \*paramFmts);

#### PQexecBulk

PQexecBulk() is used to supply data (paramValues) for a statement that was previously initialized for bulk operation using PQBulkStart().

This function can be used more than once after PQBulkStart() to send multiple blocks of data. See the example for more details.

API Definition

PGresult \*PQexecBulk(PGconn \*conn,

unsigned int nRows,

const char \*const \* paramValues,

const int \*paramLengths);

#### PQBulkFinish

This function completes the current bulk operation. You can use the prepared statement again without re-preparing it.

API Definition

PGresult \*PQBulkFinish(PGconn \*conn);

#### PQexecBulkPrepared

Alternatively, you can use the PQexecBulkPrepared() function to perform a bulk operation with a single function call. PQexecBulkPrepared() sends a request to execute a prepared statement with the given parameters, and waits for the result. This function combines the functionality of PQbulkStart(), PQexecBulk(), and PQBulkFinish(). When using this function, you are not required to initialize or terminate the bulk operation; this function starts the bulk operation, passes the data to the server, and terminates the bulk operation.

Specify a previously prepared statement in the place of stmtName. Commands that will be used repeatedly will be parsed and planned just once, rather than each time they are executed.

API Definition

PGresult \*PQexecBulkPrepared(PGconn \*conn,

const char \*stmtName,

unsigned int nCols,

unsigned int nRows,

const char \*const \*paramValues,

const int \*paramLengths,

const int \*paramFormats);

#### Example Code (Using PQBulkStart, PQexecBulk, PQBulkFinish)

The following example uses PGBulkStart, PQexecBulk, and PQBulkFinish.

void InsertDataUsingBulkStyle( PGconn \*conn )

{

PGresult \*res;

Oid paramTypes[2];

char \*paramVals[5][2];

int paramLens[5][2];

int paramFmts[2];

int i;

int a[5] = { 10, 20, 30, 40, 50 };

char b[5][10] = { "Test\_1", "Test\_2", "Test\_3", "Test\_4", "Test\_5" };

paramTypes[0] = 23;

paramTypes[1] = 1043;

res = PQprepare( conn, "stmt\_1", "INSERT INTO testtable1 values( $1, $2 )", 2, paramTypes );

PQclear( res );

paramFmts[0] = 1; /\* Binary format \*/

paramFmts[1] = 0;

for( i = 0; i < 5; i++ )

{

a[i] = htonl( a[i] );

paramVals[i][0] = &(a[i]);

paramVals[i][1] = b[i];

paramLens[i][0] = 4;

paramLens[i][1] = strlen( b[i] );

}

res = PQBulkStart(conn, "stmt\_1", 2, paramFmts);

PQclear( res );

printf( "< -- PQBulkStart -- >\n" );

res = PQexecBulk(conn, 5, (const char \*const \*)paramVals, (const int \*)paramLens);

PQclear( res );

printf( "< -- PQexecBulk -- >\n" );

res = PQexecBulk(conn, 5, (const char \*const \*)paramVals, (const int \*)paramLens);

PQclear( res );

printf( "< -- PQexecBulk -- >\n" );

res = PQBulkFinish(conn);

PQclear( res );

printf( "< -- PQBulkFinish -- >\n" );

}

#### Example Code (Using PQexecBulkPrepared)

The following example uses PQexecBulkPrepared.

void InsertDataUsingBulkStyleCombinedVersion( PGconn \*conn )

{

PGresult \*res;

Oid paramTypes[2];

char \*paramVals[5][2];

int paramLens[5][2];

int paramFmts[2];

int i;

int a[5] = { 10, 20, 30, 40, 50 };

char b[5][10] = { "Test\_1", "Test\_2", "Test\_3", "Test\_4", "Test\_5" };

paramTypes[0] = 23;

paramTypes[1] = 1043;

res = PQprepare( conn, "stmt\_2", "INSERT INTO testtable1 values( $1, $2 )", 2, paramTypes );

PQclear( res );

paramFmts[0] = 1; /\* Binary format \*/

paramFmts[1] = 0;

for( i = 0; i < 5; i++ )

{

a[i] = htonl( a[i] );

paramVals[i][0] = &(a[i]);

paramVals[i][1] = b[i];

paramLens[i][0] = 4;

paramLens[i][1] = strlen( b[i] );

}

res = PQexecBulkPrepared(conn, "stmt\_2", 2, 5, (const char \*const \*)paramVals,(const int \*)paramLens, (const int \*)paramFmts);

PQclear( res );

}

## ECPGPlus

EnterpriseDB has enhanced ECPG (the PostgreSQL pre-compiler) to create ECPGPlus. ECPGPlus allows you to include embedded SQL commands in C applications; when you use ECPGPlus to compile an application that contains embedded SQL commands, the SQL code is syntax-checked and translated into C.

ECPGPlus supports Pro\*C compatible syntax in C programs when connected to an Advanced Server database. ECPGPlus supports:

* Oracle Dynamic SQL – Method 4 (ODS-M4)
* Pro\*C compatible anonymous blocks
* A CALL statement compatible with Oracle databases

As part of ECPGPlus's Pro\*C compatibility, you do not need to include the BEGIN DECLARE SECTION and END DECLARE SECTION directives.

For more information about using ECPGPlus, please see the EDB Postgres Advanced Server ECPG Connector Guide, available from the EnterpriseDB website at:

<http://www.enterprisedb.com/products-services-training/products/documentation/enterpriseedition>

### C-preprocessor Directives

The ECPGPlus C-preprocessor enforces two behaviors that are dependent on the mode in which you invoke ECPGPlus:

* PROC mode
* non-PROC mode

Compiling in PROC mode

In PROC mode, ECPGPlus allows you to:

* Declare host variables outside of an EXEC SQL BEGIN/END DECLARE SECTION.
* Use any C variable as a host variable as long as it is of a data type compatible with ECPG.

When you invoke ECPGPlus in PROC mode (by including the -C PROC keywords), the ECPG compiler honors the following C-preprocessor directives:

#include

#if expression

#ifdef symbolName

#ifndef symbolName

#else

#elif expression

#endif

#define symbolName expansion  
#define symbolName([macro arguments]) expansion

#undef symbolName

#defined(symbolName)

Pre-processor directives are used to effect or direct the code that is received by the compiler. For example, using the following code sample:

#if HAVE\_LONG\_LONG == 1

#define BALANCE\_TYPE long long

#else

#define BALANCE\_TYPE double

#endif

...

BALANCE\_TYPE customerBalance;

If you invoke ECPGPlus with the following command-line arguments:

ecpg –C PROC –DHAVE\_LONG\_LONG=1

ECPGPlus will copy the entire fragment (without change) to the output file, but will only send the following tokens to the ECPG parser:

long long customerBalance;

On the other hand, if you invoke ECPGPlus with the following command-line arguments:

ecpg –C PROC –DHAVE\_LONG\_LONG=0

The ECPG parser will receive the following tokens:

double customerBalance;

If your code uses preprocessor directives to filter the code that is sent to the compiler, the complete code is retained in the original code, while the ECPG parser sees only the processed token stream.

Compiling in non-PROC mode

If you do not include the -C PROC command-line option:

* C preprocessor directives are copied to the output file without change.
* You must declare the type and name of each C variable that you intend to use as a host variable within an EXEC SQL BEGIN/END DECLARE section.

When invoked in non-PROC mode, ECPG implements the behavior described in the PostgreSQL Core documentation, available at:

<http://www.enterprisedb.com/products-services-training/products/documentation/enterpriseedition>

### Supported C Data Types

An ECPGPlus application must deal with two sets of data types: SQL data types (such as SMALLINT, DOUBLE PRECISION and CHARACTER VARYING) and C data types (like short, double and varchar[n]). When an application fetches data from the server, ECPGPlus will map each SQL data type to the type of the C variable into which the data is returned.

In general, ECPGPlus can convert most SQL server types into similar C types, but not all combinations are valid. For example, ECPGPlus will try to convert a SQL character value into a C integer value, but the conversion may fail (at execution time) if the SQL character value contains non-numeric characters.

The reverse is also true; when an application sends a value to the server, ECPGPlus will try to convert the C data type into the required SQL type. Again, the conversion may fail (at execution time) if the C value cannot be converted into the required SQL type.

ECPGPlus can convert any SQL type into C character values (char[n] or varchar[n]). Although it is safe to convert any SQL type to/from char[n] or varchar[n], it is often convenient to use more natural C types such as int, double, or float.

The supported C data types are:

* short
* int
* unsigned int
* long long int
* float
* double
* char[n+1]
* varchar[n+1]
* bool
* and any equivalent created by a typedef

In addition to the numeric and character types supported by C, the pgtypeslib run-time library offers custom data types (and functions to operate on those types) for dealing with date/time and exact numeric values:

* timestamp
* interval
* date
* decimal
* numeric

To use a data type supplied by pgtypeslib, you must #include the proper header file.

### Type Codes

The following table contains the type codes for externaldata types. An external data type is used to indicate the type of a C host variable. When an application binds a value to a parameter or binds a buffer to a SELECT-list item, the type code in the corresponding SQLDA descriptor (descriptor->T[column]) should be set to one of the following values:

|  |  |
| --- | --- |
| Type Code | Host Variable Type (C Data Type) |
| 1, 2, 8, 11, 12, 15, 23, 24, 91, 94, 95, 96, 97 | char[] |
| 3 | int |
| 4, 7, 21 | float |
| 5, 6 | null-terminated string (char[length+1]) |
| 9 | varchar |
| 22 | double |
| 68 | unsigned int |

The following table contains the type codes for internaldata types. An internal type code is used to indicate the type of a value as it resides in the database. The DESCRIBE SELECT LIST statement populates the data type array (descriptor->T[column]) using the following values.

|  |  |
| --- | --- |
| Internal Type Code | Server Type |
| 1 | VARCHAR2 |
| 2 | NUMBER |
| 8 | LONG |
| 11 | ROWID |
| 12 | DATE |
| 23 | RAW |
| 24 | LONG RAW |
| 96 | CHAR |
| 100 | BINARY FLOAT |
| 101 | BINARY DOUBLE |
| 104 | UROWID |
| 187 | TIMESTAMP |
| 188 | TIMESTAMP W/TIMEZONE |
| 189 | INTERVAL YEAR TO MONTH |
| 190 | INTERVAL DAY TO SECOND |
| 232 | TIMESTAMP LOCAL\_TZ |

### The SQLDA Structure

Oracle Dynamic SQL method 4 uses the SQLDA data structure to hold the data and metadata for a dynamic SQL statement. A SQLDA structure can describe a set of input parameters corresponding to the parameter markers found in the text of a dynamic statement or the result set of a dynamic statement. The layout of the SQLDA structure is:

struct SQLDA

{

int N; /\* Number of entries \*/

char \*\*V; /\* Variables \*/

int \*L; /\* Variable lengths \*/

short \*T; /\* Variable types \*/

short \*\*I; /\* Indicators \*/

int F; /\* Count of variables discovered by DESCRIBE \*/

char \*\*S; /\* Variable names \*/

short \*M; /\* Variable name maximum lengths \*/

short \*C; /\* Variable name actual lengths \*/

char \*\*X; /\* Indicator names \*/

short \*Y; /\* Indicator name maximum lengths \*/

short \*Z; /\* Indicator name actual lengths \*/

};

**Parameters**

N - maximum number of entries

The N structure member contains the maximum number of entries that the SQLDA may describe. This member is populated by the sqlald() function when you allocate the SQLDA structure. Before using a descriptor in an OPEN or FETCH statement, you must set N to the actualnumber of values described.

V - data values

The V structure member is a pointer to an array of data values.

For a SELECT-list descriptor, V points to an array of values returned by a FETCH statement (each member in the array corresponds to a column in the result set).

For a bind descriptor, V points to an array of parameter values (you must populate the values in this array before opening a cursor that uses the descriptor).

Your application must allocate the space required to hold each value.

L - length of each data value

The L structure member is a pointer to an array of lengths. Each member of this array must indicate the amount of memory available in the corresponding member of the V array. For example, if V[5] points to a buffer large enough to hold a 20-byte NULL-terminated string, L[5] should contain the value 21 (20 bytes for the characters in the string plus 1 byte for the NULL-terminator). Your application must set each member of the L array.

T - data types

The T structure member points to an array of data types, one for each column (or parameter) described by the descriptor.

For a bind descriptor, you must set each member of the T array to tell ECPGPlus the data type of each parameter.

For a SELECT-list descriptor, the DESCRIBE SELECT LIST statement sets each member of the T array to reflect the type of data found in the corresponding column.

You may change any member of the T array before executing a FETCH statement to force ECPGPlus to convert the corresponding value to a specific data type. For example, if the DESCRIBE SELECT LIST statement indicates that a given column is of type DATE, you may change the corresponding T member to request that the next FETCH statement return that value in the form of a NULL-terminated string. Each member of the T array is a numeric type code. The type codes returned by a DESCRIBE SELECT LIST statement differ from those expected by a FETCH statement. After executing a DESCRIBE SELECT LIST statement, each member of T encodes a data type anda flag indicating whether the corresponding column is nullable. You can use the sqlnul() function to extract the type code and nullable flag from a member of the T array. The signature of the sqlnul() function is as follows:

void sqlnul(unsigned short \*valType,   
 unsigned short \*typeCode,   
 int \*isNull)

For example, to find the type code and nullable flag for the third column of a descriptor named results, you would invoke sqlnul() as follows:

sqlnul(&results->T[2], &typeCode, &isNull);

I - indicator variables

The I structure member points to an array of indicator variables. This array is allocated for you when your application calls the sqlald() function to allocate the descriptor.

For a SELECT-list descriptor, each member of the I array indicates whether the corresponding column contains a NULL (non-zero) or non-NULL (zero) value.

For a bind parameter, your application must set each member of the I array to indicate whether the corresponding parameter value is NULL.

F - number of entries

The F structure member indicates how many values are described by the descriptor (the N structure member indicates the maximumnumber of values which may be described by the descriptor; F indicates the actual number of values). The value of the F member is set by ECPGPlus when you execute a DESCRIBE statement. F may be positive, negative, or zero.

For a SELECT-list descriptor, F will contain a positive value if the number of columns in the result set is equal to or less than the maximum number of values permitted by the descriptor (as determined by the N structure member); 0 if the statement is not a SELECT statement, or a negative value if the query returns more columns than allowed by the N structure member.

For a bind descriptor, F will contain a positive number if the number of parameters found in the statement is less than or equal to the maximum number of values permitted by the descriptor (as determined by the N structure member); 0 if the statement contains no parameters markers, or a negative value if the statement contains more parameter markers than allowed by the N structure member.

If F contains a positive number (after executing a DESCRIBE statement), that number reflects the count of columns in the result set (for a SELECT-list descriptor) or the number of parameter markers found in the statement (for a bind descriptor). If F contains a negative value, you may compute the absolute value of F to discover how many values (or parameter markers) are required. For example, if F contains -24 after describing a SELECT list, you know that the query returns 24 columns.

S - column/parameter names

The S structure member points to an array of NULL-terminated strings.

For a SELECT-list descriptor, the DESCRIBE SELECT LIST statement sets each member of this array to the name of the corresponding column in the result set.

For a bind descriptor, the DESCRIBE BIND VARIABLES statement sets each member of this array to the name of the corresponding bind variable.

In this release, the name of each bind variable is determined by the left-to-right order of the parameter marker within the query - for example, the name of the first parameter is always ?0, the name of the second parameter is always ?1, and so on.

M - maximum column/parameter name length

The M structure member points to an array of lengths. Each member in this array specifies the *maximum* length of the corresponding member of the S array (that is, M[0] specifies the maximum length of the column/parameter name found at S[0]). This array is populated by the sqlald() function.

C - actual column/parameter name length

The C structure member points to an array of lengths. Each member in this array specifies the actual length of the corresponding member of the S array (that is, C[0] specifies the actual length of the column/parameter name found at S[0]).

This array is populated by the DESCRIBE statement.

X - indicator variable names

The X structure member points to an array of NULL-terminated strings - each string represents the name of a NULL indicator for the corresponding value.

This array is not used by ECPGPlus, but is provided for compatibility with Pro\*C applications.

Y - maximum indicator name length

The Y structure member points to an array of lengths. Each member in this array specifies the maximum length of the corresponding member of the X array (that is, Y[0] specifies the maximum length of the indicator name found at X[0]).

This array is not used by ECPGPlus, but is provided for compatibility with Pro\*C applications.

Z - actual indicator name length

The Z structure member points to an array of lengths. Each member in this array specifies the actual length of the corresponding member of the X array (that is, Z[0] specifies the actual length of the indicator name found at X[0]).

This array is not used by ECPGPlus, but is provided for compatibility with Pro\*C applications.

### ECPGPlus Statements

An embedded SQL statement allows your client application to interact with the server, while an embedded directive is an instruction to the ECPGPlus compiler.

You can embed any Advanced Server SQL statement in a C program. Each statement should begin with the keywords EXEC SQL, and must be terminated with a semi-colon (;). Within the C program, a SQL statement takes the form:

EXEC SQL sql\_command\_body;

Where sql\_command\_body represents a standard SQL statement. You can use a host variable anywhere that the SQL statement expects a value expression.

ECPGPlus extends the PostgreSQL server-side syntax for some statements; for those statements, syntax differences are outlined in the following reference sections. For a complete reference to the supported syntax of other SQL commands, please refer to the PostgreSQL Core Documentation available at:

<http://www.postgresql.org/docs/9.5/static/sql-commands.html>

#### ALLOCATE DESCRIPTOR

Use the ALLOCATE DESCRIPTOR statement to allocate an SQL descriptor area:

EXEC SQL [FOR array\_size] ALLOCATE DESCRIPTOR descriptor\_name [WITH MAX variable\_count];

Where:

array\_size is a variable that specifies the number of array elements to allocate for the descriptor. array\_size may be an INTEGER value or a host variable.

descriptor\_name is the host variable that contains the name of the descriptor, or the name of the descriptor. This value may take the form of an identifier, a quoted string literal, or of a host variable.

variable\_count specifies the maximum number of host variables in the descriptor. The default value of variable\_count is 100.

The following code fragment allocates a descriptor named emp\_query that may be processed as an array (emp\_array):

EXEC SQL FOR :emp\_array ALLOCATE DESCRIPTOR emp\_query;

#### CALL

Use the CALL statement to invoke a procedure or function on the server. The CALL statement works only on Advanced Server. The CALL statement comes in two forms; the first form is used to call a function:

EXEC SQL CALL program\_name'('[actual\_arguments]')'   
 INTO [[:ret\_variable][:ret\_indicator]];

The second form is used to call a procedure:

EXEC SQL CALL program\_name'('[actual\_arguments]')';

Where:

program\_name is the name of the stored procedure or function that the CALL statement invokes. The program name may be schema-qualified or package-qualified (or both); if you do not specify the schema or package in which the program resides, ECPGPlus will use the value of search\_path to locate the program.

actual\_arguments specifies a comma-separated list of arguments required by the program. Note that each actual\_argument corresponds to a formal argument expected by the program. Each formal argument may be an IN parameter, an OUT parameter, or an INOUT parameter.

:ret\_variablespecifies a host variable that will receive the value returned if the program is a function.

:ret\_indicator specifies a host variable that will receive the indicator value returned, if the program is a function.

For example, the following statement invokes the get\_job\_desc function with the value contained in the :ename host variable, and captures the value returned by that function in the :job host variable:

EXEC SQL CALL get\_job\_desc(:ename)   
 INTO :job;

#### CLOSE

Use the CLOSE statement to close a cursor, and free any resources currently in use by the cursor. A client application cannot fetch rows from a closed cursor. The syntax of the CLOSE statement is:

EXEC SQL CLOSE [cursor\_name];

Where:

cursor\_name is the name of the cursor closed by the statement. The cursor name may take the form of an identifier or of a host variable.

The OPEN statement initializes a cursor. Once initialized, a cursor result set will remain unchanged unless the cursor is re-opened. You do not need to CLOSE a cursor before re-opening it.

To manually close a cursor named emp\_cursor, use the command:

EXEC SQL CLOSE emp\_cursor;

A cursor is automatically closed when an application terminates.

#### COMMIT

Use the COMMIT statement to complete the current transaction, making all changes permanent and visible to other users. The syntax is:

EXEC SQL [AT database\_name] COMMIT [WORK]   
 [COMMENT *'*text*'*] [COMMENT *'*text*'* RELEASE];

Where:

database\_name is the name of the database (or host variable that contains the name of the database) in which the work resides. This value may take the form of an unquoted string literal, or of a host variable.

For compatibility, ECPGPlus accepts the COMMENT clause without error but does *not* store any text included with the COMMENT clause.

Include the RELEASE clause to close the current connection after performing the commit.

For example, the following command commits all work performed on the dept database and closes the current connection:

EXEC SQL AT dept COMMIT RELEASE;

By default, statements are committed only when a client application performs a COMMIT statement. Include the -t option when invoking ECPGPlus to specify that a client application should invoke AUTOCOMMIT functionality. You can also control AUTOCOMMIT functionality in a client application with the following statements:

EXEC SQL SET AUTOCOMMIT TO ON

and

EXEC SQL SET AUTOCOMMIT TO OFF

#### CONNECT

Use the CONNECT statement to establish a connection to a database. The CONNECT statement is available in two forms.

The following is the first form:

EXEC SQL CONNECT   
 {{:user\_name IDENTIFIED BY :password} | :connection\_id}  
 [AT database\_name]  
 [USING :database\_string]  
 [ALTER AUTHORIZATION :new\_password];

Where:

user\_name is a host variable that contains the role that the client application will use to connect to the server.

password is a host variable that contains the password associated with that role.

connection\_id is a host variable that contains a slash-delimited user name and password used to connect to the database.

Include the AT clause to specify the database to which the connection is established. database\_name is the name of the database to which the client is connecting; specify the value in the form of a variable, or as a string literal.

Include the USING clause to specify a host variable that contains a null-terminated string identifying the database to which the connection will be established.

The ALTER AUTHORIZATION clause is supported for syntax compatibility only; ECPGPlus parses the ALTER AUTHORIZATION clause, and reports a warning.

Using the first form of the CONNECT statement, a client application might establish a connection with a host variable named user that contains the identity of the connecting role, and a host variable named password that contains the associated password using the following command:

EXEC SQL CONNECT :user IDENTIFIED BY :password;

A client application could also use the first form of the CONNECT statement to establish a connection using a single host variable named :connection\_id. In the following example, connection\_id contains the slash-delimited role name and associated password for the user:

EXEC SQL CONNECT :connection\_id;

The syntax of the second form of the CONNECT statement is:

EXEC SQL CONNECT TO database\_name   
[AS connection\_name] [credentials];

Where credentials is one of the following:

USER user\_name passwordUSER user\_name IDENTIFIED BY password  
USER user\_name USING password

In the second form:

database\_name is the name or identity of the database to which the client is connecting. Specify database\_name as a variable, or as a string literal, in one of the following forms:

database\_name[@hostname][:port]

tcp:postgresql://hostname[:port][/database\_name][options]

unix:postgresql://hostname[:port][/database\_name][options]

Where:

hostname is the name or IP address of the server on which the database resides.

port is the port on which the server listens.

You can also specify a value of DEFAULT to establish a connection with the default database, using the default role name. If you specify DEFAULT as the target database, do not include a connection\_name or credentials.

connection\_name is the name of the connection to the database. connection\_name should take the form of an identifier (that is, not a string literal or a variable). You can open multiple connections, by providing a unique connection\_name for each connection.

If you do not specify a name for a connection, ecpglib assigns a name of DEFAULT to the connection. You can refer to the connection by name (DEFAULT) in any EXEC SQL statement.

CURRENT is the most recently opened or the connection mentioned in the most-recent SET CONNECTION TO statement. If you do not refer to a connection by name in an EXEC SQL statement, ECPG assumes the name of the connection to be CURRENT.

user\_name is the role used to establish the connection with the Advanced Server database. The privileges of the specified role will be applied to all commands performed through the connection.

password is the password associated with the specified user\_name.

The following code fragment uses the second form of the CONNECT statement to establish a connection to a database named edb, using the role alice and the password associated with that role, 1safepwd:

EXEC SQL CONNECT TO edb AS acctg\_conn  
 USER 'alice' IDENTIFIED BY '1safepwd';

The name of the connection is acctg\_conn; you can use the connection name when changing the connection name using the SET CONNECTION statement.

#### DEALLOCATE DESCRIPTOR

Use the DEALLOCATE DESCRIPTOR statement to free memory in use by an allocated descriptor. The syntax of the statement is:

EXEC SQL DEALLOCATE DESCRIPTOR descriptor\_name

Where:

descriptor\_name is the name of the descriptor. This value may take the form of a quoted string literal, or of a host variable.

The following example deallocates a descriptor named emp\_query:

EXEC SQL DEALLOCATE DESCRIPTOR emp\_query;

#### DECLARE CURSOR

Use the DECLARE CURSOR statement to define a cursor. The syntax of the statement is:

EXEC SQL [AT database\_name] DECLARE cursor\_name CURSOR FOR (select\_statement | statement\_name);

Where:

database\_name is the name of the database on which the cursor operates. This value may take the form of an identifier or of a host variable. If you do not specify a database name, the default value of database\_name is the default database.

cursor\_name is the name of the cursor.

select\_statement is the text of the SELECT statement that defines the cursor result set; the SELECT statement cannot contain an INTO clause.

statement\_name is the name of a SQL statement or block that defines the cursor result set.

The following example declares a cursor named employees:

EXEC SQL DECLARE employees CURSOR FOR

SELECT

empno, ename, sal, comm

FROM

emp;

The cursor generates a result set that contains the employee number, employee name, salary and commission for each employee record that is stored in the emp table.

#### DECLARE DATABASE

Use the DECLARE DATABASE statement to declare a database identifier for use in subsequent SQL statements (for example, in a CONNECT statement). The syntax is:

EXEC SQL DECLARE database\_name DATABASE;

Where:

database\_name specifies the name of the database.

The following example demonstrates declaring an identifier for the acctg database:

EXEC SQL DECLARE acctg DATABASE;

After invoking the command declaring acctg as a database identifier, the acctg database can be referenced by name when establishing a connection or in AT clauses.

This statement has no effect and is provided for Pro\*C compatibility only.

#### DECLARE STATEMENT

Use the DECLARE STATEMENT directive to declare an identifier for an SQL statement. Advanced Server supports two versions of the DECLARE STATEMENT directive:

EXEC SQL [database\_name] DECLARE statement\_name STATEMENT;

and

EXEC SQL DECLARE STATEMENT statement\_name;

Where:

statement\_name specifies the identifier associated with the statement.

database\_name specifies the name of the database. This value may take the form of an identifier or of a host variable that contains the identifier.

A typical usage sequence that includes the DECLARE STATEMENT directive might be:

EXEC SQL DECLARE give\_raise STATEMENT; // give\_raise is now a statement handle (not prepared)

EXEC SQL PREPARE give\_raise FROM :stmtText; // give\_raise is now associated with a statement

EXEC SQL EXECUTE give\_raise;

This statement has no effect and is provided for Pro\*C compatibility only.

#### DELETE

Use the DELETE statement to delete one or more rows from a table. The syntax for the ECPGPlus DELETE statement is the same as the syntax for the SQL statement, but you can use parameter markers and host variables any place that an expression is allowed. The syntax is:

[FOR *exec\_count*] DELETE FROM [ONLY] *table* [[AS] *alias*]

[USING *using*\_*list*]

[WHERE *condition* | WHERE CURRENT OF *cursor*\_*name*]  
 [{RETURNING|RETURN} \* | *output*\_*expression* [[ AS] *output*\_*name*] [, ...] INTO *host\_variable\_list* ]

Where:

Include the FOR *exec\_count* clause to specify the number of times the statement will execute; this clause is valid only if the VALUES clause references an array or a pointer to an array.

table is the name (optionally schema-qualified) of an existing table. Include the ONLY clause to limit processing to the specified table; if you do not include the ONLY clause, any tables inheriting from the named table are also processed.

alias is a substitute name for the target table.

using\_list is a list of table expressions, allowing columns from other tables to appear in the WHERE condition.

Include the WHERE clause to specify which rows should be deleted. If you do not include a WHERE clause in the statement, DELETE will delete all rows from the table, leaving the table definition intact.

condition is an expression, host variable or parameter marker that returns a value of type BOOLEAN. Those rows for which condition returns true will be deleted.

cursor\_name is the name of the cursor to use in the WHERE CURRENT OF clause; the row to be deleted will be the one most recently fetched from this cursor. The cursor must be a non-grouping query on the DELETE statements target table. You cannot specify WHERE CURRENT OF in a DELETE statement that includes a Boolean condition.

The RETURN/RETURNING clause specifies an output\_expression or host\_variable\_list that is returned by the DELETE command after each row is deleted:

output\_expression is an expression to be computed and returned by the DELETE command after each row is deleted. output\_name is the name of the returned column; include \* to return all columns.

host\_variable\_list is a comma-separated list of host variables and optional indicator variables. Each host variable receives a corresponding value from the RETURNING clause.

For example, the following statement deletes all rows from the emp table where the sal column contains a value greater than the value specified in the host variable, :max\_sal:

DELETE FROM emp WHERE sal > :max\_sal;

For more information about using the DELETE statement, please refer to the PostgreSQL core documentation available at:

<http://www.postgresql.org/docs/9.5/static/sql-delete.html>

#### DESCRIBE

Use the DESCRIBE statement to find the number of input values required by a prepared statement or the number of output values returned by a prepared statement. The DESCRIBE statement is used to analyze a SQL statement whose shape is unknown at the time you write your application.

The DESCRIBE statement populates an SQLDA descriptor; to populate a SQL descriptor, use the ALLOCATE DESCRIPTOR and DESCRIBE…DESCRIPTOR statements.

EXEC SQL DESCRIBE BIND VARIABLES FOR statement\_name INTO descriptor*;*

or

EXEC SQL DESCRIBE SELECT LIST FOR statement\_nameINTO descriptor*;*

Where:

statement\_name is the identifier associated with a prepared SQL statement or PL/SQL block.

descriptor is the name of C variable of type SQLDA\*. You must allocate the space for the descriptor by calling sqlald() (and initialize the descriptor) before executing the DESCRIBE statement.

When you execute the first form of the DESCRIBE statement, ECPG populates the given descriptor with a description of each inputvariable required by the statement. For example, given two descriptors:

SQLDA \*query\_values\_in;  
SQLDA \*query\_values\_out;

You might prepare a query that returns information from the emp table:

EXEC SQL PREPARE get\_emp FROM  
 "SELECT ename, empno, sal FROM emp WHERE empno = ?";

The command requires one input variable (for the parameter marker (?)).

EXEC SQL DESCRIBE BIND VARIABLES   
 FOR get\_emp INTO query\_values\_in;

After describing the bind variables for this statement, you can examine the descriptor to find the number of variables required and the type of each variable.

When you execute the second form, ECPG populates the given descriptor with a description of each value returned by the statement. For example, the following statement returns three values:

EXEC SQL DESCRIBE SELECT LIST   
 FOR get\_emp INTO query\_values\_out;

After describing the select list for this statement, you can examine the descriptor to find the number of returned values and the name and type of each value.

Before executingthe statement, you must bind a variable for each input value and a variable for each output value. The variables that you bind for the input values specify the actual values used by the statement. The variables that you bind for the output values tell ECPGPlus where to put the values when you execute the statement.

This is alternate Pro\*C compatible syntax for the DESCRIBE DESCRIPTOR statement.

#### DESCRIBE DESCRIPTOR

Use the DESCRIBE DESCRIPTOR statement to retrieve information about a SQL statement, and store that information in a SQL descriptor. Before using DESCRIBE DESCRIPTOR, you must allocate the descriptor with the ALLOCATE DESCRIPTOR statement. The syntax is:

EXEC SQL DESCRIBE [INPUT | OUTPUT] statement\_identifier   
 USING [SQL] DESCRIPTOR descriptor\_name;

Where:

statement\_name is the name of a prepared SQL statement.

descriptor\_name is the name of the descriptor. descriptor\_name can be a quoted string value or a host variable that contains the name of the descriptor.

If you include the INPUT clause, ECPGPlus populates the given descriptor with a description of each inputvariable *required* by the statement.

For example, given two descriptors:

EXEC SQL ALLOCATE DESCRIPTOR query\_values\_in;  
EXEC SQL ALLOCATE DESCRIPTOR query\_values\_out;

You might prepare a query that returns information from the emp table:

EXEC SQL PREPARE get\_emp FROM  
 "SELECT ename, empno, sal FROM emp WHERE empno = ?";

The command requires one input variable (for the parameter marker (?)).

EXEC SQL DESCRIBE INPUT get\_emp USING 'query\_values\_in';

After describing the bind variables for this statement, you can examine the descriptor to find the number of variables required and the type of each variable.

If you do not specify the INPUT clause, DESCRIBE DESCRIPTOR populates the specified descriptor with the values returned by the statement.

If you include the OUTPUT clause, ECPGPlus populates the given descriptor with a description of each value returned by the statement.

For example, the following statement returns three values:

EXEC SQL DESCRIBE OUTPUT FOR get\_emp USING 'query\_values\_out';

After describing the select list for this statement, you can examine the descriptor to find the number of returned values and the name and type of each value.

#### DISCONNECT

Use the DISCONNECT statement to close the connection to the server. The syntax is:

EXEC SQL DISCONNECT [connection\_name][CURRENT][DEFAULT][ALL];

Where:

connection\_name is the connection name specified in the CONNECT statement used to establish the connection. If you do not specify a connection name, the current connection is closed.

Include the CURRENT keyword to specify that ECPGPlus should close the most-recently used connection.

Include the DEFAULT keyword to specify that ECPGPlus should close the connection named DEFAULT. If you do not specify a name when opening a connection, ECPGPlus assigns the name, DEFAULT, to the connection.

Include the ALL keyword to instruct ECPGPlus to close all active connections.

The following example creates a connection (named hr\_connection) that connects to the hr database, and then disconnects from the connection:

/\* client.pgc\*/

int main()

{

EXEC SQL CONNECT TO hr AS connection\_name;

EXEC SQL DISCONNECT connection\_name;

return(0);

}

#### EXECUTE

Use the EXECUTE statement to execute a statement previously prepared using an EXEC SQL PREPARE statement. The syntax is:

EXEC SQL [FOR array\_size] EXECUTE statement\_name   
 [USING {DESCRIPTOR SQLDA\_descriptor   
 |:host\_variable [[INDICATOR] :indicator\_variable]}];

Where:

array\_size is an integer value or a host variable that contains an integer value that specifies the number of rows to be processed. If you omit the FOR clause, the statement is executed once for each member of the array.

statement\_name specifies the name assigned to the statement when the statement was created (using the EXEC SQL PREPARE statement).

Include the USING clause to supply values for parameters within the prepared statement:

Include the DESCRIPTORSQLDA\_descriptor clause to provide an SQLDA descriptor value for a parameter.

Use a host\_variable (and an optional indicator\_variable) to provide a user-specified value for a parameter.

The following example creates a prepared statement that inserts a record into the emp table:

EXEC SQL PREPARE add\_emp (numeric, text, text, numeric) AS

INSERT INTO emp VALUES($1, $2, $3, $4);

Each time you invoke the prepared statement, provide fresh parameter values for the statement:

EXEC SQL EXECUTE add\_emp USING 8000, 'DAWSON', 'CLERK', 7788;

EXEC SQL EXECUTE add\_emp USING 8001, 'EDWARDS', 'ANALYST', 7698;

#### EXECUTE DESCRIPTOR

Use the EXECUTE statement to execute a statement previously prepared by an EXEC SQL PREPARE statement, using an SQL descriptor. The syntax is:

EXEC SQL [FOR array\_size] EXECUTE statement\_identifier   
 [USING [SQL] DESCRIPTOR descriptor\_name]  
 [INTO [SQL] DESCRIPTOR descriptor\_name];

Where:

array\_size is an integer value or a host variable that contains an integer value that specifies the number of rows to be processed. If you omit the FOR clause, the statement is executed once for each member of the array.

statement\_identifier specifies the identifier assigned to the statement with the EXEC SQL PREPARE statement.

Include the USING clause to specify values for any input parameters required by the prepared statement.

Include the INTO clause to specify a descriptor into which the EXECUTE statement will write the results returned by the prepared statement.

descriptor\_name specifies the name of a descriptor (as a single-quoted string literal), or a host variable that contains the name of a descriptor.

The following example executes the prepared statement, give\_raise, using the values contained in the descriptor stmtText:

EXEC SQL PREPARE give\_raise FROM :stmtText;

EXEC SQL EXECUTE give\_raise USING DESCRIPTOR :stmtText;

#### EXECUTE...END EXEC

Use the EXECUTE…END-EXEC statement to embed an anonymous block into a client application. The syntax is:

EXEC SQL [AT database\_name] EXECUTE anonymous\_block END-EXEC;

Where:

database\_name is the database identifier or a host variable that contains the database identifier. If you omit the AT clause, the statement will be executed on the current default database.

anonymous\_block is an inline sequence of PL/pgSQL or SPL statements and declarations. You may include host variables and optional indicator variables within the block; each such variable is treated as an IN/OUT value.

The following example executes an anonymous block:

EXEC SQL EXECUTE  
 BEGIN  
 IF (current\_user = :admin\_user\_name) THEN  
 DBMS\_OUTPUT.PUT\_LINE('You are an administrator');  
 END IF;  
END-EXEC;

Please Note: the EXECUTE…END EXEC statement is supported only by Advanced Server.

#### EXECUTE IMMEDIATE

Use the EXECUTE IMMEDIATE statement to execute a string that contains a SQL command. The syntax is:

EXEC SQL [AT database\_name] EXECUTE IMMEDIATE command\_text;

Where:

database\_name is the database identifier or a host variable that contains the database identifier. If you omit the AT clause, the statement will be executed on the current default database.

command\_text is the command executed by the EXECUTE IMMEDIATE statement.

This dynamic SQL statement is useful when you don't know the text of an SQL statement (i.e., when writing a client application). For example, a client application may prompt a (trusted) user for a statement to execute. After the user provides the text of the statement as a string value, the statement is then executed with an EXECUTE IMMEDIATE command.

The statement text may not contain references to host variables. If the statement may contain parameter markers or returns one or more values, you must use the PREPARE and DESCRIBE statements.

The following example executes the command contained in the :command\_text host variable:

EXEC SQL EXECUTE IMMEDIATE :command\_text;

#### FETCH

Use the FETCH statement to return rows from a cursor into an SQLDA descriptor or a target list of host variables. Before using a FETCH statement to retrieve information from a cursor, you must prepare the cursor using DECLARE and OPEN statements. The statement syntax is:

EXEC SQL [FOR array\_size] FETCH cursor   
 { USING DESCRIPTOR SQLDA\_descriptor}|{ INTO target\_list};

Where:

array\_size is an integer value or a host variable that contains an integer value specifying the number of rows to fetch. If you omit the FOR clause, the statement is executed once for each member of the array.

cursor is the name of the cursor from which rows are being fetched, or a host variable that contains the name of the cursor.

If you include a USING clause, the FETCH statement will populate the specified SQLDA descriptor with the values returned by the server.

If you include an INTO clause, the FETCH statement will populate the host variables (and optional indicator variables) specified in the target\_list.

The following code fragment declares a cursor named employees that retrieves the employee number, name and salary from the emp table:

EXEC SQL DECLARE employees CURSOR FOR

SELECT empno, ename, esal FROM emp;

EXEC SQL OPEN emp\_cursor;

EXEC SQL FETCH emp\_cursor INTO :emp\_no, :emp\_name, :emp\_sal;

#### FETCH DESCRIPTOR

Use the FETCH DESCRIPTOR statement to retrieve rows from a cursor into an SQL descriptor. The syntax is:

EXEC SQL [FOR array\_size] FETCH cursor  
 INTO [SQL] DESCRIPTOR descriptor\_name;

Where:

array\_size is an integer value or a host variable that contains an integer value specifying the number of rows to fetch. If you omit the FOR clause, the statement is executed once for each member of the array.

cursor is the name of the cursor from which rows are fetched, or a host variable that contains the name of the cursor. The client must DECLARE and OPEN the cursor before calling the FETCH DESCRIPTOR statement.

Include the INTO clause to specify an SQL descriptor into which the EXECUTE statement will write the results returned by the prepared statement. descriptor\_name specifies the name of a descriptor (as a single-quoted string literal), or a host variable that contains the name of a descriptor. Prior to use, the descriptor must be allocated using an ALLOCATE DESCRIPTOR statement.

The following example allocates a descriptor named row\_desc that will hold the description and the values of a specific row in the result set. It then declares and opens a cursor for a prepared statement (my\_cursor), before looping through the rows in result set, using a FETCH to retrieve the next row from the cursor into the descriptor:

EXEC SQL ALLOCATE DESCRIPTOR 'row\_desc';

EXEC SQL DECLARE my\_cursor CURSOR FOR query;

EXEC SQL OPEN my\_cursor;

for( row = 0; ; row++ )

{

EXEC SQL BEGIN DECLARE SECTION;

int col;

EXEC SQL END DECLARE SECTION;

EXEC SQL FETCH my\_cursor INTO SQL DESCRIPTOR 'row\_desc';

#### GET DESCRIPTOR

Use the GET DESCRIPTOR statement to retrieve information from a descriptor. The GET DESCRIPTOR statement comes in two forms. The first form returns the number of values (or columns) in the descriptor.

EXEC SQL GET DESCRIPTOR descriptor\_name   
 :host\_variable = COUNT;

The second form returns information about a specific value (specified by the VALUE column\_numberclause).

EXEC SQL [FOR array\_size] GET DESCRIPTOR descriptor\_name   
 VALUE column\_number {:host\_variable = descriptor\_item {,…}};

Where:

array\_size is an integer value or a host variable that contains an integer value that specifies the number of rows to be processed. If you specify an array\_size, the host\_variable must be an array of that size; for example, if array\_size is 10, :host\_variable must be a 10-member array of host\_variables. If you omit the FOR clause, the statement is executed once for each member of the array.

descriptor\_name specifies the name of a descriptor (as a single-quoted string literal), or a host variable that contains the name of a descriptor.

Include the VALUE clause to specify the information retrieved from the descriptor.

column\_number identifies the position of the variable within the descriptor.

host\_variable specifies the name of the host variable that will receive the value of the item.

descriptor\_item specifies the type of the retrieved descriptor item.

ECPGPlus implements the following descriptor\_item types:

* TYPE
* LENGTH
* OCTET\_LENGTH
* RETURNED\_LENGTH
* RETURNED\_OCTET\_LENGTH
* PRECISION
* SCALE
* NULLABLE
* INDICATOR
* DATA
* NAME

The following code fragment demonstrates using a GET DESCRIPTOR statement to obtain the number of columns entered in a user-provided string:

EXEC SQL ALLOCATE DESCRIPTOR parse\_desc;

EXEC SQL PREPARE query FROM :stmt;

EXEC SQL DESCRIBE query INTO SQL DESCRIPTOR parse\_desc;

EXEC SQL GET DESCRIPTOR parse\_desc :col\_count = COUNT;

The example allocates an SQL descriptor (named parse\_desc), before using a PREPARE statement to syntax check the string provided by the user (:stmt). A DESCRIBE statement moves the user-provided string into the descriptor, parse\_desc. The call to EXEC SQL GET DESCRIPTOR interrogates the descriptor to discover the number of columns (:col\_count) in the result set.

#### INSERT

Use the INSERT statement to add one or more rows to a table. The syntax for the ECPGPlus INSERT statement is the same as the syntax for the SQL statement, but you can use parameter markers and host variables any place that a value is allowed. The syntax is:

[FOR *exec\_count*] INSERT INTO *table* [(*column* [, ...])]

{DEFAULT VALUES |

VALUES ({*expression* | DEFAULT} [, ...])[, ...] | *query*}

[RETURNING \* | *output*\_*expression* [[ AS ] *output*\_*name*] [, ...]]

Where:

Include the FOR *exec\_count* clause to specify the number of times the statement will execute; this clause is valid only if the VALUES clause references an array or a pointer to an array.

table specifies the (optionally schema-qualified) name of an existing table.

column is the name of a column in the table. The column name may be qualified with a subfield name or array subscript. Specify the DEFAULT VALUES clause to use default values for all columns.

expression is the expression, value, host variable or parameter marker that will be assigned to the corresponding column. Specify DEFAULT to fill the corresponding column with its default value.

query specifies a SELECT statement that supplies the row(s) to be inserted.

output\_expression is an expression that will be computed and returned by the INSERT command after each row is inserted. The expression can refer to any column within the table. Specify \* to return all columns of the inserted row(s).

output\_name specifies a name to use for a returned column.

The following example adds a row to the employees table:

INSERT INTO emp (empno, ename, job, hiredate)

VALUES ('8400', :ename, 'CLERK', '2011-10-31');

Note that the INSERT statement uses a host variable (:ename) to specify the value of the ename column.

For more information about using the INSERT statement, please refer to the PostgreSQL core documentation available at:

<http://www.postgresql.org/docs/9.5/static/sql-insert.html>

#### OPEN

Use the OPEN statement to open a cursor. The syntax is:

EXEC SQL [FOR array\_size] OPEN cursor[USING parameters];

Where *parameters* is one of the following:

DESCRIPTOR SQLDA\_descriptor  
or  
 host\_variable [ [ INDICATOR ] indicator\_variable, … ]

Where:

array\_size is an integer value or a host variable that contains an integer value specifying the number of rows to fetch. If you omit the FOR clause, the statement is executed once for each member of the array.

cursor is the name of the cursor being opened.

parameters is either DESCRIPTOR SQLDA\_descriptor or a comma-separated list of host variables (and optional indicator variables) that initialize the cursor. If specifying an SQLDA\_descriptor, the descriptor must be initialized with a DESCRIBE statement.

The OPEN statement initializes a cursor using the values provided in parameters. Once initialized, the cursor result set will remain unchanged unless the cursor is closed and re-opened. A cursor is automatically closed when an application terminates.

The following example declares a cursor named employees, that queries the emp table, returning the employee number, name, salary and commission of an employee whose name matches a user-supplied value (stored in the host variable, :emp\_name).

EXEC SQL DECLARE employees CURSOR FOR

SELECT

empno, ename, sal, comm

FROM

emp

WHERE ename = :emp\_name;

EXEC SQL OPEN employees;

...

After declaring the cursor, the example uses an OPEN statement to make the contents of the cursor available to a client application.

#### OPEN DESCRIPTOR

Use the OPEN DESCRIPTOR statement to open a cursor with a SQL descriptor. The syntax is:

EXEC SQL [FOR array\_size] OPEN cursor  
 [USING [SQL] DESCRIPTOR descriptor\_name]  
 [INTO [SQL] DESCRIPTOR descriptor\_name];

Where:

array\_size is an integer value or a host variable that contains an integer value specifying the number of rows to fetch. If you omit the FOR clause, the statement is executed once for each member of the array.

cursor is the name of the cursor being opened.

descriptor\_name specifies the name of an SQL descriptor (in the form of a single-quoted string literal) or a host variable that contains the name of an SQL descriptor that contains the query that initializes the cursor.

For example, the following statement opens a cursor (named emp\_cursor), using the host variable, :employees:

EXEC SQL OPEN emp\_cursor USING DESCRIPTOR :employees;

#### PREPARE

Prepared statements are useful when a client application must perform a task multiple times; the statement is parsed, written and planned only once, rather than each time the statement is executed, saving repetitive processing time.

Use the PREPARE statement to prepare an SQL statement or PL/pgSQL block for execution. The statement is available in two forms; the first form is:

EXEC SQL [AT database\_name] PREPARE statement\_name   
 FROM sql\_statement;

The second form is:

EXEC SQL [AT database\_name] PREPARE statement\_name   
 AS sql\_statement;

Where:

database\_name is the database identifier or a host variable that contains the database identifier against which the statement will execute. If you omit the AT clause, the statement will execute against the current default database.

statement\_name is the identifier associated with a prepared SQL statement or PL/SQL block.

sql\_statement may take the form of a SELECT statement, a single-quoted string literal or host variable that contains the text of an SQL statement.

To include variables within a prepared statement, substitute placeholders ($1, $2, $3, etc.) for statement values that might change when you PREPARE the statement. When you EXECUTE the statement, provide a value for each parameter. The values must be provided in the order in which they will replace placeholders.

The following example creates a prepared statement (named add\_emp) that inserts a record into the emp table:

EXEC SQL PREPARE add\_emp (int, text, text, numeric) AS

INSERT INTO emp VALUES($1, $2, $3, $4);

Each time you invoke the statement, provide fresh parameter values for the statement:

EXEC SQL EXECUTE add\_emp(8003, 'Davis', 'CLERK', 2000.00);

EXEC SQL EXECUTE add\_emp(8004, 'Myer', 'CLERK', 2000.00);

Please note: A client application must issue a PREPARE statement within each session in which a statement will be executed; prepared statements persist only for the duration of the current session.

#### ROLLBACK

Use the ROLLBACK statement to abort the current transaction, and discard any updates made by the transaction. The syntax is:

EXEC SQL [AT database\_name] ROLLBACK [WORK]   
 [ { TO [SAVEPOINT] savepoint } | RELEASE ]

Where:

database\_name is the database identifier or a host variable that contains the database identifier against which the statement will execute. If you omit the AT clause, the statement will execute against the current default database.

Include the TO clause to abort any commands that were executed after the specified savepoint; use the SAVEPOINT statement to define the savepoint. If you omit the TO clause, the ROLLBACK statement will abort the transaction, discarding all updates.

Include the RELEASE clause to cause the application to execute an EXEC SQL COMMIT RELEASE and close the connection.

Use the following statement to rollback a complete transaction:

EXEC SQL ROLLBACK;

Invoking this statement will abort the transaction, undoing all changes, erasing any savepoints, and releasing all transaction locks. If you include a savepoint (my\_savepoint in the following example):

EXEC SQL ROLLBACK TO SAVEPOINT my\_savepoint;

Only the portion of the transaction that occurred after the my\_savepoint is rolled back; my\_savepoint is retained, but any savepoints created after my\_savepoint will be erased.

Rolling back to a specified savepoint releases all locks acquired after the savepoint.

#### SAVEPOINT

Use the SAVEPOINT statement to define a savepoint; a savepoint is a marker within a transaction. You can use a ROLLBACK statement to abort the current transaction, returning the state of the server to its condition prior to the specified savepoint. The syntax of a SAVEPOINT statement is:

EXEC SQL [AT database\_name] SAVEPOINT savepoint\_name

Where:

database\_name is the database identifier or a host variable that contains the database identifier against which the savepoint resides. If you omit the AT clause, the statement will execute against the current default database.

savepoint\_name is the name of the savepoint. If you re-use a savepoint\_name, the original savepoint is discarded.

Savepoints can only be established within a transaction block. A transaction block may contain multiple savepoints.

To create a savepoint named my\_savepoint, include the statement:

EXEC SQL SAVEPOINT my\_savepoint;

#### SELECT

ECPGPlus extends support of the SQL SELECT statement by providing the INTO host\_variables clause. The clause allows you to select specified information from an Advanced Server database into a host variable. The syntax for the SELECT statement is:

EXEC SQL [AT database\_name]

SELECT

[ hint ]

[ ALL | DISTINCT [ ON(expression, ...) ]]

select\_list INTO host\_variables

[ FROM from\_item [, from\_item ]...]

[ WHERE condition ]

[ hierarchical\_query\_clause ]

[ GROUP BY expression [, ...]]

[ HAVING condition ]

[ { UNION [ ALL ] | INTERSECT | MINUS } (subquery) ]

[ ORDER BY expression[order\_by\_options]]

[ LIMIT { count| ALL }]

[ OFFSET start [ ROW | ROWS ] ]

[ FETCH { FIRST | NEXT } [ count ] { ROW | ROWS } ONLY ]

[ FOR { UPDATE | SHARE } [OF table\_name [, ...]][NOWAIT ][...]]

Where:

database\_name is the name of the database (or host variable that contains the name of the database) in which the table resides. This value may take the form of an unquoted string literal, or of a host variable.

host\_variables is a list of host variables that will be populated by the SELECT statement. If the SELECT statement returns more than a single row, host\_variables must be an array.

ECPGPlus provides support for the additional clauses of the SQL SELECT statement as documented in the PostgreSQL Core documentation, available at:

<http://www.postgresql.org/docs/9.5/static/sql-select.html>

To use the INTO host\_variables clause, include the names of defined host variables when specifying the SELECT statement. For example, the following SELECT statement populates the :emp\_name and :emp\_sal host variables with a list of employee names and salaries:

EXEC SQL SELECT ename, sal   
 INTO :emp\_name, :emp\_sal   
 FROM emp  
 WHERE empno = 7988;

The enhanced SELECT statement also allows you to include parameter markers (question marks) in any clause where a value would be permitted. For example, the following query contains a parameter marker in the WHERE clause:

SELECT \* FROM emp WHERE dept\_no = ?;

This SELECT statement allows you to provide a value at run-time for the dept\_no parameter marker.

#### SET CONNECTION

There are (at least) three reasons you may need more than one connection in a given client application:

* You may want different privileges for different statements.
* You may need to interact with multiple databases within the same client.
* Multiple threads of execution (within a client application) cannot share a connection concurrently.

The syntax for the SET CONNECTION statement is:

EXEC SQL SET CONNECTION connection\_name;

Where:

connection\_name is the name of the connection to the database.

To use the SET CONNECTION statement, you should open the connection to the database using the second form of the CONNECT statement; include the AS clause to specify a connection\_name.

By default, the current thread uses the current connection; use the SET CONNECTION statement to specify a default connection for the current thread to use. The default connection is only used when you execute an EXEC SQL statement that does not explicitly specify a connection name. For example, the following statement will use the default connection because it does not include an AT connection\_name clause:

EXEC SQL DELETE FROM emp;

This statement will not use the default connection because it specifies a connection name using the AT connection\_name clause:

EXEC SQL AT acctg\_conn DELETE FROM emp;

For example, a client application that creates and maintains multiple connections (such as):

EXEC SQL CONNECT TO edb AS acctg\_conn  
 USER 'alice' IDENTIFIED BY 'acctpwd';

and

EXEC SQL CONNECT TO edb AS hr\_conn  
 USER 'bob' IDENTIFIED BY 'hrpwd';

Can change between the connections with the SET CONNECTION statement:

SET CONNECTION acctg\_conn;

or

SET CONNECTION hr\_conn;

The server will use the privileges associated with the connection when determining the privileges available to the connecting client. When using the acctg\_conn connection, the client will have the privileges associated with the role, alice; when connected using hr\_conn, the client will have the privileges associated with bob.

#### SET DESCRIPTOR

Use the SET DESCRIPTOR statement to assign a value to a descriptor area using information provided by the client application in the form of a host variable or an integer value. The statement comes in two forms; the first form is:

EXEC SQL [FOR array\_size] SET DESCRIPTOR descriptor\_name  
 VALUE column\_number descriptor\_item = host\_variable;

The second form is:

EXEC SQL [FOR array\_size] SET DESCRIPTOR descriptor\_name  
 COUNT = integer;

Where:

array\_size is an integer value or a host variable that contains an integer value specifying the number of rows to fetch. If you omit the FOR clause, the statement is executed once for each member of the array.

descriptor\_name specifies the name of a descriptor (as a single-quoted string literal), or a host variable that contains the name of a descriptor.

Include the VALUE clause to describe the information stored in the descriptor.

column\_number identifies the position of the variable within the descriptor.

descriptor\_item specifies the type of the descriptor item.

host\_variable specifies the name of the host variable that contains the value of the item.

ECPGPlus implements the following descriptor\_item types:

* TYPE
* LENGTH
* [REF] INDICATOR
* [REF] DATA
* [REF] RETURNED LENGTH

For example, a client application might prompt a user for a dynamically created query:

query\_text = promptUser("Enter a query");

To execute a dynamically created query, you must first prepare the query (parsing and validating the syntax of the query), and then describethe inputparameters found in the query using the EXEC SQL DESCRIBE INPUT statement.

EXEC SQL ALLOCATE DESCRIPTOR query\_params;  
EXEC SQL PREPARE emp\_query FROM :query\_text;  
  
EXEC SQL DESCRIBE INPUT emp\_query   
 USING SQL DESCRIPTOR 'query\_params';

After describing the query, the query\_params descriptor contains information about each parameter required by the query.

For this example, we'll assume that the user has entered:

SELECT ename FROM emp WHERE sal > ? AND job = ?;,

In this case, the descriptor describes two parameters:

* one for sal > ?
* one for job = ?

To discover the number of parameter markers (question marks) in the query (and therefore, the number of values you must provide before executing the query), use:

EXEC SQL GET DESCRIPTOR … :host\_variable = COUNT;

Then, you can use EXEC SQL GET DESCRIPTOR to retrieve the name of each parameter. You can also use EXEC SQL GET DESCRIPTOR to retrieve the type of each parameter (along with the number of parameters) from the descriptor, or you can supply each *value* in the form of a character string and ECPG will convert that string into the required data type.

The data type of the first parameter is numeric; the type of the second parameter is varchar. The name of the first parameter is sal; the name of the second parameter is job.

Next, loop through each parameter, prompting the user for a value, and store those values in host variables. You can use GET DESCRIPTOR … COUNT to find the number of parameters in the query.

EXEC SQL GET DESCRIPTOR 'query\_params'   
 :param\_count = COUNT;  
  
for(param\_number = 1;   
 param\_number <= param\_count;   
 param\_number++)  
{

Use GET DESCRIPTOR to copy the name of the parameter into the param\_name host variable:

EXEC SQL GET DESCRIPTOR 'query\_params'   
 VALUE :param\_number :param\_name = NAME;  
  
 reply = promptUser(param\_name);  
 if (reply == NULL)  
 reply\_ind = 1; /\* NULL \*/  
else  
 reply\_ind = 0; /\* NOT NULL \*/

To associate a value with each parameter, you use the EXEC SQL SET DESCRIPTOR statement. For example:

EXEC SQL SET DESCRIPTOR 'query\_params'   
 VALUE :param\_number DATA = :reply;  
 EXEC SQL SET DESCRIPTOR 'query\_params'   
 VALUE :param\_number INDICATOR = :reply\_ind;  
}

Now, you can use the EXEC SQL EXECUTE DESCRIPTOR statement to execute the prepared statement on the server.

#### UPDATE

Use an UPDATE statement to modify the data stored in a table. The syntax is:

EXEC SQL [AT database\_name][FOR exec\_count]   
 UPDATE [ ONLY ] table [ [ AS ] alias ]

SET {column = { expression | DEFAULT } |

(column [, ...]) = ({ expression|DEFAULT } [, ...])} [, ...]

[ FROM from\_list ]

[ WHERE condition | WHERE CURRENT OF cursor\_name ]  
 [ RETURNING \* | output\_expression [[ AS ] output\_name] [, ...] ]

Where:

database\_name is the name of the database (or host variable that contains the name of the database) in which the table resides. This value may take the form of an unquoted string literal, or of a host variable.

Include the FOR exec\_count clause to specify the number of times the statement will execute; this clause is valid only if the SET or WHERE clause contains an array.

ECPGPlus provides support for the additional clauses of the SQL UPDATE statement as documented in the PostgreSQL Core documentation, available at:

<http://www.postgresql.org/docs/9.5/static/sql-update.html>

A host variable can be used in any clause that specifies a value. To use a host variable, simply substitute a defined variable for any value associated with any of the documented UPDATE clauses.

The following UPDATE statement changes the job description of an employee (identified by the :ename host variable) to the value contained in the :new\_job host variable, and increases the employee’s salary, by multiplying the current salary by the value in the :increase host variable:

EXEC SQL UPDATE emp   
 SET job = :new\_job, sal = sal \* :increase  
 WHERE ename = :ename;

The enhanced UPDATE statement also allows you to include parameter markers (question marks) in any clause where an input value would be permitted. For example, we can write the same update statement with a parameter marker in the WHERE clause:

EXEC SQL UPDATE emp   
 SET job = ?, sal = sal \* ?  
 WHERE ename = :ename;

This UPDATE statement could allow you to prompt the user for a new value for the job column and provide the amount by which the sal column is incremented for the employee specified by :ename.

#### WHENEVER

Use the WHENEVER statement to specify the action taken by a client application when it encounters an SQL error or warning. The syntax is:

EXEC SQL WHENEVER condition action;

The following table describes the different conditions that might trigger an action:

|  |  |
| --- | --- |
| Condition | Description |
| NOT FOUND | The server returns a NOT FOUND condition when it encounters a SELECT that returns no rows, or when a FETCH reaches the end of a result set. |
| SQLERROR | The server returns an SQLERROR condition when it encounters a serious error returned by an SQL statement. |
| SQLWARNING | The server returns an SQLWARNING condition when it encounters a non-fatal warning returned by an SQL statement. |

The following table describes the actions that result from a client encountering a condition:

|  |  |
| --- | --- |
| Action | Description |
| CALL function([args]) | Instructs the client application to call the named function. |
| CONTINUE | Instructs the client application to proceed to the next statement. |
| DO BREAK | Instructs the client application to a C break statement. A break statement may appear in a loop or a switch statement. If executed, the break statement terminate the loop or the switch statement. |
| DO CONTINUE | Instructs the client application to emit a C continue statement. A continue statement may only exist within a loop, and if executed, will cause the flow of control to return to the top of the loop. |
| DO function([args]) | Instructs the client application to call the named function. |
| GOTO label or  GO TO label | Instructs the client application to proceed to the statement that contains the label. |
| SQLPRINT | Instructs the client application to print a message to standard error. |
| STOP | Instructs the client application to stop execution. |

The following code fragment prints a message if the client application encounters a warning, and aborts the application if it encounters an error:

EXEC SQL WHENEVER SQLWARNING SQLPRINT;

EXEC SQL WHENEVER SQLERROR STOP;

Include the following code to specify that a client should continue processing after warning a user of a problem:

EXEC SQL WHENEVER SQLWARNING SQLPRINT;

Include the following code to call a function if a query returns no rows, or when a cursor reaches the end of a result set:

EXEC SQL WHENEVER NOT FOUND CALL error\_handler(\_\_LINE\_\_);

# Open Client Library

The Open Client Library provides application interoperability with the Oracle Call Interface – an application that was formerly “locked in” can now work with either an Advanced Server or an Oracle database with minimal to no changes to the application code. The EnterpriseDB implementation of the Open Client Library is written in C.

Please note: EnterpriseDB does not support use of the Open Client Library with Oracle Real Application Clusters (RAC) and Oracle Exadata; the aforementioned Oracle products have not been evaluated nor certified with this EnterpriseDB product.

## Comparison with Oracle Call Interface

The following diagram compares the Open Client Library and Oracle Call Interface application stacks.

OracleTM Call Interface

EnterpriseDB’s

Open Client Library

Application Programs

Same

Application Programs

Published API

Compatible API

**Oracle’s OCI-Lib**

**Open Client Library**

libpq

**Wire-Level Protocols**

Wire-Level Protocols

“Black

Box”

Open

Source

**Oracle DBMS**

**Advanced Server**

**Oracle Database**

**EDB Postgres Database**

**UPI**

Figure 3 Open Client Library

## Compiling and Linking a Program

The EnterpriseDB Open Client Library allows applications written using the Oracle Call Interface API to connect to and access an EnterpriseDB database with minimal changes to the C source code. The EnterpriseDB Open Client Library files are named:

On Linux:

libedboci.so

On Windows:

edboci.dll

The files are installed in the connectors/edb-oci/lib subdirectory.

***Compiling and Linking a Sample Program***

The following example compiles and links the sample program edb\_demo.c in a Linux environment. The edb\_demo.c is located in the connectors/edb-oci/samples subdirectory.

1. Set the ORACLE\_HOME and EDB\_HOME environment variables.

Set ORACLE\_HOME to the complete pathname of the Oracle home directory.

For example:

export ORACLE\_HOME=/usr/lib/oracle/xe/app/oracle/product/10.2.0/server

Set EDB\_HOME to the complete pathname of the home directory.

For example:

export EDB\_HOME=/opt/PostgresPlus

1. Set LD\_LIBRARY\_PATH to the complete path of libpthread.so. By default, libpthread.so is located in /usr/lib.

export LD\_LIBRARY\_PATH=/usr/lib:$LD\_LIBRARY\_PATH

1. Set LD\_LIBRARY\_PATH to include the Advanced Server Open Client library. By default, libiconv.so.2 is located in $EDB\_HOME/connectors/edb-oci/lib.

export  
LD\_LIBRARY\_PATH=$EDB\_HOME/connectors/edb‑oci:$EDB\_HOME/  
connectors/edb-oci/lib:$LD\_LIBRARY\_PATH

1. Then, compile and link the OCI API program.

cd $EDB\_HOME/connectors/edb-oci/samples

make

## Ref Cursor Support

The Advanced Server Open Client Library supports the use of REF CURSOR's as OUT parameters in PL/SQL procedures that are compatible with Oracle. Support is provided through the following API's:

* OCIBindByName
* OCIBindByPos
* OCIBindDynamic
* OCIStmtPrepare
* OCIStmtExecute
* OCIStmtFetch
* OCIAttrGet

OCL also supports the SQLT\_RSET data type.

The following example demonstrates how to invoke a stored procedure that opens a cursor and returns a REF CURSOR as an output parameter. The code sample assumes that a PL/SQL procedure named openCursor (with an OUT parameter of type REF CURSOR) has been created on the database server, and that the required handles have been allocated:

char \* openCursor =  
 "begin \  
 openCursor(:cmdRefCursor); \  
 end;";  
OCIStmt \*stmtOpenRefCursor;  
OCIStmt \*stmtUseRefCursor;

Allocate handles for executing a stored procedure to open and use the REF CURSOR:

/\* Handle for the stored procedure to open the ref cursor \*/  
 OCIHandleAlloc((dvoid \*) envhp,  
 (dvoid \*\*) &stmtOpenRefCursor,   
 OCI\_HTYPE\_STMT,  
 0,  
 (dvoid \*\*) NULL));

/\* Handle for using the Ref Cursor \*/  
 OCIHandleAlloc((dvoid \*) envhp,  
 (dvoid \*\*) &stmtUseRefCursor,   
 OCI\_HTYPE\_STMT,  
 0,  
 (dvoid \*\*) NULL));

Then, prepare the PL/SQL block that is used to open the REF CURSOR:

OCIStmtPrepare(stmtOpenRefCursor,   
 errhp,   
 (text \*) openCursor,  
 (ub4) strlen(openCursor),   
 OCI\_NTV\_SYNTAX,   
 OCI\_DEFAULT));

Bind the PL/SQL openCursor OUT parameter:

OCIBindByPos(stmtOpenRefCursor,   
 &bndplrc1,   
 errhp,   
 1,   
 (dvoid\*) &stmtUseRefCursor,   
 /\* the returned ref cursor \*/  
 0,  
 SQLT\_RSET,   
 /\* SQLT\_RSET type representing cursor \*/  
 (dvoid \*) 0,  
 (ub2 \*) 0,  
 (ub2) 0,  
 (ub4) 0,  
 (ub4 \*) 0,  
 OCI\_DEFAULT));

Use the stmtOpenRefCursor statement handle to call the openCursor procedure:

OCIStmtExecute(svchp,   
 stmtOpenRefCursor,   
 errhp,   
 1,   
 0,   
 0,   
 0,   
 OCI\_DEFAULT);

At this point, the stmtUseRefCursor statement handle contains the reference to the cursor. To obtain the information, define output variables for the ref cursor:

/\* Define the output variables for the ref cursor \*/  
 OCIDefineByPos(stmtUseRefCursor,   
 &defnEmpNo,   
 errhp,  
 (ub4) 1,   
 (dvoid \*) &empNo,  
 (sb4) sizeof(empNo),   
 SQLT\_INT,   
 (dvoid \*) 0,  
 (ub2 \*)0,  
 (ub2 \*)0,   
 (ub4) OCI\_DEFAULT));

Then, fetch the first row of the result set into the target variables:

/\* Fetch the cursor data \*/  
 OCIStmtFetch(stmtUseRefCursor,   
 errhp,   
 (ub4) 1,   
 (ub4) OCI\_FETCH\_NEXT,  
 (ub4) OCI\_DEFAULT))

## OCL Function Reference

The following tables list the functions supported in the Open Client Library. Note that any and all header files must be supplied by the user. Advanced Server does not supply any such files.

### Connect, Authorize and Initialize Functions

Table 9‑‑ Connect, Authorize, Terminate and Initialize Functions

| Function | Description |
| --- | --- |
| OCIBreak | Aborts the specified OCI function. |
| OCIEnvCreate | Create an OCI environment. |
| OCIEnvInit | Initialize an OCI environment handle. |
| OCIInitialize | Initialize the OCI environment. |
| OCILogoff | Release a session. |
| OCILogon | Create a logon connection. |
| OCILogon2 | Create a logon session in various modes. |
| OCIReset | Resets the current operation/protocol. |
| OCIServerAttach | Establish an access path to a data source. For information about using the tnsnames.ora file, see [Section 9.6](#_Using_the_tnsnames.ora_1). |
| OCIServerDetach | Remove access to a data source. |
| OCISessionBegin | Create a user session. |
| OCISessionEnd | End a user session. |
| OCISessionGet | Get session from session pool. |
| OCISessionRelease | Release a session. |
| OCITerminate | Detach from shared memory subsystem. |

#### Using the tnsnames.ora File

The OCIServerAttach method uses a connection descriptor specified in the dblink parameter of the tnsnames.ora file. Use the tnsnames.ora file, compatible with Oracle databases, to specify database connection addresses. Advanced Server searches the user's home directory for a file named tnsnames.ora. If Advanced Server doesn't find the tnsnames.ora file in the user's home directory, it searches the path specified by TNS\_ADMIN.

The sample tnsnames.ora file contains:

EDBX =  
(DESCRIPTION =  
 (ADDRESS = (PROTOCOL = TCP)(HOST = localhost)(PORT = 5444))  
 (CONNECT\_DATA = (SERVER = DEDICATED)(SID = edb))  
)

Any parameters not included in the sample, are ignored by the Open Client Library. In the sample, SID refers to the database named edb, in the cluster running on server 'localhost' at port 5444.

A C program call to OCIServerAttach that uses the tnsnames.ora file will look like:

static text \*username = (text \*) "enterprisedb";  
static text \*password = (text \*) "edb";  
static text \*attach\_str = "EDBX";  
OCIServerAttach( srvhp, errhp, attach\_str, strlen(attach\_str), 0);

If you don't have a tnsnames.ora file, supply the connection string parameter in the form //localhost:5444/edbx.

### Handle and Descriptor Functions

Table 9‑7‑2 Handle and Descriptor Functions

| Function | Description |
| --- | --- |
| OCIAttrGet | Get handle attributes. Advanced server supports the following handle attributes: OCI\_ATTR\_USERNAME, OCI\_ATTR\_PASSWORD, OCI\_ATTR\_SERVER, OCI\_ATTR\_ENV, OCI\_ATTR\_SESSION, OCI\_ATTR\_ROW\_COUNT, OCI\_ATTR\_CHARSET\_FORM, OCI\_ATTR\_CHARSET\_ID, EDB\_ATTR\_STMT\_LEVEL\_TX, OCI\_ATTR\_MODULE |
| OCIAttrSet | Set handle attributes. Advanced server supports the following handle attributes: OCI\_ATTR\_USERNAME, OCI\_ATTR\_PASSWORD, OCI\_ATTR\_SERVER, OCI\_ATTR\_ENV, OCI\_ATTR\_SESSION, OCI\_ATTR\_ROW\_COUNT, OCI\_ATTR\_CHARSET\_FORM, OCI\_ATTR\_CHARSET\_ID, EDB\_ATTR\_STMT\_LEVEL\_TX, OCI\_ATTR\_MODULE |
| OCIDescriptorAlloc | Allocate and initialize a descriptor. |
| OCIDescriptorFree | Free an allocated descriptor. |
| OCIHandleAlloc | Allocate and initialize a handle. |
| OCIHandleFree | Free an allocated handle. |
| OCIParamGet | Get a parameter descriptor. |
| OCIParamSet | Set a parameter descriptor. |

#### EDB\_ATTR\_EMPTY\_STRINGS

By default, Advanced Server will treat an empty string as a NULL value. You can use the EDB\_ATTR\_EMPTY\_STRINGS environment attribute to control the behavior of the OCL when mapping empty strings. To modify the mapping behavior, use the OCIAttrSet() function to set EDB\_ATTR\_EMPTY\_STRINGS to one of the following:

|  |  |
| --- | --- |
| Value | Description |
| OCI\_DEFAULT | Treat an empty string as a NULL value. |
| EDB\_EMPTY\_STRINGS\_NULL | Treat an empty string as a NULL value. |
| EDB\_EMPTY\_STRINGS\_EMPTY | Treat an empty string as a string of zero length. |

To find the value of EDB\_ATTR\_EMPTY\_STRINGS, query OCIAttrGet().

#### EDB\_ATTR\_HOLDABLE

Advanced Server supports statements that execute as WITH HOLD cursors. The EDB\_ATTR\_HOLDABLE attribute specifies which statements execute as WITH HOLD cursors. The EDB\_ATTR\_HOLDABLE attribute can be set to any of the following three values:

* EDB\_WITH\_HOLD - execute as a WITH HOLD cursor
* EDB\_WITHOUT\_HOLD - execute using a protocol-level prepared statement
* OCI\_DEFAULT - see the definition that follows

You can set the attribute in an OCIStmt handle or an OCIServer handle. When you create an OCIServer handle or an OCIStmt handle, the EDB\_ATTR\_HOLDABLE attribute for that handle is set to OCI\_DEFAULT.

You can change the EDB\_ATTR\_HOLDABLE attribute for a handle by calling OCIAttrSet() and retrieve the attribute by calling OCIAttrGet().

When Advanced Server executes a SELECT statement, it examines the EDB\_ATTR\_HOLDABLE attribute in the OCIServer handle. If that attribute is set to EDB\_WITH\_HOLD, the query is executed as a WITH HOLD cursor.

If the EDB\_ATTR\_HOLDABLE attribute in the OCIServer handle is set to EDB\_WITHOUT\_HOLD, the query is executed as a normal prepared statement.

If the EDB\_ATTR\_HOLDABLE attribute in the OCIServer handle is set to OCI\_DEFAULT, Advanced Server uses the value of the EDB\_ATTR\_HOLDABLE attribute in the OCIServer handle (if the EDB\_ATTR\_HOLDABLE attribute in the OCIServer is set to EDB\_WITH\_HOLD, the query executes as a WITH HOLD cursor, otherwise, the query executes as a protocol-prepared statement).

#### EDB\_ATTR\_STMT\_LVL\_TX

Unless otherwise instructed, the OCL library will ROLLBACK the current transaction whenever the server reports an error. If you choose, you can override the automatic ROLLBACK with the edb\_stmt\_level\_tx parameter, which preserves modifications within a transaction, even if one (or several) statements raise an error within the transaction. For more information about edb\_stmt\_level\_tx, see [Section 1.3.4](#_edb_stmt_level_tx).

You can use the OCIServer attribute with OCIAttrSet() and OCIAttrGet()to enable or disable EDB\_ATTR\_STMT\_LEVEL\_TX. By default, edb\_stmt\_level\_tx is disabled. To enable edb\_stmt\_level\_tx, the client application must call OCIAttrSet():

OCIServer \*server = myServer;  
ub1 enabled = 1;  
  
OCIAttrSet(server, OCI\_HTYPE\_SERVER, &enabled,  
 sizeof(enabled), EDB\_ATTR\_STMT\_LEVEL\_TX, err);

To disable edb\_stmt\_level\_tx:

OCIServer \*server = myServer;  
ub1 enabled = 0;  
  
OCIAttrSet(server, OCI\_HTYPE\_SERVER, &enabled,  
 sizeof(enabled), EDB\_ATTR\_STMT\_LEVEL\_TX, err);

### Bind, Define and Describe Functions

Table 9‑7‑3 Bind, Define, and Describe Functions

| Function | Description |
| --- | --- |
| OCIBindByName | Bind by name. |
| OCIBindByPos | Bind by position. |
| OCIBindDynamic | Set additional attributes after bind. |
| OCIBindArrayOfStruct | Bind an array of structures for bulk operations. |
| OCIDefineArrayOfStruct | Specify the attributes of an array. |
| OCIDefineByPos | Define an output variable association. |
| OCIDefineDynamic | Set additional attributes for define. |
| OCIDescribeAny | Describe existing schema objects. |
| OCIStmtGetBindInfo | Get bind and indicator variable names and handle. |
| OCIUserCallbackRegister | Define a user-defined callback. |

### Statement Functions

Table 9‑7‑4 Statement Functions

| Function | Description |
| --- | --- |
| OCIStmtExecute | Execute a prepared SQL statement. |
| OCIStmtFetch | Fetch rows of data (deprecated). |
| OCIStmtFetch2 | Fetch rows of data. |
| OCIStmtPrepare | Prepare a SQL statement. |
| OCIStmtPrepare2 | Prepare a SQL statement. |
| OCIStmtRelease | Release a statement handle. |

### Transaction Functions

Table 9‑7‑5 Transaction Functions

| Function | Description |
| --- | --- |
| OCITransCommit | Commit a transaction. |
| OCITransRollback | Roll back a transaction. |

### XA Functions

Table 9‑7‑6 XA Functions

| Function | Description |
| --- | --- |
| xaoEnv | Returns OCL environment handle. |
| xaoSvcCtx | Returns OCL service context. |

#### xaoSvcCtx

In order to use the xaoSvcCtx function, extensions in the xaoSvcCtx or xa\_open connection string format must be provided as follows:

Oracle\_XA{+required\_fields ...}

Where required\_fields are the following:

HostName=host\_ip\_address specifies the IP address of the Advanced Server database.

PortNumber=host\_port\_number specifies the port number on which Advanced Server is running.

SqlNet=dbname specifies the database name.

Acc=P/username/password specifies the database username and password. password may be omitted in which case the field is specified as Acc=P/username/.

AppName=app\_id specifies a number that identifies the application.

The following is an example of the connection string:

Oracle\_XA+HostName=192.168.1.1+PortNumber=1533+SqlNet=XE+Acc=P/user/password+AppName=1234

### Date and Datetime Functions

Table 9‑7‑7 Date and Datetime Functions

| Function | Description |
| --- | --- |
| OCIDateAddDays | Add or subtract a number of days. |
| OCIDateAddMonths | Add or subtract a number of months. |
| OCIDateAssign | Assign a date. |
| OCIDateCheck | Check if the given date is valid. |
| OCIDateCompare | Compare two dates. |
| OCIDateDaysBetween | Find the number of days between two dates. |
| OCIDateFromText | Convert a string to a date. |
| OCIDateGetDate | Get the date portion of a date. |
| OCIDateGetTime | Get the time portion of a date. |
| OCIDateLastDay | Get the date of the last day of the month. |
| OCIDateNextDay | Get the date of the next day. |
| OCIDateSetDate | Set the date portion of a date. |
| OCIDateSetTime | Set the time portion of a date. |
| OCIDateSysDate | Get the current system date and time. |
| OCIDateToText | Convert a date to a string. |
| OCIDateTimeAssign | Perform datetime assignment. |
| OCIDateTimeCheck | Check if the date is valid. |
| OCIDateTimeCompare | Compare two datetime values. |
| OCIDateTimeConstruct | Construct a datetime descriptor. |
| OCIDateTimeConvert | Convert one datetime type to another. |
| OCIDateTimeFromArray | Convert an array of size OCI\_DT\_ARRAYLEN to an OCIDateTime descriptor. |
| OCIDateTimeFromText | Convert the given string to Oracle datetime type in the OCIDateTime descriptor according to the specified format. |
| OCIDateTimeGetDate | Get the date portion of a datetime value. |
| OCIDateTimeGetTime | Get the time portion of a datetime value. |
| OCIDateTimeGetTimeZoneName | Get the time zone name portion of a datetime value. |
| OCIDateTimeGetTimeZoneOffset | Get the time zone (hour, minute) portion of a datetime value. |
| OCIDateTimeSubtract | Take two datetime values as input and return their difference as an interval. |
| OCIDateTimeSysTimeStamp | Get the system current date and time as a timestamp with time zone. |
| OCIDateTimeToArray | Convert an OCIDateTime descriptor to an array. |
| OCIDateTimeToText | Convert the given date to a string according to the specified format. |

### Interval Functions

Table 9‑7‑8 Interval Functions

| Function | Description |
| --- | --- |
| OCIIntervalAdd | Adds two interval values. |
| OCIIntervalAssign | Copies one interval value into another interval value. |
| OCIIntervalCompare | Compares two interval values. |
| OCIIntervalGetDaySecond | Extracts days, hours, minutes, seconds and fractional seconds from an interval. |
| OCIIntervalSetDaySecond | Modifies days, hours, minutes, seconds and fractional seconds in an interval. |
| OCIIntervalGetYearMonth | Extracts year and month values from an interval. |
| OCIIntervalSetYearMonth | Modifies year and month values in an interval. |
| OCIIntervalDivide | Implements division of OCIInterval values by OCINumber values. |
| OCIIntervalMultiply | Implements multiplication of OCIInterval values by OCINumber values. |
| OCIIntervalSubtract | Subtracts one interval value from another interval value. |
| OCIIntervalToText | Extrapolates a character string from an interval. |
| OCIIntervalCheck | Verifies the validity of an interval value. |
| OCIIntervalToNumber | Converts an OCIInterval value into a OCINumber value. |
| OCIIntervalFromNumber | Converts a OCINumber value into an OCIInterval value. |
| OCIDateTimeIntervalAdd | Adds an OCIInterval value to an OCIDatetime value, resulting in an OCIDatetime value. |
| OCIDateTimeIntervalSub | Subtracts an OCIInterval value from an OCIDatetime value, resulting in an OCIDatetime value. |
| OCIIntervalFromText | Converts a text string into an interval. |
| OCIIntervalFromTZ | Converts a time zone specification into an interval value. |

### Number Functions

Table 9‑7‑9 Number Functions

| Function | Description |
| --- | --- |
| OCINumberAbs | Compute the absolute value. |
| OCINumberAdd | Adds NUMBERs. |
| OCINumberArcCos | Compute the arc cosine. |
| OCINumberArcSin | Compute the arc sine. |
| OCINumberArcTan | Compute the arc tangent. |
| OCINumberArcTan2 | Compute the arc tangent of two NUMBERs. |
| OCINumberAssign | Assign one NUMBER to another. |
| OCINumberCeil | Compute the ceiling of NUMBER. |
| OCINumberCmp | Compare NUMBERs. |
| OCINumberCos | Compute the cosine. |
| OCINumberDec | Decrement a NUMBER. |
| OCINumberDiv | Divide two NUMBERs. |
| OCINumberExp | Raise e to the specified NUMBER power. |
| OCINumberFloor | Compute the floor of a NUMBER. |
| OCINumberFromInt | Convert an integer to an Oracle NUMBER. |
| OCINumberFromReal | Convert a real to an Oracle NUMBER. |
| OCINumberFromText | Convert a string to an Oracle NUMBER. |
| OCINumberHypCos | Compute the hyperbolic cosine. |
| OCINumberHypSin | Compute the hyperbolic sine. |
| OCINumberHypTan | Compute the hyperbolic tangent. |
| OCINumberInc | Increments a NUMBER. |
| OCINumberIntPower | Raise a given base to an integer power. |
| OCINumberIsInt | Test if a NUMBER is an integer. |
| OCINumberIsZero | Test if a NUMBER is zero. |
| OCINumberLn | Compute the natural logarithm. |
| OCINumberLog | Compute the logarithm to an arbitrary base. |
| OCINumberMod | Modulo division. |
| OCINumberMul | Multiply NUMBERs. |
| OCINumberNeg | Negate a NUMBER. |
| OCINumberPower | Exponentiation to base e. |
| OCINumberPrec | Round a NUMBER to a specified number of decimal places. |
| OCINumberRound | Round a NUMBER to a specified decimal place. |
| OCINumberSetPi | Initialize a NUMBER to Pi. |
| OCINumberSetZero | Initialize a NUMBER to zero. |
| OCINumberShift | Multiply by 10, shifting specified number of decimal places. |
| OCINumberSign | Obtain the sign of a NUMBER. |
| OCINumberSin | Compute the sine. |
| OCINumberSqrt | Compute the square root of a NUMBER. |
| OCINumberSub | Subtract NUMBERs. |
| OCINumberTan | Compute the tangent. |
| OCINumberToInt | Convert a NUMBER to an integer. |
| OCINumberToReal | Convert a NUMBER to a real. |
| OCINumberToRealArray | Convert an array of NUMBER to a real array. |
| OCINumberToText | Converts a NUMBER to a string. |
| OCINumberTrunc | Truncate a NUMBER at a specified decimal place. |

### String Functions

Table 9‑7‑10 String Functions

| Function | Description |
| --- | --- |
| OCIStringAllocSize | Get allocated size of string memory in bytes. |
| OCIStringAssign | Assign string to a string. |
| OCIStringAssignText | Assign text string to a string. |
| OCIStringPtr | Get string pointer. |
| OCIStringResize | Resize string memory. |
| OCIStringSize | Get string size. |

### Cartridge Services and File I/O Interface Functions

Table 9‑7‑11 Cartridge Services and File I/O Interface Functions

| Function | Description |
| --- | --- |
| OCIFileClose | Close an open file. |
| OCIFileExists | Test to see if the file exists. |
| OCIFileFlush | Write buffered data to a file. |
| OCIFileGetLength | Get the length of a file. |
| OCIFileInit | Initialize the OCIFile package. |
| OCIFileOpen | Open a file. |
| OCIFileRead | Read from a file into a buffer. |
| OCIFileSeek | Change the current position in a file. |
| OCIFileTerm | Terminate the OCIFile package. |
| OCIFileWrite | Write buflen bytes into the file. |

### LOB Functions

Table 9‑7‑11 LOB Functions

| Function | Description |
| --- | --- |
| OCILobRead | Returns a LOB value (or a portion of a LOB value). |
| OCILOBWriteAppend | Adds data to a LOB value. |
| OCILobGetLength | Returns the length of a LOB value. |
| OCILobTrim | Trims data from the end of a LOB value. |
| OCILobOpen | Opens a LOB value for use by other LOB functions. |
| OCILobClose | Closes a LOB value. |

### Miscellaneous Functions

Table 9‑7‑12 Miscellaneous Functions

| Function | Description |
| --- | --- |
| OCIClientVersion | Return client library version. |
| OCIErrorGet | Return error message. |
| OCIPGErrorGet | Return native error messages reported by libpq or the server. The signature is:  sword OCIPGErrorGet(dvoid \*hndlp, ub4 recordno, OraText \*errcodep,ub4 errbufsiz, OraText \*bufp, ub4 bufsiz, ub4 type) |
| OCIPasswordChange | Change password. |
| OCIPing | Confirm that the connection and server are active. |
| OCIServerVersion | Get the Oracle version string. |

### Supported Data Types

Table 9‑7‑13 Supported Data Types

|  |  |  |
| --- | --- | --- |
| Function | Description | |
| ANSI\_DATE | ANSI date |
| SQLT\_AFC | ANSI fixed character |
| SQLT\_AVC | ANSI variable character |
| SQLT\_BDOUBLE | Binary double |
| SQLT\_BIN | Binary data |
| SQLT\_BFLOAT | Binary float |
| SQLT\_CHR | Character string |
| SQLT\_DAT | Oracle date |
| SQLT\_DATE | ANSI date |
| SQLT\_FLT | Float |
| SQLT\_INT | Integer |
| SQLT\_LBI | Long binary |
| SQLT\_LNG | Long |
| SQLT\_LVB | Longer long binary |
| SQLT\_LVC | Longer longs (character) |
| SQLT\_NUM | Oracle numeric |
| SQLT\_ODT | OCI date type |
| SQLT\_STR | Zero-terminated string |
| SQLT\_TIMESTAMP | Timestamp |
| SQLT\_TIMESTAMP\_TZ | Timestamp with time zone |
| SQLT\_TIMESTAMP\_LTZ | Timestamp with local time zone |
| SQLT\_UIN | Unsigned integer |
| SQLT\_VBI | VCS format binary |
| SQLT\_VCS | Variable character |
| SQLT\_VNU | Number with preceding length byte |
| SQLT\_VST | OCI string type |

## Debugger

The Debugger is a tool that gives developers and DBAs the ability to test and debug server-side programs using a graphical, dynamic environment. The types of programs that can be debugged are SPL stored procedures, functions, triggers, and packages as well as PL/pgSQL functions and triggers.

The Debugger is integrated with and invoked from the Postgres Enterprise Manager client. There are two basic ways the Debugger can be used to test programs:

* Standalone Debugging. The Debugger is used to start the program to be tested. You supply any input parameter values required by the program and you can immediately observe and step through the code of the program. Standalone debugging is the typical method used for new programs and for initial problem investigation.
* In-Context Debugging. The program to be tested is initiated by an application other than the Debugger. You first set a global breakpoint on the program to be tested. The application that makes the first call to the program encounters the global breakpoint. The application suspends execution at which point the Debugger takes control of the called program. You can then observe and step through the code of the called program as it runs within the context of the calling application. After you have completely stepped through the code of the called program in the Debugger, the suspended application resumes execution. In-context debugging is useful if it is difficult to reproduce a problem using standalone debugging due to complex interaction with the calling application.

The debugging tools and operations are the same whether using standalone or in-context debugging. The difference is in how the program to be debugged is invoked.

The following sections discuss the features and functionality of the Debugger using the standalone debugging method. The directions for starting the Debugger for in-context debugging are discussed in Section 7.5.5.3.

### Configuring the Debugger

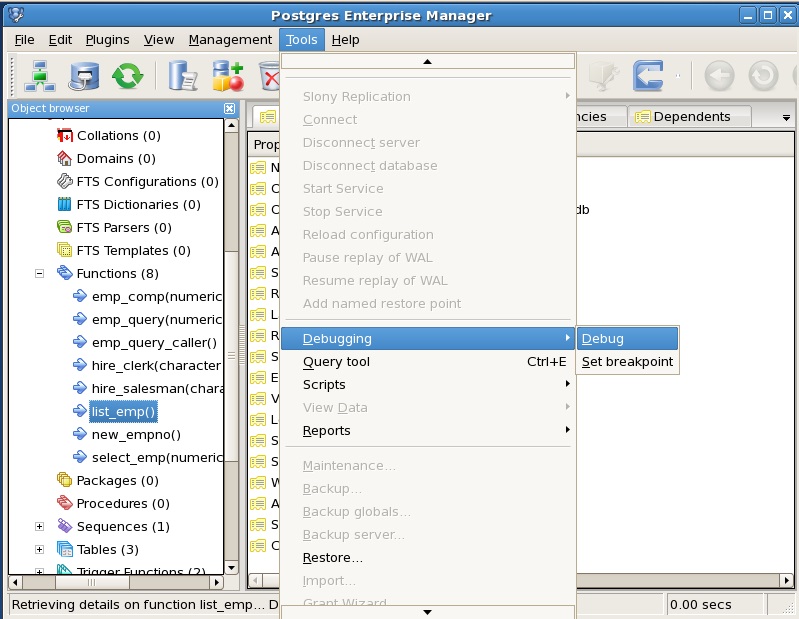
Before using the Debugger, edit the postgresql.conf file (located in the data subdirectory of your Advanced Server home directory), adding $libdir/plugin\_debugger to the libraries listed in the shared\_preload\_libraries configuration parameter:

shared\_preload\_libraries = '$libdir/dbms\_pipe,$libdir/edb\_gen,$libdir/plugin\_debugger'

After modifying the shared\_preload\_libraries parameter, you must restart the database server.

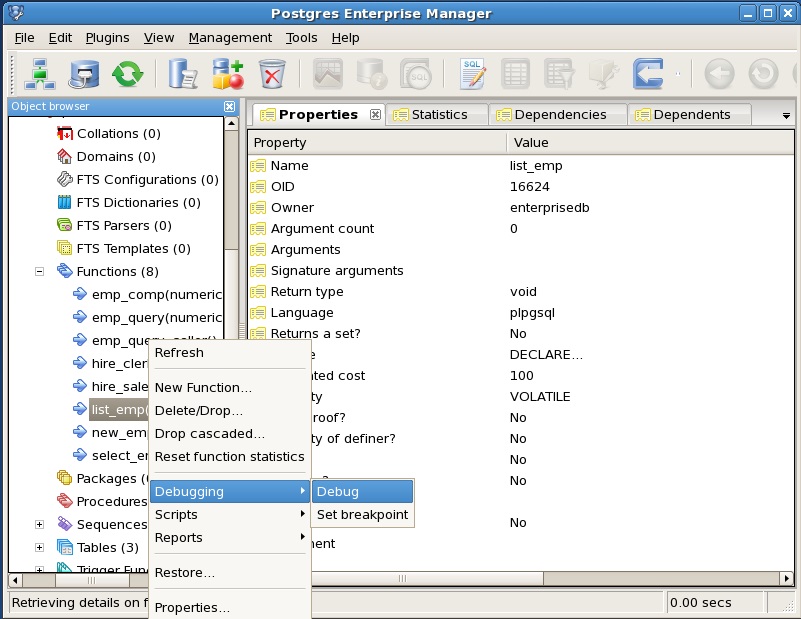
### Starting the Debugger

You can use the Postgres Enterprise Manager (PEM) client to access the Debugger for standalone debugging. To open the Debugger, highlight the name of the stored procedure or function you wish to debug in the PEM Object browser panel. Then, navigate through the Tools menu to the Debugging menu and select Debug from the submenu as shown in Figure 7.1.



*Figure 7.1 - Starting the Debugger from the Tools menu*

You can also right-click on the name of the stored procedure or function in the PEM client Object Browser, and select Debugging, and the Debug from the context menu as shown in Figure 7.2.



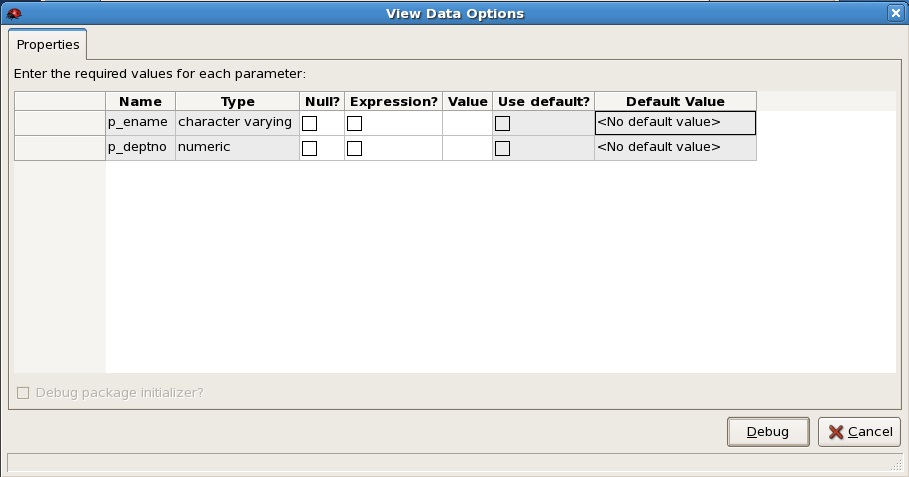
*Figure 7.2 - Starting the Debugger from the object’s context menu*

Note that triggers cannot be debugged using standalone debugging. Triggers must be debugged using in-context debugging. See Section 7.5.5.3 for information on setting a global breakpoint for in-context debugging.

To debug a package, highlight the specific procedure or function under the package node of the package you wish to debug and follow the same directions as for stored procedures and functions.

### The View Data Options Window

You can use the View Data Options window to pass parameter values when you are standalone-debugging a program that expects parameters. When you start the debugger, the View Data Options window opens automatically to display any IN or IN OUT parameters expected by the program. If the program declares no IN or IN OUT parameters, the View Data Options window does not open.



*Figure 7.3 - The View Data Options window*

Use the fields on the View Data Options window (shown in Figure 7.3) to provide a value for each parameter:

* The Name field contains the formal parameter name.
* The Type field contains the parameter data type.
* Check the Null? checkbox to indicate that the parameter is a NULL value.
* Check the Expression checkbox if the Value field contains an expression.
* The Value field contains the parameter value that will be passed to the program.
* Check the Use default? checkbox to indicate that the program should use the value in the Default Value field.
* The Default Value field contains the default value of the parameter.

Press the Enter key to select the next parameter in the list for data entry, or click on a Value field to select the parameter for data entry.

If you are debugging a procedure or function that is a member of a package that has an initialization section, check the Debug Package Initializer check box to instruct the Debugger to step into the package initialization section, allowing you to debug the initialization section code before debugging the procedure or function. If you do not select the check box, the Debugger executes the package initialization section without allowing you to see or step through the individual lines of code as they are executed.

After entering the desired parameter values, click the OK button to start the debugging process. Click the Cancel button to terminate the Debugger and return control to the PEM client.

Note: The View Data Options window does not open during in-context debugging. Instead, the application calling the program to be debugged must supply any required input parameter values.

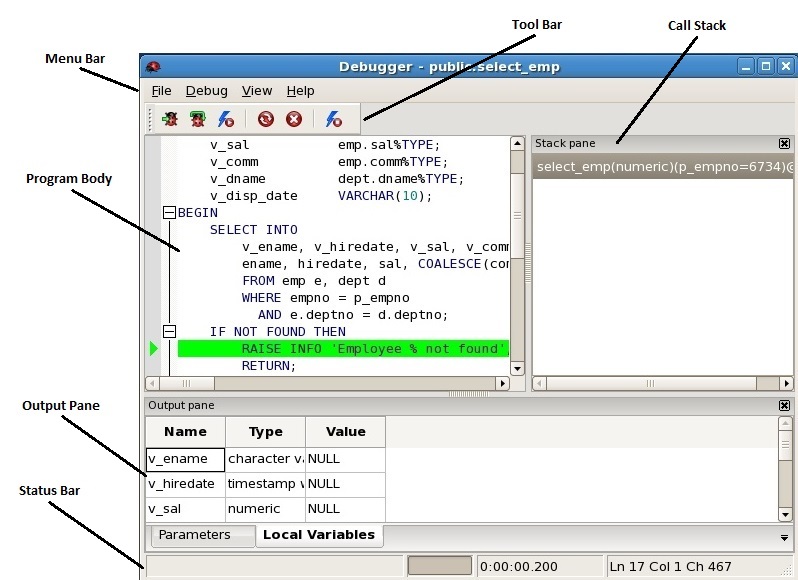
When you have completed a full debugging cycle by stepping through the program code, the View Data Options window re-opens, allowing you to enter new parameter values and repeat the debugging cycle, or end the debugging session.

### Main Debugger Window

The Main Debugger window (see Figure 7.4) contains three panes:

* the Program Body pane
* the Stack pane
* the Output pane

You can use the debugger menu bar or tool bar icons (located at the top of the debugger window) to access debugging functions.

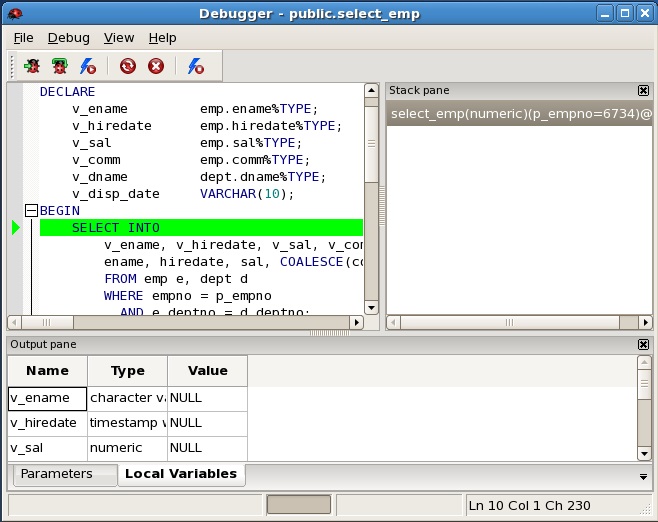


*Figure 7.4 - The Main Debugger window*

Status and error information is displayed in the status bar at the bottom of the Debugger window.

#### The Program Body Pane

The Program Body pane in the upper-left corner of the Debugger window displays the source code of the program that is being debugged.



*Figure 7.5 - The Program Body*

Figure 7.5 shows that the Debugger is about to execute the SELECT statement. The green indicator in the program body highlights the next statement to execute.

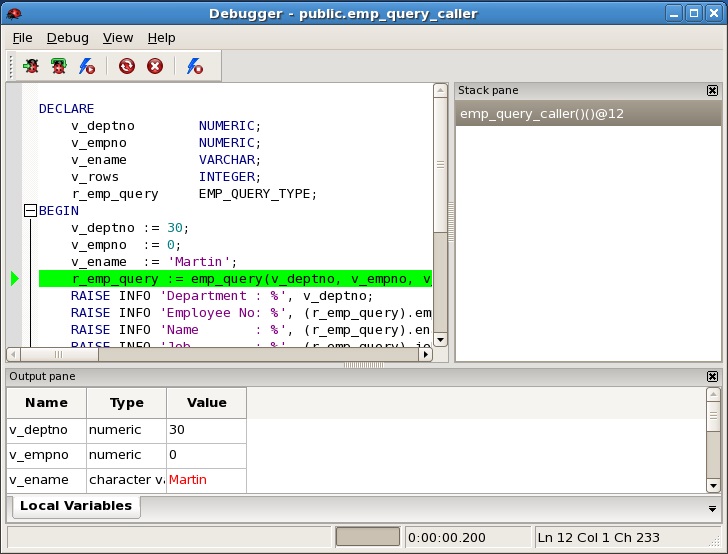
#### The Stack Pane

The Stack pane displays a list of programs that are currently on the call stack (programs that have been invoked but which have not yet completed). When a program is called, the name of the program is added to the top of the list displayed in the Stack pane; when the program ends, its name is removed from the list.

The Stack pane also displays information about program calls. The information includes:

* The location of the call within the program
* The call arguments
* The name of the program being called

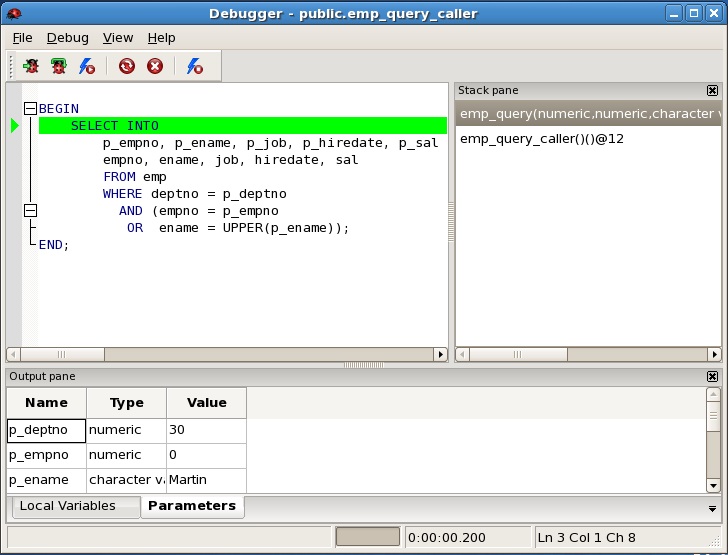
Reviewing the call stack can help you trace the course of execution through a series of nested programs.



*Figure 7.6 – A debugged program calling a subprogram*

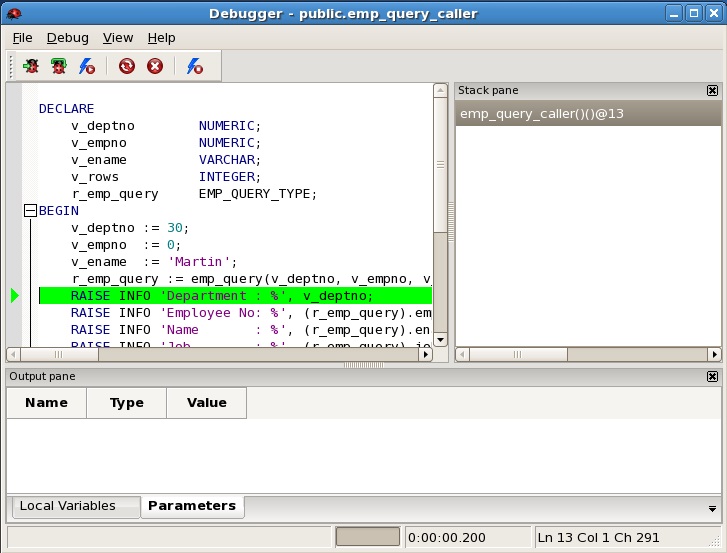
Figure 7.6 shows that emp\_query\_caller is about to call a subprogram named emp\_query. emp\_query\_caller is currently at the top of the call stack.

After the call to emp\_query executes, emp\_query is displayed at the top of the Stack pane, and its code is displayed in the Program Body frame (see Figure 7.7).



*Figure 7.7 - Debugging the called subprogram*

Upon completion of execution of the subprogram, control returns to the calling program (public.emp\_query\_caller), now displayed at the top of the Stack pane in Figure 7.8.

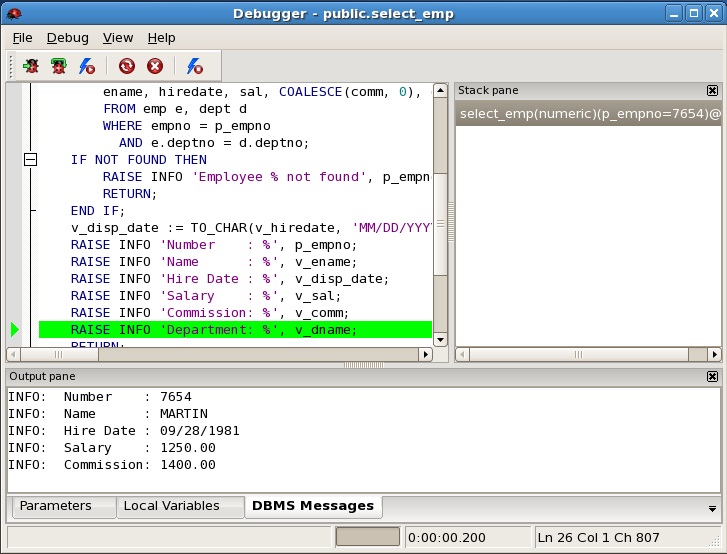


*Figure 7.8 – Control returns from debugged subprogram*

Highlight an entry in the call stack to review detailed information about the selected entry on the tabs in the Output pane. Using the call stack to navigate to another entry in the call stack will not alter the line that is currently executing.

#### The Output Pane

You can use tabs in the Output pane (see Figure 7.9) to view or modify parameter values or local variables, or to view messages generated by RAISE INFO and function results.



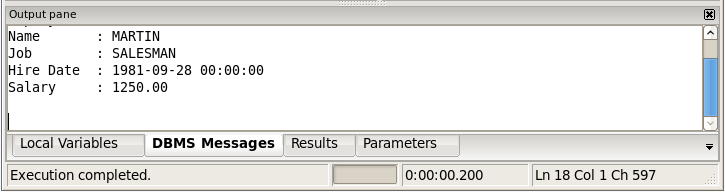
*Figure 7.9 – The DBMS Messages tab of the Output pane.*

Each tab contains a different type of information:

* The Parameters tab displays the current parameter values.
* The Local Variables tab displays the value of any variables declared within the program.
* The DBMS Messages tab displays any results returned by the program as it executes.
* The Results tab displays program results (if applicable).

#### The Status Bar

The status bar (see Figure 7.10) displays a message when the Debugger pauses, when a runtime error message is encountered, or when execution completes.



*Figure 7.10 - The Status Bar, indicating Execution completed.*

### Debugging a Program

You can perform the following operations to debug a program:

* Step through the program one line at a time
* Execute the program until you reach a breakpoint
* View and change local variable values within the program

#### Stepping Through the Code

Use the tool bar icons to step through a program with the Debugger:

Screen shot 2013-07-25 at 9 Use the Step Into icon to execute the line of code currently highlighted by the green bar in the Program Body pane, and then pause execution. If the executed code line is a call to a subprogram, the called subprogram is brought into the Program Body pane, and the first executable line of code of the subprogram is highlighted as the Debugger waits for you to perform an operation on the subprogram.

Screen shot 2013-07-25 at 9 Use the Step Over icon to execute a line of code, stepping over any subprograms invoked by that line of code. The subprogram is executed, but not debugged. If the subprogram contains a breakpoint, the debugger will stop at that breakpoint.

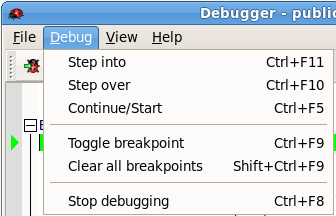
Screen shot 2013-07-25 at 9 Use the Continue icon to execute the line of code highlighted by the green bar, and continue execution until either a breakpoint is encountered or the last line of the program has been executed.

Figure 7.11 shows the locations of the Step Into, Step Over, and Continue icons on the tool bar:



*Figure 7.11 - The Step Into, Step Over, and Continue icons*

The debugging operations are also accessible through the Debug menu, as shown in Figure 7.12.



*Figure 7.12 - Debug menu options*

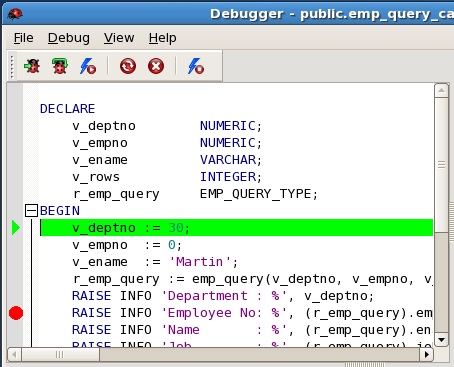
#### Using Breakpoints

As the Debugger executes a program, it pauses whenever it reaches a breakpoint. When the Debugger pauses, you can observe or change local variables, or navigate to an entry in the call stack to observe variables or set other breakpoints. The next step into, step over, or continue operation forces the debugger to resume execution with the next line of code following the breakpoint. There are two types of breakpoints:

***Local Breakpoint -*** A local breakpoint can be set at any executable line of code within a program. The Debugger pauses execution when it reaches a line where a local breakpoint has been set.

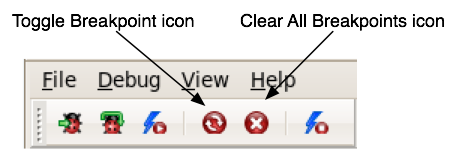
***Global Breakpoint -*** A global breakpoint will trigger when any session reaches that breakpoint. Set a global breakpoint if you want to perform in-context debugging of a program. When a global breakpoint is set on a program, the debugging session that set the global breakpoint waits until that program is invoked in another session. A global breakpoint can only be set by a superuser.

To create a local breakpoint, left-click in the grey shaded margin to the left of the line of code where you want the local breakpoint set. The Debugger displays a red dot in the margin, indicating a breakpoint has been set at the selected line of code (see Figure 7.13).



*Figure 7.13 - Set a breakpoint by clicking in left-hand margin*

You can also set a breakpoint by left-clicking in the Program Body to place your cursor, and selecting Toggle Breakpoint from Debug menu or by clicking the Toggle Breakpoint icon (see Figure 7.14). A red dot appears in the left-hand margin indicating a breakpoint has been set as the line of code.



*Figure 7.14 - The breakpoint control icons*

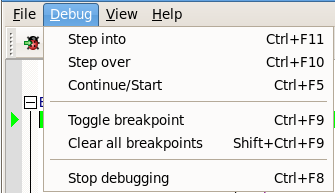
You can set as many local breakpoints as desired. Local breakpoints remain in effect for the duration of a debugging session until they are removed.

Removing a Local Breakpoint

To remove a local breakpoint, you can:

* Left click the mouse on the red breakpoint indicator in the left margin of the Program Body pane. The red dot disappears, indicating that the breakpoint has been removed.
* Use your mouse to select the location of the breakpoint in the code body, and select Toggle Breakpoint from Debug menu, or click the Toggle Breakpoint icon.

You can remove all of the breakpoints from the program that currently appears in the Program Body frame by selecting Clear all breakpoints from the Debug menu (see Figure 7.15) or by clicking the Clear All Breakpoints icon.

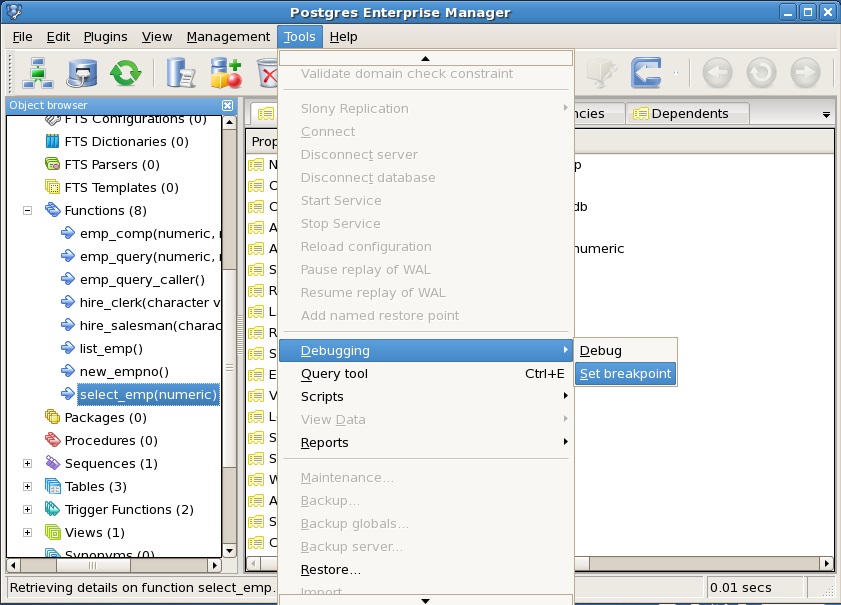


*Figure 7.15 - The breakpoint menu options*

Note: When you perform any of the preceding actions, only the breakpoints in the program that currently appears in the Program Body frame are removed. Breakpoints in called subprograms or breakpoints in programs that call the program currently appearing in the Program Body frame are not removed.

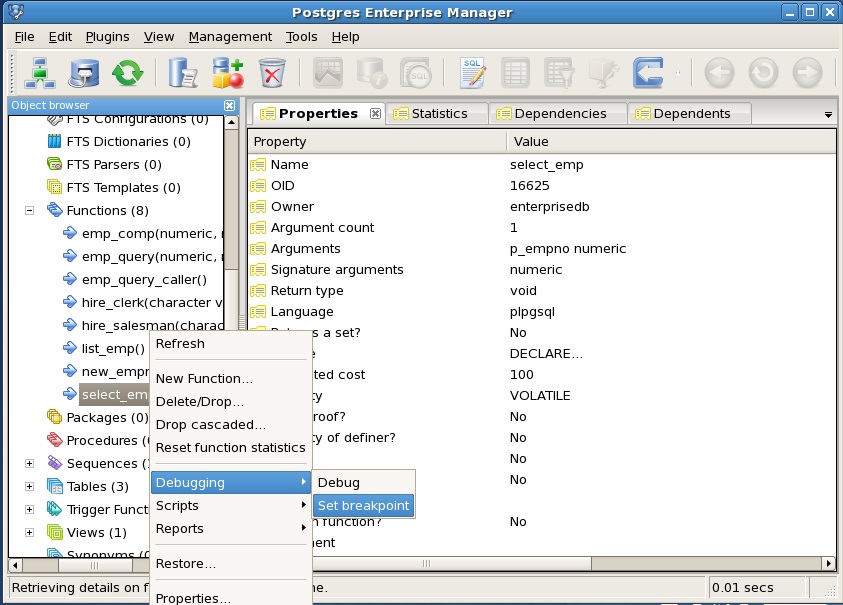
#### Setting a Global Breakpoint for In-Context Debugging

To set a global breakpoint for in-context debugging, highlight the stored procedure, function, or trigger on which you wish to set the breakpoint in the Object browser panel. Navigate through the Tools menu to select Debugging, and then Set Breakpoint (see Figure 7.16)



*Figure 7.16 - Setting a global breakpoint from the Tools menu*

Alternatively, you can right-click on the name of the stored procedure, function, or trigger on which you wish to set a global breakpoint and select Debugging, then Set Breakpoint from the context menu as shown in Figure 7.17.

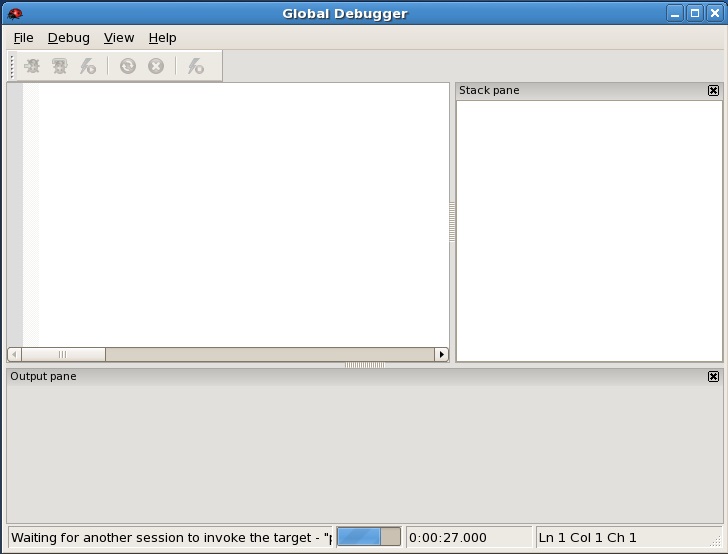
****

*Figure 7.17 - Setting a global breakpoint from the object's context menu*

To set a global breakpoint on a trigger, expand the table node that contains the trigger, highlight the specific trigger you wish to debug, and follow the same directions as for stored procedures and functions.

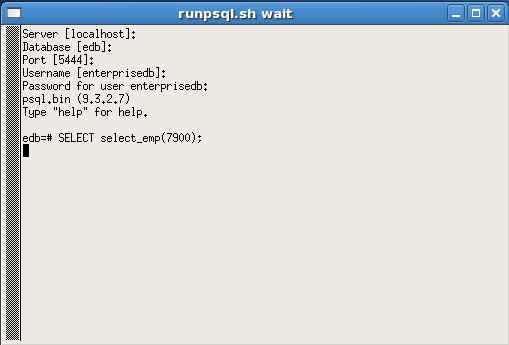
To set a global breakpoint in a package, highlight the specific procedure or function under the package node of the package you wish to debug and follow the same directions as for stored procedures and functions.

After you choose Set Breakpoint, the Debugger window opens and waits for an application to call the program to be debugged (see Figure 7.18).



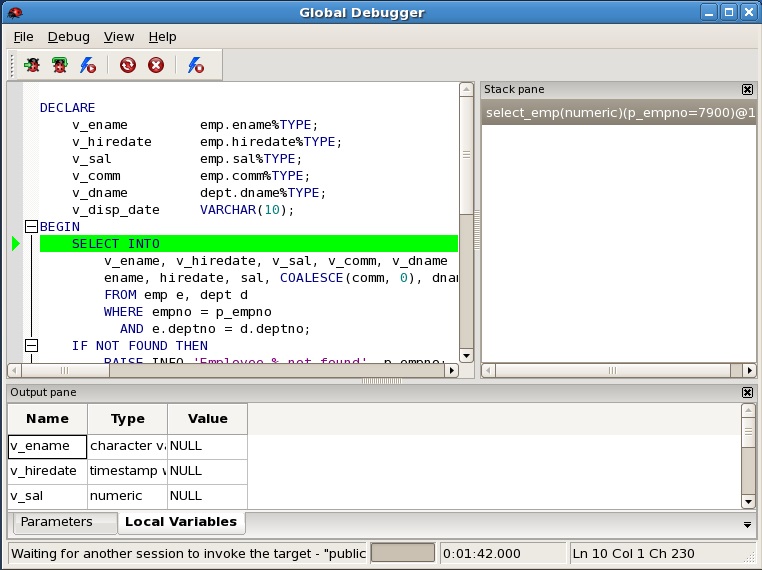
*Figure 7.18 - Waiting for invocation of program to be debugged*

In Figure 7.19, the EDB-PSQL client invokes the select\_emp function (on which a global breakpoint has been set).



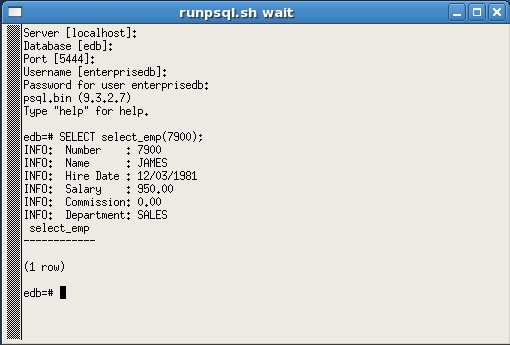
*Figure 7.19 - Application invoking program with a global breakpoint*

The select\_emp function does not complete until you step through the program in the Debugger, which now appears as shown in Figure 7.20.



*Figure 7.20 - Program on which a global breakpoint has been set*

You can now debug the program using any of the previously discussed operations such as step into, step over, and continue, or set local breakpoints. When you have stepped through execution of the program, the calling application (EDB-PSQL) regains control as shown in Figure 7.21.



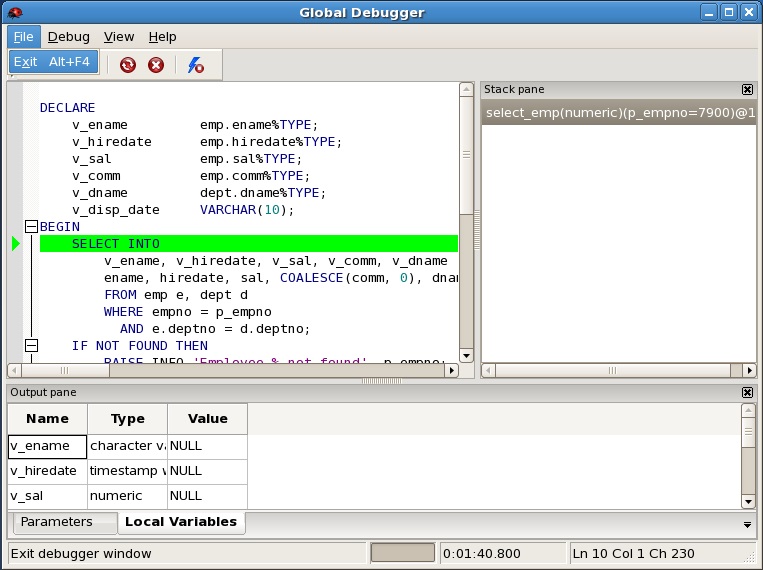
*Figure 7.21 - Application after debugging*

The select\_emp function completes execution and its output is displayed.

At this point, you can end the Debugger session by choosing Exit from the File menu. If you do not end the Debugger session, the next application that invokes the program will encounter the global breakpoint and the debugging cycle will begin again.

#### Exiting the Debugger

To end a Debugger session and exit the Debugger, select Exit from File menu or press Alt-F4 as shown by the following:



*Figure 7.22 - Exiting from the Debugger*

# Performance Analysis and Tuning

Advanced Server provides various tools for performance analysis and tuning. These features are described in this chapter.

## Dynatune

Advanced Server supports dynamic tuning of the database server to make the optimal usage of the system resources available on the host machine on which it is installed. The two parameters that control this functionality are located in the postgresql.conf file. These parameters are:

* edb\_dynatune
* edb\_dynatune\_profile

### edb\_dynatune

edb\_dynatune determines how much of the host system's resources are to be used by the database server based upon the host machine's total available resources and the intended usage of the host machine.

When Advanced Server is initially installed, the edb\_dynatune parameter is set in accordance with the selected usage of the host machine on which it was installed - i.e., development machine, mixed use machine, or dedicated server. For most purposes, there is no need for the database administrator to adjust the various configuration parameters in the postgresql.conf file in order to improve performance.

You can change the value of the edb\_dynatune parameter after the initial installation of Advanced Server by editing the postgresql.conf file. The postmaster must be restarted in order for the new configuration to take effect.

The edb\_dynatune parameter can be set to any integer value between 0 and 100, inclusive. A value of 0, turns off the dynamic tuning feature thereby leaving the database server resource usage totally under the control of the other configuration parameters in the postgresql.conf file.

A low non-zero, value (e.g., 1 - 33) dedicates the least amount of the host machine's resources to the database server. This setting would be used for a development machine where many other applications are being used.

A value in the range of 34 - 66 dedicates a moderate amount of resources to the database server. This setting might be used for a dedicated application server that may have a fixed number of other applications running on the same machine as Advanced Server.

The highest values (e.g., 67 - 100) dedicate most of the server's resources to the database server. This setting would be used for a host machine that is totally dedicated to running Advanced Server.

Once a value of edb\_dynatune is selected, database server performance can be further fine-tuned by adjusting the other configuration parameters in the postgresql.conf file. Any adjusted setting overrides the corresponding value chosen by edb\_dynatune. You can change the value of a parameter by un-commenting the configuration parameter, specifying the desired value, and restarting the database server.

### edb\_dynatune\_profile

The edb\_dynatune\_profile parameter is used to control tuning aspects based upon the expected workload profile on the database server. This parameter takes effect upon startup of the database server.

The possible values for edb\_dynatune\_profile are:

|  |  |
| --- | --- |
| Value | Usage |
| oltp | Recommended when the database server is processing heavy online transaction processing workloads. |
| reporting | Recommended for database servers used for heavy data reporting. |
| mixed | Recommended for servers that provide a mix of transaction processing and data reporting. |

## Infinite Cache

Note: Infinite Cache has been deprecated and may be removed in a future release. Please contact your EnterpriseDB Account Manager or <mailto:sales@enterprisedb.com> for more information.

Database performance is typically governed by two competing factors:

* Memory access is fast; disk access is slow.
* Memory space is scarce; disk space is abundant.

Advanced Server tries very hard to minimize disk I/O by keeping frequently used data in memory. When the first server process starts, it creates an in-memory data structure known as the buffer cache. The buffer cache is organized as a collection of 8K (8192 byte) pages: each page in the buffer cache corresponds to a page in some table or index. The buffer cache is shared between all processes servicing a given database.

When you select a row from a table, Advanced Server reads the page that contains the row into the shared buffer cache. If there isn't enough free space in the cache, Advanced Server evicts some other page from the cache. If Advanced Server evicts a page that has been modified, that data is written back out to disk; otherwise, it is simply discarded. Index pages are cached in the shared buffer cache as well.

Figure 1.1 demonstrates the flow of data in a typical Advanced Server session:

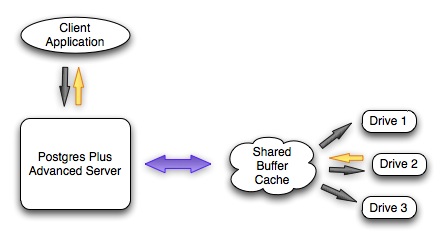


Figure 1.1 – Data Flow

A client application sends a query to the Postgres server and the server searches the shared buffer cache for the required data. If the requested data is found in the cache, the server immediately sends the data back to the client. If not, the server reads the page that holds the data into the shared buffer cache, evicting one or more pages if necessary. If the server decides to evict a page that has been modified, that page is written to disk.

As you can see, a query will execute much faster if the required data is found in the shared buffer cache.

One way to improve performance is to increase the amount of memory that you can devote to the shared buffer cache. However, most computers impose a strict limit on the amount of RAM that you can install. To help circumvent this limit, Infinite Cache lets you utilize memory from other computers connected to your network.

With Infinite Cache properly configured, Advanced Server will dedicate a portion of the memory installed on each cache server as a secondary memory cache. When a client application sends a query to the server, the server first searches the shared buffer cache for the required data; if the requested data is not found in the cache, the server searches for the necessary page in one of the cache servers.

Figure 1.2 shows the flow of data in an Advanced Server session with Infinite Cache:

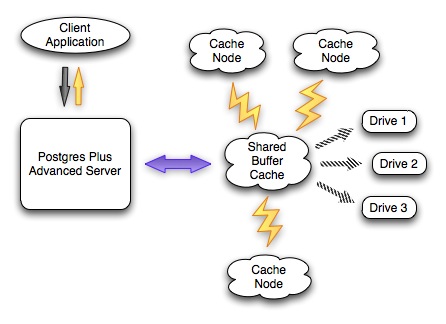


Figure 1.2 – Data flow with Infinite Cache

When a client application sends a query to the server, the server searches the shared buffer cache for the required data. If the requested data is found in the cache, the server immediately sends the data back to the client. If not, the server sends a request for the page to a specific cache server; if the cache server holds a copy of the page it sends the data back to the server and the server copies the page into the shared buffer cache. If the required page is not found in the primary cache (the shared buffer cache) or in the secondary cache (the cloud of cache servers), Advanced Server must read the page from disk. Infinite Cache improves performance by utilizing RAM from other computers on your network in order to avoid reading frequently accessed data from disk.

Updating the Cache Node Configuration

You can add or remove cache servers without restarting the database server by adding or deleting cache nodes from the list defined in the edb\_icache\_servers configuration parameter. For more information about changing the configuration parameter, see Section 8.2.2.2.

When you add one or more cache nodes, the server re-allocates the cache, dividing the cache evenly amongst the servers; each of the existing cache servers loses a percentage of the information that they have cached. You can calculate the percentage of the cache that remains valid with the following formula:

(existing\_nodes \* 100) / (existing\_nodes + new\_nodes)

For example, if an Advanced Server installation with three existing cache nodes adds an additional cache node, 75% of the existing cache remains valid after the reconfiguration.

If cache nodes are removed from a server, the data that has been stored on the remaining cache nodes is preserved. If one cache server is removed from a set of five cache servers, Advanced Server preserves the 80% of the distributed cache that is stored on the four remaining cache nodes.

When you change the cache server configuration (by adding or removing cache servers), the portion of the cache configuration that is preserved is not re-written unless the cache is completely re-warmed using the edb\_icache\_warm() function or edb\_icache\_warm utility. If you do not re-warm the cache servers, new cache servers will accrue cache data as queries are performed on the server.

Infinite Cache Offers a Second Performance Advantage: Compression.

Without Infinite Cache, Advanced Server will read each page from disk as an 8K chunk; when a page resides in the shared buffer cache, it consumes 8K of RAM. With Infinite Cache, Postgres can compress each page before sending it to a cache server. A compressed page can take significantly less room in the secondary cache, making more space available for other data and effectively increasing the size of the cache. A compressed page consumes less network bandwidth as well, decreasing the amount of time required to retrieve a page from the secondary cache.

The fact that Infinite Cache can compress each page may make it attractive to configure a secondary cache server on the same computer that runs your Postgres server. If, for example, your computer is configured with 6GB of RAM, you may want to allocate a smaller amount (say 1GB) for the primary cache (the shared buffer cache) and a larger amount (4GB) to the secondary cache (Infinite Cache), reserving 1GB for the operating system. Since the secondary cache resides on the same computer, there is very little overhead involved in moving data between the primary and secondary cache. All data stored in the Infinite Cache is compressed so the secondary cache can hold many more pages than would fit into the (uncompressed) shared buffer cache. If you had allocated 5GB to the shared buffer cache, the cache could hold no more than 65000 pages (approximately). By assigning 4GB of memory to Infinite Cache, the cache may be able to hold 130000 pages (at 2x compression), 195000 pages (at 3x compression) or more. The compression factor that you achieve is determined by the amount of redundancy in the data itself and the edb\_icache\_compression\_level parameter.

To use Infinite Cache, you must specify a list of one or more cache servers (computers on your network) and start the edb\_icache daemon on each of those servers.

Infinite Cache is supported on Linux, HPUX and Solaris systems only.

Please Note: Infinite Cache and the effective\_io\_concurrency parameter can potentially interfere with each other. You should disable asynchronous I/O requests (by setting the value of effective\_io\_concurrency to 0 in the postgresql.conf file) if you enable the Infinite Cache feature.

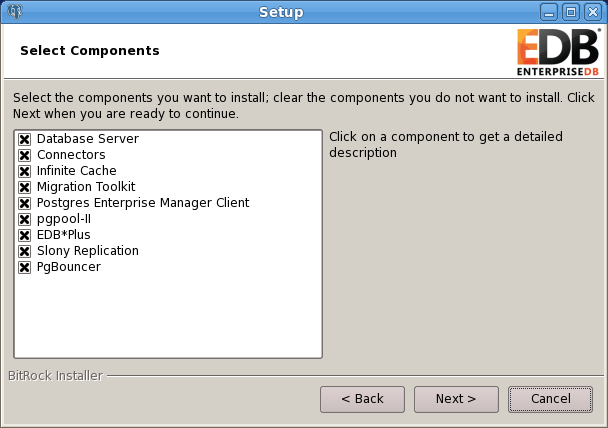
### Installing Infinite Cache

Advanced Server includes Infinite Cache functionality as part of a standard installation with either the graphical installer or the RPM installer. You can also optionally install only the Infinite Cache daemon on a supporting cache server.

For information about using the RPM packages to install Infinite Cache, please see the EDB Postgres Advanced Server Installation Guide available at:

<http://www.enterprisedb.com/products-services-training/products/documentation/enterpriseedition>

To use the graphical installer to install Advanced Server with Infinite Cache functionality, confirm that the box next to the Database Server option (located on the Setup: Select Components window, shown in Figure 8.3) is selected when running the installation wizard.

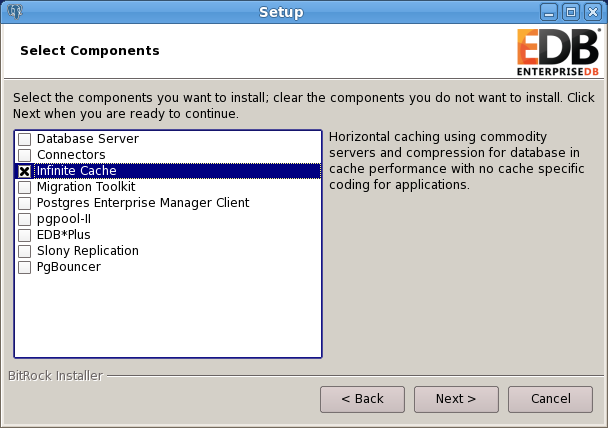


*Figure 8.3: The Setup: Select Components window.*

The Database Server option installs the following Infinite Cache components:

* The ppas-infinitecache service script.
* The Infinite Cache configuration file (ppas-infinitecache).
* A command line tool that allows you to pre-load the cache servers (edb-icache-warm).
* The edb\_icache libraries (code libraries required by the edb-icache daemon).

The graphical installation wizard can selectively install only the Infinite Cache daemon on a cache server. To install the edb-icache daemon on a cache server, deploy the installation wizard on the machine hosting the cache; when the Setup: Select Components window opens, de-select all options except Infinite Cache (as shown in Figure 8.4).



*Figure 8.4: Installing only the Infinite Cache Daemon.*

The Infinite Cache Daemon option installs the following:

* The ppas-infinitecache service script.
* The Infinite Cache configuration file (ppas-infinitecache).
* A command line tool that allows you to pre-load the cache servers (edb-icache-warm).
* The edb\_icache libraries (code libraries required by the edb-icache daemon).

### Configuring the Infinite Cache Server

Configuring Infinite Cache is a three-step process:

* Specify Infinite Cache server settings in the Infinite Cache configuration file.
* Modify the Advanced Server postgresql.conf file, enabling Infinite Cache, and specifying connection and compression settings.
* Start the Infinite Cache service.

#### Modifying Infinite Cache Settings

The Infinite Cache configuration file is named ppas-infinitecache, and contains two parameters and their associated values:

PORT=11211

CACHESIZE=500

To modify a parameter, open the ppas-infinitecache file (located in the /opt/PostgresPlus/infinitecache/etc directory) with your editor of choice, and modify the parameter values:

PORT

Use the PORT variable to specify the port where Infinite Cache will listen for connections from Advanced Server.

CACHESIZE

Use the CACHESIZE variable to specify the size of the cache (in MB).

#### Enabling Infinite Cache

The postgresql.conf file includes three configuration parameters that control the behavior of Infinite Cache. The postgresql.conf file is read each time you start the Advanced Server database server. To modify a parameter, open the postgresql.conf file (located in the $PGDATA directory) with your editor of choice, and edit the section of the configuration file shown below:

# - Infinite Cache

#edb\_enable\_icache = off

#edb\_icache\_servers = '' #'host1:port1,host2,ip3:port3,ip4'

#edb\_icache\_compression\_level = 6

Lines that begin with a pound sign (#) are treated as comments; to enable a given parameter, remove the pound sign and specify a value for the parameter. When you've updated and saved the configuration file, restart the database server for the changes to take effect.

edb\_enable\_icache

Use the edb\_enable\_icache parameter to enable or disable Infinite Cache. When edb\_enable\_icache is set to on, Infinite Cache is enabled; if the parameter is set to off, Infinite Cache is disabled.

If you set edb\_enable\_icache to on, you must also specify a list of cache servers by setting the edb\_icache\_servers parameter (described in the next section).

The default value of edb\_enable\_icache is off.

edb\_icache\_servers

The edb\_icache\_servers parameter specifies a list of one or more servers with active edb-icache daemons. edb\_icache\_servers is a string value that takes the form of a comma-separated list of *hostname:port* pairs. You can specify each pair in any of the following forms:

* *hostname*
* *IP-address*
* *hostname:portnumber*
* *IP-address:portnumber*

If you do not specify a port number, Infinite Cache assumes that the cache server is listening at port 11211. This configuration parameter will take effect only if edb\_enable\_icache is set to on. Use the edb\_icache\_servers parameter to specify a maximum of 128 cache nodes.

edb\_icache\_compression\_level

The edb\_icache\_compression\_level parameter controls the compression level that is applied to each page before storing it in the distributed Infinite Cache. This parameter must be an integer in the range 0 to 9.

* A compression level of 0 disables compression; it uses no CPU time for compression, but requires more storage space and network resources to process.
* A compression level of 9 invokes the maximum amount of compression; it increases the load on the CPU, but less data flows across the network, so network demand is reduced. Each page takes less room on the Infinite Cache server, so memory requirements are reduced.
* A compression level of 5 or 6 is a reasonable compromise between the amount of compression received and the amount of CPU time invested.

By default, edb\_icache\_compression\_level is set to 6.

When Advanced Server reads data from disk, it typically reads the data in 8K increments. If edb\_icache\_compression\_level is set to 0, each time Advanced Server sends an 8K page to the Infinite Cache server that page is stored (uncompressed) in 8K of cache memory. If the edb\_icache\_compression\_level parameter is set to 9, Advanced Server applies the maximum compression possible before sending it to the Infinite Cache server, so a page that previously took 8K of cached memory might take 2K of cached memory. Exact compression numbers are difficult to predict, as they are dependent on the nature of the data on each page.

The compression level must be set by the superuser and can be changed for the current session while the server is running. The following command disables the compression mechanism for the currently active session:

SET edb\_icache\_compression\_level = 0

The following example shows a typical collection of Infinite Cache settings:

edb\_enable\_icache = on

edb\_icache\_servers = 'localhost,192.168.2.1:11200,192.168.2.2'

edb\_icache\_compression\_level = 6

Please Note: Infinite Cache and the effective\_io\_concurrency parameter can potentially interfere with each other. You should disable asynchronous I/O requests (by setting the value of effective\_io\_concurrency to 0 in the postgresql.conf file) if you enable the Infinite Cache feature. By default, effective\_io\_concurrency is set to 1.

#### Controlling the Infinite Cache Server

Linux

On Linux, the Infinite Cache service script is named ppas-infinitecache. The service script resides in the /etc/init.d directory. You can control the Infinite Cache service, or check the status of the service with the following command:

/etc/init.d/ppas-infinitecache *action*

Where *action* specifies:

* start to start the service.
* stop to stop the service
* restart to stop and then start the service.
* status to return the status of the service.

### Dynamically Modifying Infinite Cache Server Nodes

You can dynamically modify the Infinite Cache server nodes; to change the Infinite Cache server configuration, use the edb\_icache\_servers parameter in the postgresql.conf file to:

* specify additional cache information to add a server/s.
* delete server information to remove a server/s.
* specify additional server information and delete existing server information to both add and delete servers during the same reload operation.

After updating the edb\_icache\_servers parameter in the postgresql.conf file, you must reload the configuration parameters for the changes to take effect.

To reload the configuration parameters, navigate through the Postgres Plus Advanced Server 9.4 menu to the Expert Configuration menu, and select the Reload Configuration option. If prompted, enter your password to reload the configuration parameters.

Alternatively, you can use the pg\_ctl reload command to update the server's configuration parameters at the command line:

pg\_ctl reload -D *data*\_*directory*

Where *data*\_*directory* specifies the complete path to the data directory.

Please Note: If the server detects a problem with the value specified for the edb\_icache\_servers parameter during a server reload, it will ignore changes to the parameter and use the last valid parameter value. If you are performing a server restart, and the parameter contains an invalid value, the server will return an error.

### Controlling the edb-icache Daemons

edb-icache is a high-performance memory caching daemon that distributes and stores data in shared buffers. The server transparently interacts with edb-icache daemons to store and retrieve data.

Before starting the database server, the edb-icache daemons must be running on each server node. Log into each server and start the edb-icache server (on that host) by issuing the following command:

# edb-icache -u enterprisedb -d -m 1024

Where:

-u

-u specifies the user name

-m

-m specifies the amount of memory to be used by edb-icache. The default is 64MB.

-d

-d designates that the service should run in the background

To gracefully kill an edb-icache daemon (close any in-use files, flush buffers, and exit), execute the command:

# killall -TERM edb-icache

If the edb-icache daemon refuses to die, you may need to use the following command:

# killall -KILL edb-icache

#### Command Line Options

To view the command line options for the edb-icache daemon, use the following command from the edb\_Infinite Cache subdirectory, located in the Advanced Server installation directory:

# edb-icache -h

The command line options are:

|  |  |
| --- | --- |
| Parameter | Description |
| -p <port\_number> | The TCP port number the Infinite Cache daemon is listening on. The default is 11211. |
| -U <UDP\_number> | The UDP port number the Infinite Cache daemon is listening on. The default is 0 (off). |
| -s <pathname> | The Unix socket pathname the Infinite Cache daemon is listening on. If included, the server limits access to the host on which the Infinite Cache daemon is running, and disables network support for Infinite Cache. |
| -a <mask> | The access mask for the Unix socket, in octal form. The default value is 0700. |
| -l <ip\_addr> | Specifies the IP address that the daemon is listening on. If an individual address is not specified, the default value is INDRR\_ANY; all IP addresses assigned to the resource are available to the daemon. |
| -d | Run as a daemon. |
| -r | Maximize core file limit. |
| -u <username> | Assume the identity of the specified user (when run as root). |
| -m <numeric> | Max memory to use for items in megabytes. Default is 64 MB. |
| -M | Return error on memory exhausted (rather than removing items). |
| -c <numeric> | Max simultaneous connections. Default is 1024. |
| -k | Lock down all paged memory. Note that there is a limit on how much memory you may lock. Trying to allocate more than that would fail, so be sure you set the limit correctly for the user you started the daemon with (not for -u <username> user; under sh this is done with 'ulimit -S -l NUM\_KB'). |
| -v | Verbose (print errors/warnings while in event loop). |
| -vv | Very verbose (include client commands and responses). |
| -vvv | Extremely verbose (also print internal state transitions). |
| -h | Print the help text and exit. |
| -i | Print memcached and libevent licenses. |
| -P <file> | Save PID in <file>, only used with -d option. |
| -f <factor> | Chunk size growth factor. Default value is 1.25. |
| -n <bytes> | Minimum space allocated for key+value+flags. Default is 48. |
| -L | Use large memory pages (if available). Increasing the memory page size could reduce the number of transition look-aside buffer misses and improve the performance. To get large pages from the OS, Infinite Cache will allocate the total item-cache in one large chunk. |
| -D <char> | Use <char> as the delimiter between key prefixes and IDs. This is used for per-prefix stats reporting. The default is":" (colon).  If this option is specified, stats collection is enabled automatically; if not, then it may be enabled by sending the stats detail on command to the server. |
| -t <num> | Specifies the number of threads to use. Default is 4. |
| -R | Maximum number of requests per event; this parameter limits the number of requests process for a given connection to prevent starvation, default is 20. |
| -C | Disable use of CAS (check and set). |
| -b | Specifies the backlog queue limit, default is 1024. |
| -B | Specifies the binding protocol. Possible values are ascii, binary or auto; default value is auto. |
| -I | Override the size of each slab page. Specifies the max item size; default 1 MB, minimum size is 1 k, maximum is 128 MB). |

#### edb-icache-tool

edb-icache-tool provides a command line interface that queries the edb-icache daemon to retrieve statistical information about a specific cache node. The syntax is:

edb-icache-tool <*host*[:*port*]> stats

host specifies the address of the host that you are querying.

port specifies the port that the daemon is listening on.

edb-icache-tool retrieves the statistics described in the following table:

|  |  |
| --- | --- |
| Statistic | Description |
| accepting\_conns | Will this server accept new connection(s)? 1 if yes, otherwise 0. |
| auth\_cmds | Number of authentication commands handled by this server, success or failure. |
| auth\_errors | Number of failed authentications. |
| bytes | Total number of bytes in use. |
| bytes\_read | Total number of bytes received by this server (from the network). |
| bytes\_written | Total number of bytes sent by this server (to the network). |
| cas\_badval | Number of keys that have been compared and swapped by this server but the comparison (original) value did not match the supplied value. |
| cas\_hits | Number of keys that have been compared and swapped by this server and found present. |
| cas\_misses | Number of keys that have been compared and swapped by this server and not found. |
| cmd\_flush | Cumulative number of flush requests sent to this server. |
| cmd\_get | Cumulative number of read requests sent to this server. |
| cmd\_set | Cumulative number of write requests sent to this server. |
| conn\_yields | Number of times any connection yielded to another due to hitting the edb-icache -R limit. |
| connection\_structures | Number of connection structures allocated by the server. |
| curr\_connections | Number of open connections. |
| curr\_items | Number of items currently stored by the server. |
| decr\_hits | Number of decrement requests satisfied by this server. |
| decr\_misses | Number of decrement requests not satisfied by this server. |
| delete\_hits | Number of delete requests satisfied by this server. |
| delete\_misses | Number of delete requests not satisfied by this server. |
| evictions | Number of valid items removed from cache to free memory for new items. |
| get\_hits | Number of read requests satisfied by this server. |
| get\_misses | Number of read requests not satisfied by this server. |
| incr\_hits | Number of increment requests satisfied by this server. |
| incr\_misses | Number of increment requests not satisfied by this server. |
| limit\_maxbytes | Number of bytes allocated on this server for storage. |
| listen\_disabled\_num | Cumulative number of times this server has hit its connection limit. |
| pid | Process ID (on cache server). |
| pointer\_size | Default pointer size on host OS (usually 32 or 64). |
| reclaimed | Number of times an entry was stored using memory from an expired entry. |
| rusage\_user | Accumulated user time for this process (seconds.microseconds). |
| rusage\_system | Accumulated system time for this process (seconds.microseconds). |
| threads | Number of worker threads requested. |
| total\_time | Number of seconds since this server's base date (usually midnight, January 1, 1970, UTC). |
| total\_connections | Total number of connections opened since the server started running. |
| total\_items | Total number of items stored by this server (cumulative). |
| uptime | Amount of time that server has been active. |
| version | edb-icache version. |

In the following example, edb-icache-tool retrieves statistical information about an Infinite Cache server located at the address, 192.168.23.85 and listening on port 11213:

# edb-icache-tool 192.168.23.85:11213 stats

Field Value

accepting\_conns 1

auth\_cmds 0

auth\_errors 0

bytes 52901223

bytes\_read 188383848

bytes\_written 60510385

cas\_badval 0

cas\_hits 0

cas\_misses 0

cmd\_flush 1

cmd\_get 53139

cmd\_set 229120

conn\_yields 0

connection\_structures 34

curr\_connections 13

curr\_items 54953

decr\_hits 0

decr\_misses 0

delete\_hits 0

delete\_misses 0

evictions 0

get\_hits 52784

get\_misses 355

incr\_hits 0

incr\_misses 0

limit\_maxbytes 314572800

listen\_disabled\_num 0

pid 7226

pointer\_size 32

reclaimed 0

rusage\_system 10.676667

rusage\_user 3.068191

threads 4

time 1320919080

total\_connections 111

total\_items 229120

uptime 7649

version 1.4.5

### Warming the edb-icache Servers

When the server starts, the primary and secondary caches are empty. When Advanced Server processes a client request, the server reads the required data from disk and stores a copy in each cache. You can improve server performance by warming (or pre-loading) the data into the memory cache before a client asks for it.

There are two advantages to warming the cache. Advanced Server will find data in the cache the first time it is requested by a client application, instead of waiting for it to be read from disk. Also, manually warming the cache with the data that your applications are most likely to need saves time by avoiding future random disk reads. If you don't warm the cache at startup, Advanced Server performance may not reach full speed until the client applications happen to load commonly used data into the cache.

There are several ways to load pages to warm the Infinite Cache server nodes. You can:

* Use the edb\_icache\_warm utility to warm the caches from the command line.
* Use the edb\_icache\_warm() function from within edb-psql.
* Use the edb\_icache\_warm() function via scripts to warm the cache.

While it is not necessary to re-warm the cache after making changes to an existing cache configuration, re-warming the cache can improve performance by bringing the new configuration of cache servers up-to-date.

#### The edb\_icache\_warm() Function

The edb\_icache\_warm() function comes in two variations; the first variation warms not only the table, but any indexes associated with the table. If you use the second variation, you must make additional calls to warm any associated indexes.

The first form of the edb\_icache\_warm() function warms the given table and any associated indexes into the cache. The signature is:

edb\_icache\_warm(table\_name)

You may specify table\_name as a table name, OID, or regclass value.

# edb-psql edb -c "select edb\_icache\_warm('accounts')"

When you call the first form of edb\_icache\_warm(), Advanced Server reads each page in the given table, compresses the page (if configured to do so), and then sends the compressed data to an Infinite Cache server. edb\_icache\_warm() also reads, compresses, and caches each page in each index defined for the given table.

The second form of the edb\_icache\_warm() function warms the pages that contain the specified range of bytes into the cache. The signature of the second form is:

edb\_icache\_warm(table-spec, startbyte, endbyte):

You must make subsequent calls to specify indexes separately when using this form of the edb\_icache\_warm() function.

# edb-psql edb -c "select edb\_icache\_warm('accounts', 1, 10000)"

The edb\_icache\_warm() function is typically called by a utility program (such as the edb\_icache\_warm utility) to spread the warming process among multiple processes that operate in parallel.

#### Using the edb\_icache\_warm Utility

You can use the edb\_icache\_warm command-line utility to load the cache servers with specified tables, allowing fast access to relevant data from the cache.

The syntax for edb\_icache\_warm is:

# edb\_icache\_warm –d database –t tablename

The only required parameter is tablename. tablename can be specified with or without the -t option. All other parameters are optional; if omitted, default values are inferred from Advanced Server environment variables.

The options for edb\_icache\_warm are:

|  |  |  |
| --- | --- | --- |
| Option | Variable | Description |
| -h | Hostname | The name of the host running Advanced Server. Include this parameter if you are running Advanced Server on a remote host.  The default value is PGHOST. |
| -p | Portname | Port in use by Advanced Server. Default value is PGPORT. |
| -j | process count | Number of (parallel) processes used to warm the cache. The default value is 1. |
| -U | Username | The Advanced Server username. Unless specified, this defaults to PGUSER. |
| -d | Database | The name of database containing the tables to be warmed. Default value is PGDATABASE. |
| -t | Tablename | Name of table to be warmed. The index for the table is also warmed. Required. |

### Retrieving Statistics from Infinite Cache

#### Using edb\_icache\_stats()

You can view Infinite Cache statistics by using the edb\_icache\_stats() function at the edb-psql command line (or any other query tool). The edb\_icache\_stats() function returns a result set that reflects the state of an Infinite Cache node or nodes and the related usage statistics. The result set includes:

|  |  |
| --- | --- |
| Statistic | Description |
| hostname | Host name (or IP address) of server |
| Port | Port number at which edb-icache daemon is listening |
| State | Health of this server |
| write\_failures | Number of write failures |
| Bytes | Total number of bytes in use |
| bytes\_read | Total number of bytes received by this server (from the network) |
| bytes\_written | Total number of bytes sent by this server (to the network) |
| cmd\_get | Cumulative number of read requests sent to this server |
| cmd\_set | Cumulative number of write requests sent to this server |
| connection\_structures | Number of connection structures allocated by the server |
| curr\_connections | Number of open connections |
| curr\_items | Number of items currently stored by the server |
| Evictions | Number of valid items removed from cache to free memory for new items |
| get\_hits | Number of read requests satisfied by this server |
| get\_misses | Number of read requests not satisfied by this server |
| limit\_maxbytes | Number of bytes allocated on this server for storage |
| Pid | Process ID (on cache server) |
| pointer\_size | Default pointer size on host OS (usually 32 or 64) |
| rusage\_user | Accumulated user time for this process (seconds.microseconds) |
| rusage\_system | Accumulated system time for this process (seconds.microseconds) |
| Threads | Number of worker threads requested |
| total\_time | Number of seconds since this server's base date (usually midnight, January 1, 1970, UTC) |
| total\_connections | Total number of connections opened since the server started running |
| total\_items | Total number of items stored by this server (cumulative) |
| Uptime | Amount of time that server has been active |
| Version | edb-icache version |

You can use SQL queries to view Infinite Cache statistics. To view the server status of all Infinite Cache nodes:

SELECT hostname, port, state FROM edb\_icache\_stats()

hostname | port | state

---------------+-------+--------

192.168.23.85 | 11211 | UNHEALTHY

192.168.23.85 | 11212 | ACTIVE

(2 rows)

Use the following command to view complete statistics (shown here using edb-psql's expanded display mode, \x) for a specified node:

SELECT \* FROM edb\_icache\_stats() WHERE hostname = '192.168.23.85:11211'

-[RECORD 1]-----------+--------------

hostname | 192.168.23.85

port | 11211

state | ACTIVE

write\_failures | 0

bytes | 225029460

bytes\_read | 225728252

bytes\_written | 192806774

cmd\_get | 23313

cmd\_set | 27088

connection\_structures | 53

curr\_connections | 3

curr\_items | 27088

evictions | 0

get\_hits | 23266

get\_misses | 47

limit\_maxbytes | 805306368

pid | 4240

pointer\_size | 32

rusage\_user | 0.481926

rusage\_system | 1.583759

threads | 1

total\_time | 1242199782

total\_connections | 66

total\_items | 27088

uptime | 714

version | 1.2.6

#### edb\_icache\_server\_list

The edb\_icache\_server\_list view exposes information about the status and health of all Infinite Cache servers listed in the edb\_icache\_servers GUC. The edb\_icache\_server\_list view is created using the edb\_icache stats() API. The view exposes the following information for each server:

|  |  |
| --- | --- |
| Statistic | Description |
| Hostname | Host name (or IP address) of server |
| Port | Port number at which edb-icache daemon is listening |
| State | Health of this server |
| write\_failures | Number of write failures |
| total\_memory | Number of bytes allocated to the cache on this server |
| memory\_used | Number of bytes currently used by the cache |
| memory\_free | Number of unused bytes remaining in the cache |
| hit\_ratio | Percentage of cache hits |

The state column will contain one of the following four values, reflecting the health of the given server:

|  |  |
| --- | --- |
| Server State | Description |
| Active | The server is known to be up and running. |
| Unhealthy | An error occurred while interacting with the cache server. Postgres will attempt to re-establish the connection with the server. |
| Offline | Postgres can no longer contact the given server. |
| Manual Offline | You have taken the server offline with the edb\_icache\_server\_enable() function. |

Use the following SELECT statement to return the health of each node in the Infinite Cache server farm:

SELECT hostname, port, state FROM edb\_icache\_server\_list

hostname | port | state

---------------+-------+-------

192.168.23.85 | 11211 | ACTIVE

192.168.23.85 | 11212 | ACTIVE

(2 rows)

Use the following command to view complete details about a specific Infinite Cache node (shown here using edb-psql's \x expanded-view option):

SELECT \* FROM edb\_icache\_server\_list WHERE hostname = '192.168.23.85:11211'

-[RECORD 1]-----------+--------------

hostname | 192.168.23.85

port | 11211

state | ACTIVE

write\_failures | 0

total\_memory | 805306368

memory\_used | 225029460

memory\_free | 580276908

hit\_ratio | 99.79

### Retrieving Table Statistics

Advanced Server provides six system views that contain statistical information on a per-table basis. The views are:

* pg\_statio\_all\_tables
* pg\_statio\_sys\_tables
* pg\_statio\_user\_tables
* pg\_statio\_all\_indexes
* pg\_statio\_sys\_indexes
* pg\_statio\_user\_indexes

You can use standard SQL queries to view and compare the information stored in the views. The views contain information that will allow you to observe the effectiveness of the Advanced Server buffer cache and the icache servers.

#### pg\_statio\_all\_tables

The pg\_statio\_all\_tables view contains one row for each table in the database. The view contains the following information:

|  |  |
| --- | --- |
| Column Name | Description |
| relid | The OID of the table. |
| schemaname | The name of the schema that the table resides in. |
| relname | The name of the table. |
| heap\_blks\_read | The number of heap blocks read. |
| heap\_blks\_hit | The number of heap blocks hit. |
| heap\_blks\_icache\_hit | The number of heap blocks found on an icache server. |
| idx\_blks\_read | The number of index blocks read. |
| idx\_blks\_hit | The number of index blocks hit. |
| idx\_blks\_icache\_hit | The number of index blocks found on an icache server. |
| toast\_blks\_read | The number of toast blocks read. |
| toast\_blks\_hit | The number of toast blocks hit. |
| toast\_blks\_icache\_hit | The number of toast blocks found on an icache server. |
| tidx\_blks\_read | The number of index toast blocks read. |
| tidx\_blks\_hit | The number of index toast blocks hit. |
| tidx\_blks\_icache\_hit | The number of index toast blocks found on an icache server. |

You can execute a simple query to view performance statistics for a specific table:

SELECT \* FROM pg\_statio\_all\_tables WHERE relname=’jobhist’;

-[ RECORD 1 ]---------+---------

relid | 16402

schemaname | public

relname | jobhist

heap\_blks\_read | 1

heap\_blks\_hit | 51

heap\_blks\_icache\_hit | 0

idx\_blks\_read | 2

idx\_blks\_hit | 17

idx\_blks\_icache\_hit | 0

toast\_blks\_read |

toast\_blks\_hit |

toast\_blks\_icache\_hit |

tidx\_blks\_read |

tidx\_blks\_hit |

tidx\_blks\_icache\_hit |

Or, you can view the statistics by activity level. The following example displays the statistics for the ten tables that have the greatest heap\_blks\_icache\_hit activity:

SELECT \* FROM pg\_statio\_all\_tables ORDER BY heap\_blks\_icache\_hit DESC LIMIT 10;

relid schemaname relname

heap\_blks\_read heap\_blks\_hit heap\_blks\_icache\_hit

idx\_blks\_read idx\_blks\_hit idx\_blks\_icache\_hit

toast\_blks\_read toast\_blks\_hit toast\_blks\_icache\_hit

tidx\_blks\_read tidx\_blks\_hit tidx\_blks\_icache\_hit

-----------------------------------------------------------------------------

16390 public pgbench\_accounts

264105 71150 81498

13171 282541 18053

1259 pg\_catalog pg\_class

22 2904 18

14 3449 11

1249 pg\_catalog pg\_attribute

49 1619 16

17 2841 13

1255 pg\_catalog pg\_proc

38 276 11

33 682 16

0 0 0

0 0 0

2619 pg\_catalog pg\_statistic

20 295 8

4 436 4

0 0 0

0 0 0

2617 pg\_catalog pg\_operator

20 293 8

19 791 10

2602 pg\_catalog pg\_amop

10 721 6

13 1154 13

2610 pg\_catalog pg\_index

10 633 6

8 719 8

1247 pg\_catalog pg\_type

17 235 5

12 433 4

2615 pg\_catalog pg\_namespace

4 260 4

6 330 4

0 0 0

0 0 0

(10 rows)

#### pg\_statio\_sys\_tables

The pg\_statio\_sys\_tables view contains one row for each table in a system-defined schema. The statistical information included in this view is the same as for pg\_statio\_all\_tables.

#### pg\_statio\_user\_tables

The pg\_statio\_user\_tables view contains one row for each table in a user-defined schema. The statistical information in this view is the same as for pg\_statio\_all\_tables.

#### pg\_statio\_all\_indexes

The pg\_statio\_all\_indexes view contains one row for each index in the current database. The view contains the following information:

|  |  |
| --- | --- |
| Column Name | Description |
| relid | The OID of the indexed table |
| indexrelid | The OID of the index. |
| schemaname | The name of the schema that the table resides in. |
| relname | The name of the table. |
| indexrelname | The name of the index |
| idx\_blks\_read | The number of index blocks read. |
| idx\_blks\_hit | The number of index blocks hit. |
| idx\_blks\_icache\_hit | The number of index blocks found on an icache server. |

You can execute a simple query to view performance statistics for the indexes on a specific table:

SELECT \* FROM pg\_statio\_all\_indexes WHERE relname=’pg\_attribute’;

-[ RECORD 1 ]---------+---------

relid | 1249

indexrelid | 2658

schemaname | pg\_catalog

relname | pg\_attribute

indexrelname | pg\_attribute\_relid\_attnam\_index

idx\_blks\_read | 10

idx\_blks\_hit | 1200

idx\_blks\_icache\_hit | 0

-[ RECORD 2 ]---------+---------

relid | 1249

indexrelid | 2659

schemaname | pg\_catalog

relname | pg\_attribute

indexrelname | pg\_attribute\_relid\_attnum\_index

idx\_blks\_read | 12

idx\_blks\_hit | 3917

idx\_blks\_icache\_hit | 0

The result set from the query includes the statistical information for two indexes; the pg\_attribute table has two indexes.

You can also view the statistics by activity level. The following example displays the statistics for the ten indexes that have the greatest idx\_blks\_icache\_hit activity:

SELECT \* FROM pg\_statio\_all\_indexes ORDER BY idx\_blks\_icache\_hit DESC LIMIT 10;

relid indexrelid schemaname relname

indexrelname idx\_blks\_read idx\_blks\_hit idx\_blks\_icache\_hit

-----------------------------------------------------------------------------

16390 16401 public pgbench\_accounts

pgbench\_accounts\_pkey 13171 282541 18053

1249 2659 pg\_catalog pg\_attribute

pg\_attr\_relid\_attnum\_index 14 2749 13

1255 2690 pg\_catalog proc

pg\_proc\_oid\_index 16 580 12

1259 2663 pg\_catalog pg\_class

pg\_class\_relname\_nsp\_index 10 2019 7

2602 2654 pg\_catalog pg\_amop

pg\_amop\_opr\_fam\_index 7 453 7

2603 2655 pg\_catalog pg\_amproc

pg\_amproc\_fam\_proc\_index 6 605 6

2617 2688 pg\_catalog pg\_operator

pg\_operator\_oid\_index 7 452 6

2602 2653 pg\_catalog pg\_amop

pg\_amop\_fam\_strat\_index 6 701 6

2615 2684 pg\_catalog pg\_namespace pg\_namespace\_nspname\_index 4 328 4

1262 2672 pg\_catalog pg\_database pg\_database\_oid\_index 4 254 4

#### pg\_statio\_sys\_indexes

The pg\_statio\_sys\_indexes view contains one row for each index on the system tables. The statistical information in this view is the same as in pg\_statio\_all\_indexes.

#### pg\_statio\_user\_indexes

The pg\_statio\_user\_indexes view contains one row for each index on a table that resides in a user-defined schema. The statistical information in this view is the same as in pg\_statio\_all\_indexes.

### edb\_icache\_server\_enable()

You can use the edb\_icache\_server\_enable() function to take the Infinite Cache server offline for maintenance or other planned downtime. The syntax is:

void edb\_icache\_server\_enable(*host* TEXT, *port* INTEGER, *online* BOOL)

host specifies the host that you want to disable. The host name may be specified by name or numeric address.

port specifies the port number that the Infinite Cache server is listening on.

online specifies the state of the Infinite Cache server. The value of online must be true or false.

To take a server offline, specify the host that you want to disable, the port number that the Infinite Cache server is listening on, and false. To bring the Infinite Cache server back online, specify the host name and port number, and pass a value of true.

The state of a server taken offline with the edb\_icache\_server\_enable() function is MANUAL OFFLINE. Advanced Server will not automatically reconnect to an Infinite Cache server that you have taken offline with edb\_icache\_server\_enable(..., false); you must bring the server back online by calling edb\_icache\_server\_enable(..., true).

### Infinite Cache Log Entries

When you start Advanced Server, a message that includes Infinite Cache status, cache node count and cache node size is written to the server log. The following example shows the server log for an active Infinite Cache installation with two 750 MB cache servers:

\*\* EnterpriseDB Dynamic Tuning Agent\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* System Utilization: 66 % \*

\* Autovacuum Naptime: 60 Seconds \*

\* Infinite Cache: on \*

\* Infinite Cache Servers: 2 \*

\* Infinite Cache Size: 1.500 GB \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

### Allocating Memory to the Cache Servers

As mentioned earlier in this document, each computer imposes a limit on the amount of physical memory that you can install. However, modern operating systems typically simulate a larger address space so that programs can transparently access more memory than is actually installed. This "virtual memory" allows a computer to run multiple programs that may simultaneously require more memory than is physically available. For example, you may run an e-mail client, a web browser, and a database server which each require 1GB of memory on a machine that contains only 2GB of physical RAM. When the operating system runs out of physical memory, it starts swapping bits and pieces of the currently running programs to disk to make room to satisfy your current demand for memory.

*This can bring your system to a grinding halt.*

Since the primary goal of Infinite Cache is to improve performance by limiting disk I/O, you should avoid dedicating so much memory to Infinite Cache that the operating system must start swapping data to disk. If the operating system begins to swap to disk, you lose the benefits offered by Infinite Cache.

The overall demand for physical memory can vary throughout the day; if the server is frequently idle, you may never encounter swapping. If you have dedicated a large portion of physical memory to the cache, and system usage increases, the operating system may start swapping. To get the best performance and avoid disk swapping, dedicate a server node to Infinite Cache so other applications on that computer will not compete for physical memory.

## Index Advisor

The Index Advisor utility helps determine which columns you should index to improve performance in a given workload. Index Advisor considers B-tree (single-column or composite) index types, and does not identify other index types (GIN, GiST, Hash) that may improve performance. Index Advisor is installed with EDB Postgres Advanced Server.

Index Advisor works with Advanced Server's query planner by creating *hypothetical* *indexes* that the query planner uses to calculate execution costs as if such indexes were available. Index Advisor identifies the indexes by analyzing SQL queries supplied in the workload.

There are three ways to use Index Advisor to analyze SQL queries:

* Invoke the Index Advisor utility program, supplying a text file containing the SQL queries that you wish to analyze; Index Advisor will generate a text file with CREATE INDEX statements for the recommended indexes.
* Provide queries at the EDB-PSQL command line that you want Index Advisor to analyze.
* Access Index Advisor through the Postgres Enterprise Manager client. When accessed via the PEM client, Index Advisor works with SQL Profiler, providing indexing recommendations on code captured in SQL traces. For more information about using SQL Profiler and Index Advisor with PEM, please see Section 8.4 of the PEM Getting Started Guide available from the EnterpriseDB website at:

<http://www.enterprisedb.com/products-services-training/products/postgres-enterprise-manager>

Index Advisor will attempt to make indexing recommendations on INSERT, UPDATE, DELETE and SELECT statements. When invoking Index Advisor, you supply the workload in the form of a set of queries (if you are providing the command in an SQL file) or an EXPLAIN statement (if you are specifying the SQL statement at the psql command line). Index Advisor displays the query plan and estimated execution cost for the supplied query, but does not actually execute the query.

During the analysis, Index Advisor compares the query execution costs with and without hypothetical indexes. If the execution cost using a hypothetical index is less than the execution cost without it, both plans are reported in the EXPLAIN statement output, metrics that quantify the improvement are calculated, and Index Advisor generates the CREATE INDEX statement needed to create the index.

If no hypothetical index can be found that reduces the execution cost, Index Advisor displays only the original query plan output of the EXPLAIN statement.

Index Advisor does not actually create indexes on the tables. Use the CREATE INDEX statements supplied by Index Advisor to add any recommended indexes to your tables.

A script supplied with Advanced Server creates the table in which Index Advisor stores the indexing recommendations generated by the analysis; the script also creates a function and a view of the table to simplify the retrieval and interpretation of the results.

If you choose to forego running the script, Index Advisor will log recommendations in a temporary table that is available only for the duration of the Index Advisor session.

### Index Advisor Components

The Index Advisor shared library interacts with the query planner to make indexing recommendations. The Advanced Server installer creates the following shared library in the libdir subdirectory of your Advanced Server home directory:

On Linux:

index\_advisor.so

On Windows:

index\_advisor.dll

Please note that libraries in the libdir directory can only be loaded by a superuser. A database administrator can allow a non-superuser to use Index Advisor by manually copying the Index Advisor file from the libdir directory into the libdir/plugins directory (under your Advanced Server home directory). Only a trusted non-superuser should be allowed access to the plugin; this is an unsafe practice in a production environment.

The installer also creates the Index Advisor utility program and setup script:

pg\_advise\_index

pg\_advise\_index is a utility program that reads a user-supplied input file containing SQL queries and produces a text file containing CREATE INDEX statements that can be used to create the indexes recommended by the Index Advisor. The pg\_advise\_index program is located in the bin subdirectory of the Advanced Server home directory.

index\_advisor.sql

index\_advisor.sql is a script that creates a permanent Index Advisor log table along with a function and view to facilitate reporting of recommendations from the log table. The script is located in the share/contrib subdirectory of the Advanced Server directory.

The index\_advisor.sql script creates the index\_advisor\_log table, the show\_index\_recommendations() function and the index\_recommendations view. These database objects must be created in a schema that is accessible by, and included in the search path of the role that will invoke Index Advisor.

index\_advisor\_log

Index Advisor logs indexing recommendations in the index\_advisor\_log table. If Index Advisor does not find the index\_advisor\_log table in the user's search path, Index Advisor will store any indexing recommendations in a temporary table of the same name. The temporary table exists only for the duration of the current session.

show\_index\_recommendations()

show\_index\_recommendations() is a PL/pgSQL function that interprets and displays the recommendations made during a specific Index Advisor session (as identified by its backend process ID).

index\_recommendations

Index Advisor creates the index\_recommendations view based on information stored in the index\_advisor\_log table during a query analysis. The view produces output in the same format as the show\_index\_recommendations() function, but contains Index Advisor recommendations for all stored sessions, while the result set returned by the show\_index\_recommendations() function are limited to a specified session.

### Index Advisor Configuration

Index Advisor does not require any configuration to generate recommendations that are available only for the duration of the current session; to store the results of multiple sessions, you must create the index\_advisor\_log table (where Advanced Server will store Index Advisor recommendations). To create the index\_advisor\_log table , you must run the index\_advisor.sql script.

When selecting a storage schema for the Index Advisor table, function and view, keep in mind that all users that invoke Index Advisor (and query the result set) must have USAGE privileges on the schema. The schema must be in the search path of all users that are interacting with the Index Advisor.

1. Place the selected schema at the start of your search\_path parameter. For example, if your search path is currently:

search\_path=public, accounting

and you want the Index Advisor objects to be created in a schema named advisor, use the command:

SET search\_path = advisor, public, accounting;

1. Run the index\_advisor.sql script to create the database objects. If you are running the psql client, you can use the command:

\i *full*\_*pathname*/index\_advisor.sql

Specify the pathname to the index\_advisor.sql script in place of *full\_pathname*.

1. Grant privileges on the index\_advisor\_log table to all Index Advisor users; this step is not necessary if the Index Advisor user is a superuser, or the owner of these database objects.

* Grant SELECT and INSERT privileges on the index\_advisor\_log table to allow a user to invoke Index Advisor.
* Grant DELETE privileges on the index\_advisor\_log table to allow the specified user to delete the table contents.
* Grant SELECT privilege on the index\_recommendations view.

The following example demonstrates the creation of the Index Advisor database objects in a schema named ia, which will then be accessible to an Index Advisor user with user name ia\_user:

$ edb-psql -d edb -U enterprisedb

edb-psql (9.5.0.0)

Type "help" for help.

edb=# CREATE SCHEMA ia;  
CREATE SCHEMA  
edb=# SET search\_path TO ia;

SET  
edb=# \i /opt/PostgresPlus/9.5AS/share/contrib/index\_advisor.sql  
CREATE TABLE  
CREATE INDEX  
CREATE INDEX  
CREATE FUNCTION  
CREATE FUNCTION  
CREATE VIEW  
edb=# GRANT USAGE ON SCHEMA ia TO ia\_user;  
GRANT  
edb=# GRANT SELECT, INSERT, DELETE ON index\_advisor\_log TO ia\_user;  
GRANT  
edb=# GRANT SELECT ON index\_recommendations TO ia\_user;  
GRANT

While using Index Advisor, the specified schema (ia) must be included in ia\_user's search\_path parameter.

### Using Index Advisor

When you invoke Index Advisor, you must supply a workload; the workload is either a query (specified at the command line), or a file that contains a set of queries (executed by the pg\_advise\_index() function). After analyzing the workload, Index Advisor will either store the result set in a temporary table, or in a permanent table. You can review the indexing recommendations generated by Index Advisor and use the CREATE INDEX statements generated by Index Advisor to create the recommended indexes.

Note: You should not run Index Advisor in read-only transactions.

The following examples assume that superuser enterprisedb is the Index Advisor user, and the Index Advisor database objects have been created in a schema in the search\_path of superuser enterprisedb.

The examples in the following sections use the table created with the statement shown below:

CREATE TABLE t( a INT, b INT );  
INSERT INTO t SELECT s, 99999 - s FROM generate\_series(0,99999) AS s;

ANALYZE t;

The resulting table contains the following rows:

a | b

-------+-------

0 | 99999

1 | 99998

2 | 99997

3 | 99996

.

.

.

99997 | 2

99998 | 1

99999 | 0

#### Using the pg\_advise\_index Utility

When invoking the pg\_advise\_index utility, you must include the name of a file that contains the queries that will be executed by pg\_advise\_index; the queries may be on the same line, or on separate lines, but each query must be terminated by a semicolon. Queries within the file should not begin with the EXPLAIN keyword.

The following example shows the contents of a sample workload.sql file:

SELECT \* FROM t WHERE a = 500;

SELECT \* FROM t WHERE b < 1000;

Run the pg\_advise\_index program as shown in the code sample below:

$ pg\_advise\_index -d edb -h localhost -U enterprisedb -s 100M -o advisory.sql workload.sql

poolsize = 102400 KB

load workload from file 'workload.sql'

Analyzing queries .. done.

size = 2184 KB, benefit = 1684.720000

size = 2184 KB, benefit = 1655.520000

/\* 1. t(a): size=2184 KB, benefit=1684.72 \*/

/\* 2. t(b): size=2184 KB, benefit=1655.52 \*/

/\* Total size = 4368KB \*/

In the code sample, the -d, -h, and -U options are psql connection options.

-s

-s is an optional parameter that limits the maximum size of the indexes recommended by Index Advisor. If Index Advisor does not return a result set, -s may be set too low.

-o

The recommended indexes are written to the file specified after the -o option.

The information displayed by the pg\_advise\_index program is logged in the index\_advisor\_log table. In response to the command shown in the example, Index Advisor writes the following CREATE INDEX statements to the advisory.sql output file

create index idx\_t\_1 on t (a);  
create index idx\_t\_2 on t (b);

You can create the recommended indexes at the psql command line with the CREATE INDEX statements in the file, or create the indexes by executing the advisory.sql script.

$ edb-psql -d edb -h localhost -U enterprisedb -e -f advisory.sql

create index idx\_t\_1 on t (a);

CREATE INDEX

create index idx\_t\_2 on t (b);

CREATE INDEX

#### Using Index Advisor at the psql Command Line

You can use Index Advisor to analyze SQL statements entered at the edb-psql (or psql) command line; the following steps detail loading the Index Advisor plugin and using Index Advisor:

1. Connect to the server with the edb-psql command line utility, and load the Index Advisor plugin:

$ edb-psql -d edb -U enterprisedb   
…  
edb=# LOAD 'index\_advisor';  
LOAD

1. Use the edb-psql command line to invoke each SQL command that you would like Index Advisor to analyze. Index Advisor stores any recommendations for the queries in the index\_advisor\_log table. If the index\_advisor\_log table does not exist in the user's search\_path, a temporary table is created with the same name. This temporary table exists only for the duration of the user's session.

After loading the Index Advisor plugin, Index Advisor will analyze all SQL statements and log any indexing recommendations for the duration of the session.

If you would like Index Advisor to analyze a query (and make indexing recommendations) without actually executing the query, preface the SQL statement with the EXPLAIN keyword.

If you do not preface the statement with the EXPLAIN keyword, Index Advisor will analyze the statement while the statement executes, writing the indexing recommendations to the index\_advisor\_log table for later review.

In the example that follows, the EXPLAIN statement displays the normal query plan, followed by the query plan of the same query, if the query were using the recommended hypothetical index:

edb=# EXPLAIN SELECT \* FROM t WHERE a < 10000;

QUERY PLAN

-----------------------------------------------------------------------------

Seq Scan on t (cost=0.00..1693.00 rows=10105 width=8)

Filter: (a < 10000)

Result (cost=0.00..337.10 rows=10105 width=8)

One-Time Filter: '===[ HYPOTHETICAL PLAN ]==='::text

-> Index Scan using "<hypothetical-index>:1" on t   
 (cost=0.00..337.10 rows=10105 width=8)

Index Cond: (a < 10000)

(6 rows)

edb=# EXPLAIN SELECT \* FROM t WHERE a = 100;

QUERY PLAN

-----------------------------------------------------------------------------

Seq Scan on t (cost=0.00..1693.00 rows=1 width=8)

Filter: (a = 100)

Result (cost=0.00..8.28 rows=1 width=8)

One-Time Filter: '===[ HYPOTHETICAL PLAN ]==='::text

-> Index Scan using "<hypothetical-index>:3" on t   
 (cost=0.00..8.28 rows=1 width=8)

Index Cond: (a = 100)

(6 rows)

For information about reviewing the recommended queries, see Section 8.3.4.

After loading the Index Advisor plugin, the default value of index\_advisor.enabled is on. The Index Advisor plugin must be loaded to use a SET or SHOW command to display the current value of index\_advisor.enabled.

You can use the index\_advisor.enabled parameter to temporarily disable Index Advisor without interrupting the psql session:

edb=# SET index\_advisor.enabled TO off;  
SET

To enable Index Advisor, set the parameter to on:

edb=# SET index\_advisor.enabled TO on;  
 SET

### Reviewing the Index Advisor Recommendations

There are several ways to review the index recommendations generated by Index Advisor. You can:

* Query the index\_advisor\_log table.
* Run the show\_index\_recommendations function.
* Query the index\_recommendations view.

#### Using the show\_index\_recommendations() Function

To review the recommendations of the Index Advisor utility using the show\_index\_recommendations() function, call the function, specifying the process ID of the session:

SELECT show\_index\_recommendations( pid );

Where pid is the process ID of the current session. If you do not know the process ID of your current session, passing a value of NULL will also return a result set for the current session.

The following code fragment shows an example of a row in a result set:

edb=# SELECT show\_index\_recommendations(null);

show\_index\_recommendations

---------------------------------------------------------------------

create index idx\_t\_a on t(a);/\* size: 2184 KB, benefit: 3040.62,

gain: 1.39222666981456 \*/

(1 row)

In the example, create index idx\_t\_a on t(a) is the SQL statement needed to create the index suggested by Index Advisor. Each row in the result set shows:

* The command required to create the recommended index.
* The maximum estimated size of the index.
* The calculated benefit of using the index.
* The estimated gain that will result from implementing the index.

You can display the results of all Index Advisor sessions from the following view:

SELECT \* FROM index\_recommendations;

#### Querying the index\_advisor\_log Table

Index Advisor stores indexing recommendations in a table named index\_advisor\_log. Each row in the index\_advisor\_log table contains the result of a query where Index Advisor determines it can recommend a hypothetical index to reduce the execution cost of that query.

| Column | Type | Description |
| --- | --- | --- |
| reloid | oid | OID of the base table for the index |
| relname | name | Name of the base table for the index |
| attrs | integer[] | Recommended index columns (identified by column number) |
| benefit | real | Calculated benefit of the index for this query |
| index\_size | integer | Estimated index size in disk-pages |
| backend\_pid | integer | Process ID of the process generating this recommendation |
| timestamp | timestamp | Date/Time when the recommendation was generated |

You can query the index\_advisor\_log table at the psql command line. The following example shows the index\_advisor\_log table entries resulting from two Index Advisor sessions. Each session contains two queries, and can be identified (in the table below) by a different backend\_pid value. For each session, Index Advisor generated two index recommendations.

edb=# SELECT \* FROM index\_advisor\_log;

reloid | relname | attrs | benefit | index\_size | backend\_pid | timestamp

--------+---------+-------+---------+------------+-------------+----------------------------------

16651 | t | {1} | 1684.72 | 2184 | 3442 | 22-MAR-11 16:44:32.712638 -04:00

16651 | t | {2} | 1655.52 | 2184 | 3442 | 22-MAR-11 16:44:32.759436 -04:00

16651 | t | {1} | 1355.9 | 2184 | 3506 | 22-MAR-11 16:48:28.317016 -04:00

16651 | t | {1} | 1684.72 | 2184 | 3506 | 22-MAR-11 16:51:45.927906 -04:00

(4 rows)

Index Advisor added the first two rows to the table after analyzing the following two queries executed by the pg\_advise\_index utility:

SELECT \* FROM t WHERE a = 500;

SELECT \* FROM t WHERE b < 1000;

The value of 3442 in column backend\_pid identifies these results as coming from the session with process ID 3442.

The value of 1 in column attrs in the first row indicates that the hypothetical index is on the first column of the table (column a of table t).

The value of 2 in column attrs in the second row indicates that the hypothetical index is on the second column of the table (column b of table t).

Index Advisor added the last two rows to the table after analyzing the following two queries (executed at the psql command line):

edb=# EXPLAIN SELECT \* FROM t WHERE a < 10000;

QUERY PLAN

----------------------------------------------------------------------------------------------

Seq Scan on t (cost=0.00..1693.00 rows=10105 width=8)

Filter: (a < 10000)

Result (cost=0.00..337.10 rows=10105 width=8)

One-Time Filter: '===[ HYPOTHETICAL PLAN ]==='::text

-> Index Scan using "<hypothetical-index>:1" on t (cost=0.00..337.10 rows=10105 width=8)

Index Cond: (a < 10000)

(6 rows)

edb=# EXPLAIN SELECT \* FROM t WHERE a = 100;

QUERY PLAN

----------------------------------------------------------------------------------------

Seq Scan on t (cost=0.00..1693.00 rows=1 width=8)

Filter: (a = 100)

Result (cost=0.00..8.28 rows=1 width=8)

One-Time Filter: '===[ HYPOTHETICAL PLAN ]==='::text

-> Index Scan using "<hypothetical-index>:3" on t (cost=0.00..8.28 rows=1 width=8)

Index Cond: (a = 100)

(6 rows)

The values in the benefit column of the index\_advisor\_log table are calculated using the following formula:

benefit = (normal execution cost) - (execution cost with hypothetical index)

The value of the benefit column for the last row of the index\_advisor\_log table (shown in the example) is calculated using the query plan for the following SQL statement:

EXPLAIN SELECT \* FROM t WHERE a = 100;

The execution costs of the different execution plans are evaluated and compared:

benefit = (Seq Scan on t cost) - (Index Scan using <hypothetical-index>)

and the benefit is added to the table:

benefit = 1693.00 - 8.28  
benefit = 1684.72

You can delete rows from the index\_advisor\_log table when you no longer have the need to review the results of the queries stored in the row.

#### Querying the index\_recommendations View

The index\_recommendations view contains the calculated metrics and the CREATE INDEX statements to create the recommended indexes for all sessions whose results are currently in the index\_advisor\_log table. You can display the results of all stored Index Advisor sessions by querying the index\_recommendations view as shown below:

SELECT \* FROM index\_recommendations;

Using the example shown in the previous section (Querying the index\_advisor\_log Table), the index\_recommendations view displays the following:

edb=# SELECT \* FROM index\_recommendations;

backend\_pid | show\_index\_recommendations

-------------+---------------------------------------------------------------------------------------------

3442 | create index idx\_t\_a on t(a);/\* size: 2184 KB, benefit: 1684.72, gain: 0.771392654586624 \*/

3442 | create index idx\_t\_b on t(b);/\* size: 2184 KB, benefit: 1655.52, gain: 0.758021539820856 \*/

3506 | create index idx\_t\_a on t(a);/\* size: 2184 KB, benefit: 3040.62, gain: 1.39222666981456 \*/

(3 rows)

Within each session, the results of all queries that benefit from the same recommended index are combined to produce one set of metrics per recommended index, reflected in the fields named benefit and gain.

The formulas for the fields are as follows:

size = MAX(index size of all queries)

benefit = SUM(benefit of each query)

gain = SUM(benefit of each query) / MAX(index size of all queries)

So for example, using the following query results from the process with a backend\_pid of 3506:

reloid | relname | attrs | benefit | index\_size | backend\_pid | timestamp

--------+---------+-------+---------+------------+-------------+----------------------------------

16651 | t | {1} | 1355.9 | 2184 | 3506 | 22-MAR-11 16:48:28.317016 -04:00

16651 | t | {1} | 1684.72 | 2184 | 3506 | 22-MAR-11 16:51:45.927906 -04:00

The metrics displayed from the index\_recommendations view for backend\_pid 3506 are:

backend\_pid | show\_index\_recommendations

-------------+---------------------------------------------------------------------------------------------

3506 | create index idx\_t\_a on t(a);/\* size: 2184 KB, benefit: 3040.62, gain: 1.39222666981456 \*/

The metrics from the view are calculated as follows:

benefit = (benefit from 1st query) + (benefit from 2nd query)

benefit = 1355.9 + 1684.72

benefit = 3040.62

and

gain = ((benefit from 1st query) + (benefit from 2nd query)) / MAX(index size of all queries)

gain = (1355.9 + 1684.72) / MAX(2184, 2184)

gain = 3040.62 / 2184

gain = 1.39223

The gain metric is useful when comparing the relative advantage of the different recommended indexes derived during a given session. The larger the gain value, the better the cost effectiveness derived from the index weighed against the possible disk space consumption of the index.

### Limitations

Index Advisor does not consider Index Only scans; it does consider Index scans when making recommendations.

Index Advisor ignores any computations found in the WHERE clause. Effectively, the index field in the recommendations will not be any kind of expression; the field will be a simple column name.

Index Advisor does not consider inheritance when recommending hypothetical indexes. If a query references a parent table, Index Advisor does not make any index recommendations on child tables.

Restoration of a pg\_dump backup file that includes the index\_advisor\_log table or any tables for which indexing recommendations were made and stored in the index\_advisor\_log table, may result in "broken links" between the index\_advisor\_log table and the restored tables referenced by rows in the index\_advisor\_log table because of changes in object identifiers (OIDs).

If it is necessary to display the recommendations made prior to the backup, you can replace the old OIDs in the reloid column of the index\_advisor\_log table with the new OIDs of the referenced tables using the SQL UPDATE statement:

UPDATE index\_advisor\_log SET reloid = new\_oid WHERE reloid = old\_oid;

## SQL Profiler

Inefficient SQL code is one of, if not the leading cause of database performance problems. The challenge for database administrators and developers is locating and then optimizing this code in large, complex systems.

SQL Profiler helps you locate and optimize poorly running SQL code.

Specific features and benefits of SQL Profiler include the following:

* On-Demand Traces. You can capture SQL traces at any time by manually setting up your parameters and starting the trace.
* Scheduled Traces. For inconvenient times, you can also specify your trace parameters and schedule them to run at some later time.
* Save Traces. Execute your traces and save them for later review.
* Trace Filters. Selectively filter SQL captures by database and by user, or capture every SQL statement sent by all users against all databases.
* Trace Output Analyzer. A graphical table lets you quickly sort and filter queries by duration or statement, and a graphical or text based EXPLAIN plan lays out your query paths and joins.
* Index Advisor Integration. Once you have found your slow queries and optimized them, you can also let the Index Advisor recommend the creation of underlying table indices to further improve performance.

For more information about SQL Profiler and Postgres Enterprise Manager, visit the EnterpriseDB website at:

[http://www.enterprisedb.com/postgres-enterprise-manager](http://www.enterprisedb.com/products-services-training/products/postgres-enterprise-manager)

## Query Optimization Hints

When you invoke a DELETE, INSERT, SELECT or UPDATE command, the server generates a set of execution plans; after analyzing those execution plans, the server selects a plan that will (generally) return the result set in the least amount of time. The server's choice of plan is dependent upon several factors:

* The estimated execution cost of data handling operations.
* Parameter values assigned to parameters in the Query Tuning section of the postgresql.conf file.
* Column statistics that have been gathered by the [ANALYZE](http://www.enterprisedb.com/docs/en/9.3/pg/sql-analyze.html) command.

As a rule, the query planner will select the least expensive plan. You can use an optimizer hint to influence the server as it selects a query plan. An optimizer hint is a directive (or multiple directives) embedded in a comment-like syntax that immediately follows a DELETE, INSERT, SELECT or UPDATE command. Keywords in the comment instruct the server to employ or avoid a specific plan when producing the result set.

Synopsis

{ DELETE | INSERT | SELECT | UPDATE } /\*+ { hint [ comment ] } [...] \*/

statement\_body

{ DELETE | INSERT | SELECT | UPDATE } --+ { hint [ comment ] } [...]

statement\_body

Optimizer hints may be included in either of the forms shown above. Note that in both forms, a plus sign (+) must immediately follow the /\* or -- opening comment symbols, with no intervening space, or the server will not interpret the following tokens as hints.

If you are using the first form, the hint and optional comment may span multiple lines. The second form requires all hints and comments to occupy a single line; the remainder of the statement must start on a new line.

Description

Please Note:

* The database server will always try to use the specified hints if at all possible.
* If a planner method parameter is set so as to disable a certain plan type, then this plan will not be used even if it is specified in a hint, unless there are no other possible options for the planner. Examples of planner method parameters are enable\_indexscan, enable\_seqscan, enable\_hashjoin, enable\_mergejoin, and enable\_nestloop. These are all Boolean parameters.
* Remember that the hint is embedded within a comment. As a consequence, if the hint is misspelled or if any parameter to a hint such as view, table, or column name is misspelled, or non-existent in the SQL command, there will be no indication that any sort of error has occurred. No syntax error will be given and the entire hint is simply ignored.
* If an alias is used for a table or view name in the SQL command, then the alias name, not the original object name, must be used in the hint. For example, in the command, SELECT /\*+ FULL(acct) \*/ \* FROM accounts acct ..., acct, the alias for accounts, must be specified in the FULL hint, not the table name, accounts.
* Use the EXPLAIN command to ensure that the hint is correctly formed and the planner is using the hint. See the EDB Postgres documentation set for information on the EXPLAIN command.
* In general, optimizer hints should not be used in production applications. Typically, the table data changes throughout the life of the application. By ensuring that the more dynamic columns are ANALYZEd frequently, the column statistics will be updated to reflect value changes and the planner will use such information to produce the least cost plan for any given command execution. Use of optimizer hints defeats the purpose of this process and will result in the same plan regardless of how the table data changes.

Parameters

hint

An optimizer hint directive.

comment

A string with additional information. Note that there are restrictions as to what characters may be included in the comment. Generally, comment may only consist of alphabetic, numeric, the underscore, dollar sign, number sign and space characters. These must also conform to the syntax of an identifier. Any subsequent hint will be ignored if the comment is not in this form.

statement\_body

The remainder of the DELETE, INSERT, SELECT, or UPDATE command.

The following sections describe the optimizer hint directives in more detail.

### Default Optimization Modes

There are a number of optimization modes that can be chosen as the default setting for an Advanced Server database cluster. This setting can also be changed on a per session basis by using the ALTER SESSION command as well as in individual DELETE, SELECT, and UPDATE commands within an optimizer hint. The configuration parameter that controls these default modes is named OPTIMIZER\_MODE. The following table shows the possible values.

Table 3‑8‑1 Default Optimization Modes

| Hint | Description |
| --- | --- |
| ALL\_ROWS | Optimizes for retrieval of all rows of the result set. |
| CHOOSE | Does no default optimization based on assumed number of rows to be retrieved from the result set. This is the default. |
| FIRST\_ROWS | Optimizes for retrieval of only the first row of the result set. |
| FIRST\_ROWS\_10 | Optimizes for retrieval of the first 10 rows of the results set. |
| FIRST\_ROWS\_100 | Optimizes for retrieval of the first 100 rows of the result set. |
| FIRST\_ROWS\_1000 | Optimizes for retrieval of the first 1000 rows of the result set. |
| FIRST\_ROWS(n) | Optimizes for retrieval of the first n rows of the result set. This form may not be used as the object of the ALTER SESSION SET OPTIMIZER\_MODE command. It may only be used in the form of a hint in a SQL command. |

These optimization modes are based upon the assumption that the client submitting the SQL command is interested in viewing only the first “n” rows of the result set and will then abandon the remainder of the result set. Resources allocated to the query are adjusted as such.

Examples

Alter the current session to optimize for retrieval of the first 10 rows of the result set.

ALTER SESSION SET OPTIMIZER\_MODE = FIRST\_ROWS\_10;

The current value of the OPTIMIZER\_MODE parameter can be shown by using the SHOW command. Note that this command is a utility dependent command. In PSQL, the SHOW command is used as follows:

SHOW OPTIMIZER\_MODE;

optimizer\_mode

----------------

first\_rows\_10

(1 row)

The SHOW command has the following syntax:

SHOW PARAMETER OPTIMIZER\_MODE;

NAME

--------------------------------------------------

VALUE

--------------------------------------------------

optimizer\_mode

first\_rows\_10

The following example shows an optimization mode used in a SELECT command as a hint:

SELECT /\*+ FIRST\_ROWS(7) \*/ \* FROM emp;

empno | ename | job | mgr | hiredate | sal | comm | deptno

-------+--------+-----------+------+--------------------+---------+---------+--------

7369 | SMITH | CLERK | 7902 | 17-DEC-80 00:00:00 | 800.00 | | 20

7499 | ALLEN | SALESMAN | 7698 | 20-FEB-81 00:00:00 | 1600.00 | 300.00 | 30

7521 | WARD | SALESMAN | 7698 | 22-FEB-81 00:00:00 | 1250.00 | 500.00 | 30

7566 | JONES | MANAGER | 7839 | 02-APR-81 00:00:00 | 2975.00 | | 20

7654 | MARTIN | SALESMAN | 7698 | 28-SEP-81 00:00:00 | 1250.00 | 1400.00 | 30

7698 | BLAKE | MANAGER | 7839 | 01-MAY-81 00:00:00 | 2850.00 | | 30

7782 | CLARK | MANAGER | 7839 | 09-JUN-81 00:00:00 | 2450.00 | | 10

7788 | SCOTT | ANALYST | 7566 | 19-APR-87 00:00:00 | 3000.00 | | 20

7839 | KING | PRESIDENT | | 17-NOV-81 00:00:00 | 5000.00 | | 10

7844 | TURNER | SALESMAN | 7698 | 08-SEP-81 00:00:00 | 1500.00 | 0.00 | 30

7876 | ADAMS | CLERK | 7788 | 23-MAY-87 00:00:00 | 1100.00 | | 20

7900 | JAMES | CLERK | 7698 | 03-DEC-81 00:00:00 | 950.00 | | 30

7902 | FORD | ANALYST | 7566 | 03-DEC-81 00:00:00 | 3000.00 | | 20

7934 | MILLER | CLERK | 7782 | 23-JAN-82 00:00:00 | 1300.00 | | 10

(14 rows)

### Access Method Hints

The following hints influence how the optimizer accesses relations to create the result set.

Table 3‑8‑2 Access Method Hints

| Hint | Description |
| --- | --- |
| FULL(table) | Perform a full sequential scan on table. |
| INDEX(table [ index ] [...]) | Use index on table to access the relation. |
| NO\_INDEX(table [ index ] [...]) | Do not use index on table to access the relation. |

In addition, the ALL\_ROWS, FIRST\_ROWS, and FIRST\_ROWS(n) hints of Table 3‑8‑1 can be used.

Examples

The sample application does not have sufficient data to illustrate the effects of optimizer hints so the remainder of the examples in this section will use a banking database created by the pgbench application located in the PostgresPlus\9.5AS\bin subdirectory.

The following steps create a database named, bank, populated by the tables, accounts, branches, tellers, and history. The –s 5 option specifies a scaling factor of five which results in the creation of five branches, each with 100,000 accounts, resulting in a total of 500,000 rows in the accounts table and five rows in the branches table. Ten tellers are assigned to each branch resulting in a total of 50 rows in the tellers table.

Note, if using Linux use the export command instead of the SET PATH command as shown below.

export PATH=/opt/PostgresPlus/9.5AS/bin:$PATH

The following example was run in Windows.

SET PATH=C:\PostgresPlus\9.5AS\bin;%PATH%

createdb -U enterprisedb bank

CREATE DATABASE

pgbench -i -s 5 -U enterprisedb -d bank

creating tables...

10000 tuples done.

20000 tuples done.

30000 tuples done.

.

.

.

470000 tuples done.

480000 tuples done.

490000 tuples done.

500000 tuples done.

set primary key...

vacuum...done.

Ten transactions per client are then processed for eight clients for a total of 80 transactions. This will populate the history table with 80 rows.

pgbench –U enterprisedb –d bank –c 8 –t 10

.

.

.

transaction type: TPC-B (sort of)

scaling factor: 5

number of clients: 8

number of transactions per client: 10

number of transactions actually processed: 80/80

tps = 6.023189 (including connections establishing)

tps = 7.140944 (excluding connections establishing)

The table definitions are shown below:

\d accounts

Table "public.accounts"

Column | Type | Modifiers

----------+---------------+-----------

aid | integer | not null

bid | integer |

abalance | integer |

filler | character(84) |

Indexes:

"accounts\_pkey" PRIMARY KEY, btree (aid)

\d branches

Table "public.branches"

Column | Type | Modifiers

----------+---------------+-----------

bid | integer | not null

bbalance | integer |

filler | character(88) |

Indexes:

"branches\_pkey" PRIMARY KEY, btree (bid)

\d tellers

Table "public.tellers"

Column | Type | Modifiers

----------+---------------+-----------

tid | integer | not null

bid | integer |

tbalance | integer |

filler | character(84) |

Indexes:

"tellers\_pkey" PRIMARY KEY, btree (tid)

\d history

Table "public.history"

Column | Type | Modifiers

--------+-----------------------------+-----------

tid | integer |

bid | integer |

aid | integer |

delta | integer |

mtime | timestamp without time zone |

filler | character(22) |

The EXPLAIN command shows the plan selected by the query planner. In the following example, aid is the primary key column, so an indexed search is used on index, accounts\_pkey.

EXPLAIN SELECT \* FROM accounts WHERE aid = 100;

QUERY PLAN

-------------------------------------------------------------------------------

Index Scan using accounts\_pkey on accounts (cost=0.00..8.32 rows=1 width=97)

Index Cond: (aid = 100)

(2 rows)

The FULL hint is used to force a full sequential scan instead of using the index as shown below:

EXPLAIN SELECT /\*+ FULL(accounts) \*/ \* FROM accounts WHERE aid = 100;

QUERY PLAN

-------------------------------------------------------------

Seq Scan on accounts (cost=0.00..14461.10 rows=1 width=97)

Filter: (aid = 100)

(2 rows)

The NO\_INDEX hint also forces a sequential scan as shown below:

EXPLAIN SELECT /\*+ NO\_INDEX(accounts accounts\_pkey) \*/ \* FROM accounts WHERE aid = 100;

QUERY PLAN

-------------------------------------------------------------

Seq Scan on accounts (cost=0.00..14461.10 rows=1 width=97)

Filter: (aid = 100)

(2 rows)

In addition to using the EXPLAIN command as shown in the prior examples, more detailed information regarding whether or not a hint was used by the planner can be obtained by setting the client\_min\_messages and trace\_hints configuration parameters as follows:

SET client\_min\_messages TO info;

SET trace\_hints TO true;

The SELECT command with the NO\_INDEX hint is repeated below to illustrate the additional information produced when the aforementioned configuration parameters are set.

EXPLAIN SELECT /\*+ NO\_INDEX(accounts accounts\_pkey) \*/ \* FROM accounts WHERE aid = 100;

INFO: [HINTS] Index Scan of [accounts].[accounts\_pkey] rejected because of NO\_INDEX hint.

INFO: [HINTS] Bitmap Heap Scan of [accounts].[accounts\_pkey] rejected because of NO\_INDEX hint.

QUERY PLAN

-------------------------------------------------------------

Seq Scan on accounts (cost=0.00..14461.10 rows=1 width=97)

Filter: (aid = 100)

(2 rows)

Note that if a hint is ignored, the INFO: [HINTS] line will not appear. This may be an indication that there was a syntax error or some other misspelling in the hint as shown in the following example where the index name is misspelled.

EXPLAIN SELECT /\*+ NO\_INDEX(accounts accounts\_xxx) \*/ \* FROM accounts WHERE aid = 100;

QUERY PLAN

-------------------------------------------------------------------------------

Index Scan using accounts\_pkey on accounts (cost=0.00..8.32 rows=1

width=97)

Index Cond: (aid = 100)

(2 rows

### Specifying a Join Order

Include the ORDERED directive to instruct the query optimizer to join tables in the order in which they are listed in the FROM clause. If you do not include the ORDERED keyword, the query optimizer will choose the order in which to join the tables.

For example, the following command allows the optimizer to choose the order in which to join the tables listed in the FROM clause:

SELECT e.ename, d.dname, h.startdate  
 FROM emp e, dept d, jobhist h  
 WHERE d.deptno = e.deptno  
 AND h.empno = e.empno;

The following command instructs the optimizer to join the tables in the ordered specified:

SELECT /\*+ ORDERED \*/ e.ename, d.dname, h.startdate  
 FROM emp e, dept d, jobhist h  
 WHERE d.deptno = e.deptno  
 AND h.empno = e.empno;

In the ORDERED version of the command, Advanced Server will first join emp e with dept d before joining the results with jobhist h. Without the ORDERED directive, the join order is selected by the query optimizer.

Please note: the ORDERED directive does not work for outer joins that contain a '+' sign.

### Joining Relations Hints

When two tables are to be joined, there are three possible plans that may be used to perform the join.

* Nested Loop Join – The right table is scanned once for every row in the left table.
* Merge Sort Join – Each table is sorted on the join attributes before the join starts. The two tables are then scanned in parallel and the matching rows are combined to form the join rows.
* Hash Join – The right table is scanned and its join attributes are loaded into a hash table using its join attributes as hash keys. The left table is then scanned and its join attributes are used as hash keys to locate the matching rows from the right table.

The following table lists the optimizer hints that can be used to influence the planner to use one type of join plan over another.

Table 3‑8‑3 Join Hints

| Hint | Description |
| --- | --- |
| USE\_HASH(table [...]) | Use a hash join with a hash table created from the join attributes of table. |
| NO\_USE\_HASH(table [...]) | Do not use a hash join created from the join attributes of table. |
| USE\_MERGE(table [...]) | Use a merge sort join for table. |
| NO\_USE\_MERGE(table [...]) | Do not use a merge sort join for table. |
| USE\_NL(table [...]) | Use a nested loop join for table. |
| NO\_USE\_NL(table [...]) | Do not use a nested loop join for table. |

Examples

In the following example, a join is performed on the branches and accounts tables. The query plan shows that a hash join is used by creating a hash table from the join attribute of the branches table.

EXPLAIN SELECT b.bid, a.aid, abalance FROM branches b, accounts a WHERE b.bid = a.bid;

QUERY PLAN

--------------------------------------------------------------------------

Hash Join (cost=1.11..20092.70 rows=500488 width=12)

Hash Cond: (a.bid = b.bid)

-> Seq Scan on accounts a (cost=0.00..13209.88 rows=500488 width=12)

-> Hash (cost=1.05..1.05 rows=5 width=4)

-> Seq Scan on branches b (cost=0.00..1.05 rows=5 width=4)

(5 rows)

By using the USE\_HASH(a) hint, the planner is forced to create the hash table from the accounts join attribute instead of from the branches table. Note the use of the alias, a, for the accounts table in the USE\_HASH hint.

EXPLAIN SELECT /\*+ USE\_HASH(a) \*/ b.bid, a.aid, abalance FROM branches b, accounts a WHERE b.bid = a.bid;

QUERY PLAN

--------------------------------------------------------------------------------

Hash Join (cost=21909.98..30011.52 rows=500488 width=12)

Hash Cond: (b.bid = a.bid)

-> Seq Scan on branches b (cost=0.00..1.05 rows=5 width=4)

-> Hash (cost=13209.88..13209.88 rows=500488 width=12)

-> Seq Scan on accounts a (cost=0.00..13209.88 rows=500488 width=12)

(5 rows)

Next, the NO\_USE\_HASH(a b) hint forces the planner to use an approach other than hash tables. The result is a nested loop.

EXPLAIN SELECT /\*+ NO\_USE\_HASH(a b) \*/ b.bid, a.aid, abalance FROM branches b, accounts a WHERE b.bid = a.bid;

QUERY PLAN

--------------------------------------------------------------------------

Nested Loop (cost=1.05..69515.84 rows=500488 width=12)

Join Filter: (b.bid = a.bid)

-> Seq Scan on accounts a (cost=0.00..13209.88 rows=500488 width=12)

-> Materialize (cost=1.05..1.11 rows=5 width=4)

-> Seq Scan on branches b (cost=0.00..1.05 rows=5 width=4)

(5 rows)

Finally, the USE\_MERGE hint forces the planner to use a merge join.

EXPLAIN SELECT /\*+ USE\_MERGE(a) \*/ b.bid, a.aid, abalance FROM branches b, accounts a WHERE b.bid = a.bid;

QUERY PLAN

--------------------------------------------------------------------------------

Merge Join (cost=69143.62..76650.97 rows=500488 width=12)

Merge Cond: (b.bid = a.bid)

-> Sort (cost=1.11..1.12 rows=5 width=4)

Sort Key: b.bid

-> Seq Scan on branches b (cost=0.00..1.05 rows=5 width=4)

-> Sort (cost=69142.52..70393.74 rows=500488 width=12)

Sort Key: a.bid

-> Seq Scan on accounts a (cost=0.00..13209.88 rows=500488 width=12)

(8 rows)

In this three-table join example, the planner first performs a hash join on the branches and history tables, then finally performs a nested loop join of the result with the accounts\_pkey index of the accounts table.

EXPLAIN SELECT h.mtime, h.delta, b.bid, a.aid FROM history h, branches b, accounts a WHERE h.bid = b.bid AND h.aid = a.aid;

QUERY PLAN

--------------------------------------------------------------------------------------

Nested Loop (cost=1.11..207.95 rows=26 width=20)

-> Hash Join (cost=1.11..25.40 rows=26 width=20)

Hash Cond: (h.bid = b.bid)

-> Seq Scan on history h (cost=0.00..20.20 rows=1020 width=20)

-> Hash (cost=1.05..1.05 rows=5 width=4)

-> Seq Scan on branches b (cost=0.00..1.05 rows=5 width=4)

-> Index Scan using accounts\_pkey on accounts a (cost=0.00..7.01 rows=1

width=4)

Index Cond: (h.aid = a.aid)

(8 rows)

This plan is altered by using hints to force a combination of a merge sort join and a hash join.

EXPLAIN SELECT /\*+ USE\_MERGE(h b) USE\_HASH(a) \*/ h.mtime, h.delta, b.bid, a.aid FROM history h, branches b, accounts a WHERE h.bid = b.bid AND h.aid = a.aid;

QUERY PLAN

-------------------------------------------------------------------------------------------

Merge Join (cost=23480.11..23485.60 rows=26 width=20)

Merge Cond: (h.bid = b.bid)

-> Sort (cost=23479.00..23481.55 rows=1020 width=20)

Sort Key: h.bid

-> Hash Join (cost=21421.98..23428.03 rows=1020 width=20)

Hash Cond: (h.aid = a.aid)

-> Seq Scan on history h (cost=0.00..20.20 rows=1020

width=20)

-> Hash (cost=13209.88..13209.88 rows=500488 width=4)

-> Seq Scan on accounts a (cost=0.00..13209.88

rows=500488 width=4)

-> Sort (cost=1.11..1.12 rows=5 width=4)

Sort Key: b.bid

-> Seq Scan on branches b (cost=0.00..1.05 rows=5 width=4)

(12 rows)

### Global Hints

Thus far, hints have been applied directly to tables that are referenced in the SQL command. It is also possible to apply hints to tables that appear in a view when the view is referenced in the SQL command. The hint does not appear in the view, itself, but rather in the SQL command that references the view.

When specifying a hint that is to apply to a table within a view, the view and table names are given in dot notation within the hint argument list.

Synopsis

hint(view.table)

Parameters

hint

Any of the hints in Table 3‑8‑2 or Table 3‑8‑3.

view

The name of the view containing table.

table

The table on which the hint is to be applied.

Examples

A view named, tx, is created from the three-table join of history, branches, and accounts shown in the final example of Section 8.5.3.

CREATE VIEW tx AS SELECT h.mtime, h.delta, b.bid, a.aid FROM history h, branches b, accounts a WHERE h.bid = b.bid AND h.aid = a.aid;

The query plan produced by selecting from this view is show below:

EXPLAIN SELECT \* FROM tx;

QUERY PLAN

--------------------------------------------------------------------------------------

Nested Loop (cost=1.11..207.95 rows=26 width=20)

-> Hash Join (cost=1.11..25.40 rows=26 width=20)

Hash Cond: (h.bid = b.bid)

-> Seq Scan on history h (cost=0.00..20.20 rows=1020 width=20)

-> Hash (cost=1.05..1.05 rows=5 width=4)

-> Seq Scan on branches b (cost=0.00..1.05 rows=5 width=4)

-> Index Scan using accounts\_pkey on accounts a (cost=0.00..7.01 rows=1

width=4)

Index Cond: (h.aid = a.aid)

(8 rows)

The same hints that were applied to this join at the end of Section 8.5.3 can be applied to the view as follows:

EXPLAIN SELECT /\*+ USE\_MERGE(tx.h tx.b) USE\_HASH(tx.a) \*/ \* FROM tx;

QUERY PLAN

------------------------------------------------------------------------------------------

-

Merge Join (cost=23480.11..23485.60 rows=26 width=20)

Merge Cond: (h.bid = b.bid)

-> Sort (cost=23479.00..23481.55 rows=1020 width=20)

Sort Key: h.bid

-> Hash Join (cost=21421.98..23428.03 rows=1020 width=20)

Hash Cond: (h.aid = a.aid)

-> Seq Scan on history h (cost=0.00..20.20 rows=1020

width=20)

-> Hash (cost=13209.88..13209.88 rows=500488 width=4)

-> Seq Scan on accounts a (cost=0.00..13209.88

rows=500488 width=4)

-> Sort (cost=1.11..1.12 rows=5 width=4)

Sort Key: b.bid

-> Seq Scan on branches b (cost=0.00..1.05 rows=5 width=4)

(12 rows)

In addition to applying hints to tables within stored views, hints can be applied to tables within subqueries as illustrated by the following example. In this query on the sample application emp table, employees and their managers are listed by joining the emp table with a subquery of the emp table identified by the alias, b.

SELECT a.empno, a.ename, b.empno "mgr empno", b.ename "mgr ename" FROM emp a, (SELECT \* FROM emp) b WHERE a.mgr = b.empno;

empno | ename | mgr empno | mgr ename

-------+--------+-----------+-----------

7902 | FORD | 7566 | JONES

7788 | SCOTT | 7566 | JONES

7521 | WARD | 7698 | BLAKE

7844 | TURNER | 7698 | BLAKE

7654 | MARTIN | 7698 | BLAKE

7900 | JAMES | 7698 | BLAKE

7499 | ALLEN | 7698 | BLAKE

7934 | MILLER | 7782 | CLARK

7876 | ADAMS | 7788 | SCOTT

7782 | CLARK | 7839 | KING

7698 | BLAKE | 7839 | KING

7566 | JONES | 7839 | KING

7369 | SMITH | 7902 | FORD

(13 rows)

The plan chosen by the query planner is shown below:

EXPLAIN SELECT a.empno, a.ename, b.empno "mgr empno", b.ename "mgr ename" FROM emp a, (SELECT \* FROM emp) b WHERE a.mgr = b.empno;

QUERY PLAN

-------------------------------------------------------------------

Merge Join (cost=2.81..3.08 rows=13 width=26)

Merge Cond: (a.mgr = emp.empno)

-> Sort (cost=1.41..1.44 rows=14 width=20)

Sort Key: a.mgr

-> Seq Scan on emp a (cost=0.00..1.14 rows=14 width=20)

-> Sort (cost=1.41..1.44 rows=14 width=13)

Sort Key: emp.empno

-> Seq Scan on emp (cost=0.00..1.14 rows=14 width=13)

(8 rows)

A hint can be applied to the emp table within the subquery to perform an index scan on index, emp\_pk, instead of a table scan. Note the difference in the query plans.

EXPLAIN SELECT /\*+ INDEX(b.emp emp\_pk) \*/ a.empno, a.ename, b.empno "mgr empno", b.ename "mgr ename" FROM emp a, (SELECT \* FROM emp) b WHERE a.mgr = b.empno;

QUERY PLAN

---------------------------------------------------------------------------

Merge Join (cost=1.41..13.21 rows=13 width=26)

Merge Cond: (a.mgr = emp.empno)

-> Sort (cost=1.41..1.44 rows=14 width=20)

Sort Key: a.mgr

-> Seq Scan on emp a (cost=0.00..1.14 rows=14 width=20)

-> Index Scan using emp\_pk on emp (cost=0.00..12.46 rows=14 width=13)

(6 rows)

### Using the APPEND Optimizer Hint

By default, Advanced Server will add new data into the first available free-space in a table (vacated by vacuumed records). Include the APPEND directive after an INSERT or SELECT command to instruct the server to bypass mid-table free space, and affix new rows to the end of the table. This optimizer hint can be particularly useful when bulk loading data.

The syntax is:

/\*+APPEND\*/

For example, the following command instructs the server to append the data in the INSERT statement to the end of the sales table:

INSERT /\*+APPEND\*/ INTO sales VALUES  
(10, 10, '01-Mar-2011', 10, 'OR');

Note that Advanced Server supports the APPEND hint when adding multiple rows in a single INSERT statement:

INSERT /\*+APPEND\*/ INTO sales VALUES  
(20, 20, '01-Aug-2011', 20, 'NY'),  
(30, 30, '01-Feb-2011', 30, 'FL'),  
(40, 40, '01-Nov-2011', 40, 'TX');

The APPEND hint can also be included in the SELECT clause of an INSERT INTO statement:

INSERT INTO sales\_history SELECT /\*+APPEND\*/ FROM sales;

### Conflicting Hints

If a command includes two or more conflicting hints, the server will ignore the contradictory hints. The following table lists hints that are contradictory to each other.

Table 3‑8‑4 Conflicting Hints

| Hint | Conflicting Hint |
| --- | --- |
| ALL\_ROWS | FIRST\_ROWS - all formats |
| FULL(table) | INDEX(table [ index ]) |
| INDEX(table) | FULL(table)  NO\_INDEX(table) |
| INDEX(table index) | FULL(table)  NO\_INDEX(table index) |
| USE\_HASH(table) | NO\_USE\_HASH(table) |
| USE\_MERGE(table) | NO\_USE\_MERGE(table) |
| USE\_NL(table) | NO\_USE\_NL(table) |

## DBMS\_PROFILER

The DBMS\_PROFILER package collects and stores performance information about the PL/pgSQL and SPL statements that are executed during a profiling session; you can review the performance information in the tables and views provided by the profiler.

DBMS\_PROFILER works by recording a set of performance-related counters and timers for each line of PL/pgSQL or SPL statement that executes within a profiling session. The counters and timers are stored in a table named SYS.PLSQL\_PROFILER\_DATA. When you complete a profiling session, DBMS\_PROFILER will write a row to the performance statistics table for each line of PL/pgSQL or SPL code that executed within the session. For example, if you execute the following function:

1 - CREATE OR REPLACE FUNCTION getBalance(acctNumber INTEGER)

2 - RETURNS NUMERIC AS $$

3 - DECLARE

4 - result NUMERIC;

5 - BEGIN

6 - SELECT INTO result balance FROM acct WHERE id = acctNumber;

7 -

8 - IF (result IS NULL) THEN

9 - RAISE INFO 'Balance is null';

10- END IF;

11-

12- RETURN result;

13- END;

14- $$ LANGUAGE 'plpgsql';

DBMS\_PROFILER adds one PLSQL\_PROFILER\_DATA entry for each line of code within the getBalance() function (including blank lines and comments). The entry corresponding to the SELECT statement executed exactly one time; and required a very small amount of time to execute. On the other hand, the entry corresponding to the RAISE INFO statement executed once or not at all (depending on the value for the balance column).

Some of the lines in this function contain no executable code so the performance statistics for those lines will always contain zero values.

To start a profiling session, invoke the DBMS\_PROFILER.START\_PROFILER function (or procedure). Once you've invoked START\_PROFILER, Advanced Server will profile every PL/pgSQL or SPL function, procedure, trigger, or anonymous block that your session executes until you either stop or pause the profiler (by calling STOP\_PROFILER or PAUSE\_PROFILER).

It is important to note that when you start (or resume) the profiler, the profiler will only gather performance statistics for functions/procedures/triggers that start after the call to START\_PROFILER (or RESUME\_PROFILER).

While the profiler is active, Advanced Server records a large set of timers and counters in memory; when you invoke the STOP\_PROFILER (or FLUSH\_DATA) function/procedure, DBMS\_PROFILER writes those timers and counters to a set of three tables:

* SYS.PLSQL\_PROFILER\_RAWDATA  
  Contains the performance counters and timers for each statement executed within the session.
* SYS.PLSQL\_PROFILER\_RUNS  
  Contains a summary of each run (aggregating the information found in PLSQL\_PROFILER\_RAWDATA).
* SYS.PLSQL\_PROFILER\_UNITS  
  Contains a summary of each code unit (function, procedure, trigger, or anonymous block) executed within a session.

In addition, DBMS\_PROFILER defines a view, SYS.PLSQL\_PROFILER\_DATA, which contains a subset of the PLSQL\_PROFILER\_RAWDATA table.

Please note that a non-superuser may gather profiling information, but may not view that profiling information unless a superuser grants specific privileges on the profiling tables (stored in the SYS schema). This permits a non-privileged user to gather performance statistics without exposing information that the administrator may want to keep secret.

### Querying the DBMS\_PROFILER Tables and View

The following step-by-step example uses DBMS\_PROFILER to retrieve performance information for procedures, functions, and triggers included in the sample data distributed with Advanced Server.

1. Open the EDB-PSQL command line, and establish a connection to the Advanced Server database. Use an EXEC statement to start the profiling session:

acctg=# EXEC dbms\_profiler.start\_profiler('profile list\_emp');

EDB-SPL Procedure successfully completed

(Note: the call to start\_profiler() includes a comment that DBMS\_PROFILER associates with the profiler session).

1. Then call the list\_emp function:

acctg=# SELECT list\_emp();

INFO: EMPNO ENAME

INFO: ----- -------

INFO: 7369 SMITH

INFO: 7499 ALLEN

INFO: 7521 WARD

INFO: 7566 JONES

INFO: 7654 MARTIN

INFO: 7698 BLAKE

INFO: 7782 CLARK

INFO: 7788 SCOTT

INFO: 7839 KING

INFO: 7844 TURNER

INFO: 7876 ADAMS

INFO: 7900 JAMES

INFO: 7902 FORD

INFO: 7934 MILLER

list\_emp

----------

(1 row)

1. Stop the profiling session with a call to dbms\_profiler.stop\_profiler:

acctg=# EXEC dbms\_profiler.stop\_profiler;

EDB-SPL Procedure successfully completed

1. Start a new session with the dbms\_profiler.start\_profiler function (followed by a new comment):

acctg=# EXEC dbms\_profiler.start\_profiler('profile get\_dept\_name and emp\_sal\_trig');

EDB-SPL Procedure successfully completed

1. Invoke the get\_dept\_name function:

acctg=# SELECT get\_dept\_name(10);

get\_dept\_name

---------------

ACCOUNTING

(1 row)

1. Execute an UPDATE statement that causes a trigger to execute:

acctg=# UPDATE memp SET sal = 500 WHERE empno = 7902;

INFO: Updating employee 7902

INFO: ..Old salary: 3000.00

INFO: ..New salary: 500.00

INFO: ..Raise : -2500.00

INFO: User enterprisedb updated employee(s) on 04-FEB-14

UPDATE 1

1. Terminate the profiling session and flush the performance information to the profiling tables:

acctg=# EXEC dbms\_profiler.stop\_profiler;

EDB-SPL Procedure successfully completed

1. Now, query the plsql\_profiler\_runs table to view a list of the profiling sessions, arranged by runid:

acctg=# SELECT \* FROM plsql\_profiler\_runs;

runid | related\_run | run\_owner | run\_date | run\_comment | run\_total\_time | run\_system\_info | run\_comment1 | spare1

-------+-------------+--------------+---------------------------+----------------------------------------+----------------+-----------------+--------------+--------

1 | | enterprisedb | 04-FEB-14 09:32:48.874315 | profile list\_emp | 4154 | | |

2 | | enterprisedb | 04-FEB-14 09:41:30.546503 | profile get\_dept\_name and emp\_sal\_trig | 2088 | | |

(2 rows)

1. Query the plsql\_profiler\_units table to view the amount of time consumed by each unit (each function, procedure, or trigger):

acctg=# SELECT \* FROM plsql\_profiler\_units;

runid | unit\_number | unit\_type | unit\_owner | unit\_name | unit\_timestamp | total\_time | spare1 | spare2

-------+-------------+-----------+--------------+---------------------------------+----------------+------------+--------+--------

1 | 16999 | FUNCTION | enterprisedb | list\_emp() | | 4 | |

2 | 17002 | FUNCTION | enterprisedb | user\_audit\_trig() | | 1 | |

2 | 17000 | FUNCTION | enterprisedb | get\_dept\_name(p\_deptno numeric) | | 1 | |

2 | 17004 | FUNCTION | enterprisedb | emp\_sal\_trig() | | 1 | |

(4 rows)

1. Query the plsql\_profiler\_rawdata table to view a list of the wait event counters and wait event times:

acctg=# SELECT runid, sourcecode, func\_oid, line\_number, exec\_count, tuples\_returned, time\_total FROM plsql\_profiler\_rawdata;

runid | sourcecode | func\_oid | line\_number | exec\_count | tuples\_returned | time\_total

-------+------------------------------------------------------------------+----------+-------------+------------+-----------------+------------

1 | DECLARE | 16999 | 1 | 0 | 0 | 0

1 | v\_empno NUMERIC(4); | 16999 | 2 | 0 | 0 | 0

1 | v\_ename VARCHAR(10); | 16999 | 3 | 0 | 0 | 0

1 | emp\_cur CURSOR FOR | 16999 | 4 | 0 | 0 | 0

1 | SELECT empno, ename FROM memp ORDER BY empno; | 16999 | 5 | 0 | 0 | 0

1 | BEGIN | 16999 | 6 | 0 | 0 | 0

1 | OPEN emp\_cur; | 16999 | 7 | 0 | 0 | 0

1 | RAISE INFO 'EMPNO ENAME'; | 16999 | 8 | 1 | 0 | 0.001621

1 | RAISE INFO '----- -------'; | 16999 | 9 | 1 | 0 | 0.000301

1 | LOOP | 16999 | 10 | 1 | 0 | 4.6e-05

1 | FETCH emp\_cur INTO v\_empno, v\_ename; | 16999 | 11 | 1 | 0 | 0.001114

1 | EXIT WHEN NOT FOUND; | 16999 | 12 | 15 | 0 | 0.000206

1 | RAISE INFO '% %', v\_empno, v\_ename; | 16999 | 13 | 15 | 0 | 8.3e-05

1 | END LOOP; | 16999 | 14 | 14 | 0 | 0.000773

1 | CLOSE emp\_cur; | 16999 | 15 | 0 | 0 | 0

1 | RETURN; | 16999 | 16 | 1 | 0 | 1e-05

1 | END; | 16999 | 17 | 1 | 0 | 0

1 | | 16999 | 18 | 0 | 0 | 0

2 | DECLARE | 17002 | 1 | 0 | 0 | 0

2 | v\_action VARCHAR(24); | 17002 | 2 | 0 | 0 | 0

2 | v\_text TEXT; | 17002 | 3 | 0 | 0 | 0

2 | BEGIN | 17002 | 4 | 0 | 0 | 0

2 | IF TG\_OP = 'INSERT' THEN | 17002 | 5 | 0 | 0 | 0

2 | v\_action := ' added employee(s) on '; | 17002 | 6 | 1 | 0 | 0.000143

2 | ELSIF TG\_OP = 'UPDATE' THEN | 17002 | 7 | 0 | 0 | 0

2 | v\_action := ' updated employee(s) on '; | 17002 | 8 | 0 | 0 | 0

2 | ELSIF TG\_OP = 'DELETE' THEN | 17002 | 9 | 1 | 0 | 3.2e-05

2 | v\_action := ' deleted employee(s) on '; | 17002 | 10 | 0 | 0 | 0

2 | END IF; | 17002 | 11 | 0 | 0 | 0

2 | v\_text := 'User ' || USER || v\_action || CURRENT\_DATE; | 17002 | 12 | 0 | 0 | 0

2 | RAISE INFO ' %', v\_text; | 17002 | 13 | 1 | 0 | 0.000383

2 | RETURN NULL; | 17002 | 14 | 1 | 0 | 6.3e-05

2 | END; | 17002 | 15 | 1 | 0 | 3.6e-05

2 | | 17002 | 16 | 0 | 0 | 0

2 | DECLARE | 17000 | 1 | 0 | 0 | 0

2 | v\_dname VARCHAR(14); | 17000 | 2 | 0 | 0 | 0

2 | BEGIN | 17000 | 3 | 0 | 0 | 0

2 | SELECT INTO v\_dname dname FROM dept WHERE deptno = p\_deptno; | 17000 | 4 | 0 | 0 | 0

2 | RETURN v\_dname; | 17000 | 5 | 1 | 0 | 0.000647

2 | IF NOT FOUND THEN | 17000 | 6 | 1 | 0 | 2.6e-05

2 | RAISE INFO 'Invalid department number %', p\_deptno; | 17000 | 7 | 0 | 0 | 0

2 | RETURN ''; | 17000 | 8 | 0 | 0 | 0

2 | END IF; | 17000 | 9 | 0 | 0 | 0

2 | END; | 17000 | 10 | 0 | 0 | 0

2 | | 17000 | 11 | 0 | 0 | 0

2 | DECLARE | 17004 | 1 | 0 | 0 | 0

2 | sal\_diff NUMERIC(7,2); | 17004 | 2 | 0 | 0 | 0

2 | BEGIN | 17004 | 3 | 0 | 0 | 0

2 | IF TG\_OP = 'INSERT' THEN | 17004 | 4 | 0 | 0 | 0

2 | RAISE INFO 'Inserting employee %', NEW.empno; | 17004 | 5 | 1 | 0 | 8.4e-05

2 | RAISE INFO '..New salary: %', NEW.sal; | 17004 | 6 | 0 | 0 | 0

2 | RETURN NEW; | 17004 | 7 | 0 | 0 | 0

2 | END IF; | 17004 | 8 | 0 | 0 | 0

2 | IF TG\_OP = 'UPDATE' THEN | 17004 | 9 | 0 | 0 | 0

2 | sal\_diff := NEW.sal - OLD.sal; | 17004 | 10 | 1 | 0 | 0.000355

2 | RAISE INFO 'Updating employee %', OLD.empno; | 17004 | 11 | 1 | 0 | 0.000177

2 | RAISE INFO '..Old salary: %', OLD.sal; | 17004 | 12 | 1 | 0 | 5.5e-05

2 | RAISE INFO '..New salary: %', NEW.sal; | 17004 | 13 | 1 | 0 | 3.1e-05

2 | RAISE INFO '..Raise : %', sal\_diff; | 17004 | 14 | 1 | 0 | 2.8e-05

2 | RETURN NEW; | 17004 | 15 | 1 | 0 | 2.7e-05

2 | END IF; | 17004 | 16 | 1 | 0 | 1e-06

2 | IF TG\_OP = 'DELETE' THEN | 17004 | 17 | 0 | 0 | 0

2 | RAISE INFO 'Deleting employee %', OLD.empno; | 17004 | 18 | 0 | 0 | 0

2 | RAISE INFO '..Old salary: %', OLD.sal; | 17004 | 19 | 0 | 0 | 0

2 | RETURN OLD; | 17004 | 20 | 0 | 0 | 0

2 | END IF; | 17004 | 21 | 0 | 0 | 0

2 | END; | 17004 | 22 | 0 | 0 | 0

2 | | 17004 | 23 | 0 | 0 | 0

(68 rows)

1. Query the plsql\_profiler\_data view to review a subset of the information found in plsql\_profiler\_rawdata table:

acctg=# SELECT \* FROM plsql\_profiler\_data;

runid | unit\_number | line# | total\_occur | total\_time | min\_time | max\_time | spare1 | spare2 | spare3 | spare4

-------+-------------+-------+-------------+------------+----------+----------+--------+--------+--------+--------

1 | 16999 | 1 | 0 | 0 | 0 | 0 | | | |

1 | 16999 | 2 | 0 | 0 | 0 | 0 | | | |

1 | 16999 | 3 | 0 | 0 | 0 | 0 | | | |

1 | 16999 | 4 | 0 | 0 | 0 | 0 | | | |

1 | 16999 | 5 | 0 | 0 | 0 | 0 | | | |

1 | 16999 | 6 | 0 | 0 | 0 | 0 | | | |

1 | 16999 | 7 | 0 | 0 | 0 | 0 | | | |

1 | 16999 | 8 | 1 | 0.001621 | 0.001621 | 0.001621 | | | |

1 | 16999 | 9 | 1 | 0.000301 | 0.000301 | 0.000301 | | | |

1 | 16999 | 10 | 1 | 4.6e-05 | 4.6e-05 | 4.6e-05 | | | |

1 | 16999 | 11 | 1 | 0.001114 | 0.001114 | 0.001114 | | | |

1 | 16999 | 12 | 15 | 0.000206 | 5e-06 | 7.8e-05 | | | |

1 | 16999 | 13 | 15 | 8.3e-05 | 2e-06 | 4.7e-05 | | | |

1 | 16999 | 14 | 14 | 0.000773 | 4.7e-05 | 0.000116 | | | |

1 | 16999 | 15 | 0 | 0 | 0 | 0 | | | |

1 | 16999 | 16 | 1 | 1e-05 | 1e-05 | 1e-05 | | | |

1 | 16999 | 17 | 1 | 0 | 0 | 0 | | | |

1 | 16999 | 18 | 0 | 0 | 0 | 0 | | | |

2 | 17002 | 1 | 0 | 0 | 0 | 0 | | | |

2 | 17002 | 2 | 0 | 0 | 0 | 0 | | | |

2 | 17002 | 3 | 0 | 0 | 0 | 0 | | | |

2 | 17002 | 4 | 0 | 0 | 0 | 0 | | | |

2 | 17002 | 5 | 0 | 0 | 0 | 0 | | | |

2 | 17002 | 6 | 1 | 0.000143 | 0.000143 | 0.000143 | | | |

2 | 17002 | 7 | 0 | 0 | 0 | 0 | | | |

2 | 17002 | 8 | 0 | 0 | 0 | 0 | | | |

2 | 17002 | 9 | 1 | 3.2e-05 | 3.2e-05 | 3.2e-05 | | | |

2 | 17002 | 10 | 0 | 0 | 0 | 0 | | | |

2 | 17002 | 11 | 0 | 0 | 0 | 0 | | | |

2 | 17002 | 12 | 0 | 0 | 0 | 0 | | | |

2 | 17002 | 13 | 1 | 0.000383 | 0.000383 | 0.000383 | | | |

2 | 17002 | 14 | 1 | 6.3e-05 | 6.3e-05 | 6.3e-05 | | | |

2 | 17002 | 15 | 1 | 3.6e-05 | 3.6e-05 | 3.6e-05 | | | |

2 | 17002 | 16 | 0 | 0 | 0 | 0 | | | |

2 | 17000 | 1 | 0 | 0 | 0 | 0 | | | |

2 | 17000 | 2 | 0 | 0 | 0 | 0 | | | |

2 | 17000 | 3 | 0 | 0 | 0 | 0 | | | |

2 | 17000 | 4 | 0 | 0 | 0 | 0 | | | |

2 | 17000 | 5 | 1 | 0.000647 | 0.000647 | 0.000647 | | | |

2 | 17000 | 6 | 1 | 2.6e-05 | 2.6e-05 | 2.6e-05 | | | |

2 | 17000 | 7 | 0 | 0 | 0 | 0 | | | |

2 | 17000 | 8 | 0 | 0 | 0 | 0 | | | |

2 | 17000 | 9 | 0 | 0 | 0 | 0 | | | |

2 | 17000 | 10 | 0 | 0 | 0 | 0 | | | |

2 | 17000 | 11 | 0 | 0 | 0 | 0 | | | |

2 | 17004 | 1 | 0 | 0 | 0 | 0 | | | |

2 | 17004 | 2 | 0 | 0 | 0 | 0 | | | |

2 | 17004 | 3 | 0 | 0 | 0 | 0 | | | |

2 | 17004 | 4 | 0 | 0 | 0 | 0 | | | |

2 | 17004 | 5 | 1 | 8.4e-05 | 8.4e-05 | 8.4e-05 | | | |

2 | 17004 | 6 | 0 | 0 | 0 | 0 | | | |

2 | 17004 | 7 | 0 | 0 | 0 | 0 | | | |

2 | 17004 | 8 | 0 | 0 | 0 | 0 | | | |

2 | 17004 | 9 | 0 | 0 | 0 | 0 | | | |

2 | 17004 | 10 | 1 | 0.000355 | 0.000355 | 0.000355 | | | |

2 | 17004 | 11 | 1 | 0.000177 | 0.000177 | 0.000177 | | | |

2 | 17004 | 12 | 1 | 5.5e-05 | 5.5e-05 | 5.5e-05 | | | |

2 | 17004 | 13 | 1 | 3.1e-05 | 3.1e-05 | 3.1e-05 | | | |

2 | 17004 | 14 | 1 | 2.8e-05 | 2.8e-05 | 2.8e-05 | | | |

2 | 17004 | 15 | 1 | 2.7e-05 | 2.7e-05 | 2.7e-05 | | | |

2 | 17004 | 16 | 1 | 1e-06 | 1e-06 | 1e-06 | | | |

2 | 17004 | 17 | 0 | 0 | 0 | 0 | | | |

2 | 17004 | 18 | 0 | 0 | 0 | 0 | | | |

2 | 17004 | 19 | 0 | 0 | 0 | 0 | | | |

2 | 17004 | 20 | 0 | 0 | 0 | 0 | | | |

2 | 17004 | 21 | 0 | 0 | 0 | 0 | | | |

2 | 17004 | 22 | 0 | 0 | 0 | 0 | | | |

2 | 17004 | 23 | 0 | 0 | 0 | 0 | | | |

(68 rows)

### DBMS\_PROFILER Functions and Procedures

The DBMS\_PROFILER package collects and stores performance information about the PL/pgSQL and SPL statements that are executed during a profiling session; use the functions and procedures listed below to control the profiling tool.

Table ‑ DBMS\_PROFILER Functions/Procedures

| Function/Procedure | Function or Procedure | Return Type | Description |
| --- | --- | --- | --- |
| FLUSH\_DATA | Function and Procedure | Status Code or Exception | Flushes performance data collected in the current session without terminating the session (profiling continues). |
| GET\_VERSION (major OUT, minor OUT) | Procedure | n/a | Returns the version number of this package. |
| INTERNAL\_VERSION\_CHECK | Function | Status Code | Confirms that the current version of the profiler will work with the current database. |
| PAUSE\_PROFILER | Function and Procedure | Status Code or Exception | Pause data collection. |
| RESUME\_PROFILER | Function and Procedure | Status Code or Exception | Resume data collection. |
| START\_PROFILER[run\_comment, run\_comment1, run\_number OUT] | Functions and Procedures | Status Code or Exception | Start data collection. |
| STOP\_PROFILER | Function and Procedure | Status Code or Exception | Stop data collection and flush performance data to PLSQL\_PROFILER\_RAWDATA. |

Return Values

The functions within the DBMS\_PROFILER package return a status code to indicate success or failure; the DBMS\_PROFILER procedures raise an exception only if they encounter a failure. The status codes and messages returned by the functions, and the exceptions raised by the procedures are listed in the table below.

|  |  |  |  |
| --- | --- | --- | --- |
| Status Code | Message | Exception | Description |
| -1 | error version | version\_mismatch | The profiler version and the database are incompatible. |
| 0 | success | n/a | The operation completed successfully. |
| 1 | error\_param | profiler\_error | The operation received an incorrect parameter. |
| 2 | error\_io | profiler\_error | The data flush operation has failed. |

#### FLUSH\_DATA

The FLUSH\_DATA procedure or function flushes the data collected in the current session without terminating the profiler session. The data is flushed to the tables listed in Section 6.3 of the EDB Postgres Advanced Server Performance Features Guide. The signature of the FLUSH\_DATA function is:

DBMS\_PROFILER.FLUSH\_DATA  
 RETURN INTEGER;

The signature of the FLUSH\_DATA procedure is:

DBMS\_PROFILER.FLUSH\_DATA;

#### GET\_VERSION

The GET\_VERSION procedure returns the version of DBMS\_PROFILER. The procedure signature is:

DBMS\_PROFILER.GET\_VERSION( major OUT INTEGER

minor OUT INTEGER);

Parameters

major

The major version number of DBMS\_PROFILER.

minor

The minor version number of DBMS\_PROFILER.

#### INTERNAL\_VERSION\_CHECK

The INTERNAL\_VERSION\_CHECK function confirms that the current version of DBMS\_PROFILER will work with the current database. The function signature is:

DBMS\_PROFILER.INTERNAL\_VERSION\_CHECK  
 RETURN INTEGER;

#### PAUSE\_PROFILER

The PAUSE\_PROFILER function or procedure pauses a profiling session. The function signature is:

DBMS\_PROFILER.PAUSE\_PROFILER  
 RETURN INTEGER;

The signature of the PAUSE\_PROFILER procedure is:

DBMS\_PROFILER.PAUSE\_PROFILER;

#### RESUME\_PROFILER

The RESUME\_PROFILER function or procedure resumes a paused profiling session. The function signature is:

DBMS\_PROFILER.RESUME\_PROFILER  
 RETURN INTEGER;

The signature of the RESUME\_PROFILER procedure is:

DBMS\_PROFILER.RESUME\_PROFILER;

#### START\_PROFILER

The START\_PROFILER function or procedure starts a data collection session. The START\_PROFILER function has two forms:

DBMS\_PROFILER.START\_PROFILER(  
 run\_comment IN TEXT := sysdate,   
 run\_comment1 IN TEXT := '',   
 run\_number OUT INTEGER)   
 RETURN INTEGER;

DBMS\_PROFILER.START\_PROFILER(  
 run\_comment IN TEXT := sysdate,   
 run\_comment1 IN TEXT := '')   
 RETURN INTEGER;

The START\_PROFILER procedure has two forms:

DBMS\_PROFILER.START\_PROFILER (  
 run\_comment IN TEXT := sysdate,   
 run\_comment1 IN TEXT := '');

DBMS\_PROFILER.START\_PROFILER (  
 run\_comment IN TEXT := sysdate,   
 run\_comment1 IN TEXT := '',   
 run\_number OUT INTEGER);

Parameters

run\_comment

A user-defined comment for the profiler session; the default value is sysdate.

run\_comment1

An additional user-defined comment for the profiler session; the default value is ''.

run\_number

The session number of the profiler session.

#### STOP\_PROFILER

The STOP\_PROFILER function or procedure stops a profiling session and flushes the performance information to the DBMS\_PROFILER tables and view. The STOP\_PROFILER function signature is:

DBMS\_PROFILER.STOP\_PROFILER;

RETURN INTEGER;

The signature of the START\_PROFILER procedure is:

DBMS\_PROFILER.STOP\_PROFILER;

### DBMS\_PROFILER - Reference

The Advanced Server installer creates the following tables and views that you can query to review PL/SQL performance profile information:

| Table Name | Description |
| --- | --- |
| PLSQL\_PROFILER\_RUNS | Table containing information about all profiler runs, organized by runid. |
| PLSQL\_PROFILER\_UNITS | Table containing information about all profiler runs, organized by unit. |
| PLSQL\_PROFILER\_DATA | View containing performance statistics. |
| PLSQL\_PROFILER\_RAWDATA | Table containing the performance statistics and the extended performance statistics for DRITA counters and timers. |

#### PLSQL\_PROFILER\_RUNS

The PLSQL\_PROFILER\_RUNS table contains the following columns:

| Column | Data Type | Description |
| --- | --- | --- |
| runid | INTEGER (NOT NULL) | Unique identifier (plsql\_profiler\_runnumber) |
| related\_run | INTEGER | The runid of a related run. |
| run\_owner | TEXT | The role that recorded the profiling session. |
| run\_date | TIMESTAMP WITHOUT TIME ZONE | The profiling session start time. |
| run\_comment | TEXT | User comments relevant to this run |
| run\_total\_time | BIGINT | Run time (in nanoseconds) |
| run\_system\_info | TEXT | Currently Unused |
| run\_comment1 | TEXT | Additional user comments |
| spare1 | TEXT | Currently Unused |

#### PLSQL\_PROFILER\_UNITS

The PLSQL\_PROFILER\_UNITS table contains the following columns:

| Column | Data Type | Description |
| --- | --- | --- |
| runid | INTEGER | Unique identifier (plsql\_profiler\_runnumber) |
| unit\_number | OID | Corresponds to the OID of the row in the pg\_proc table that identifies the unit. |
| unit\_type | TEXT | PL/SQL function, procedure, trigger or anonymous block |
| unit\_owner | TEXT | The identity of the role that owns the unit. |
| unit\_name | TEXT | The complete signature of the unit. |
| unit\_timestamp | TIMESTAMP WITHOUT TIME ZONE | Creation date of the unit (currently NULL). |
| total\_time | BIGINT | Time spent within the unit (in nanoseconds) |
| spare1 | BIGINT | Currently Unused |
| spare2 | BIGINT | Currently Unused |

#### PLSQL\_PROFILER\_DATA

The PLSQL\_PROFILER\_DATA view contains the following columns:

| Column | Data Type | Description |
| --- | --- | --- |
| runid | INTEGER | Unique identifier (plsql\_profiler\_runnumber) |
| unit\_number | OID | Object ID of the unit that contains the current line. |
| line# | INTEGER | Current line number of the profiled workload. |
| total\_occur | BIGINT | The number of times that the line was executed. |
| total\_time | DOUBLE PRECISION | The amount of time spent executing the line. |
| min\_time | DOUBLE PRECISION | The minimum execution time for the line. |
| max\_time | DOUBLE PRECISION | The maximum execution time for the line. |
| spare1 | NUMBER | Currently Unused |
| spare2 | NUMBER | Currently Unused |
| spare3 | NUMBER | Currently Unused |
| spare4 | NUMBER | Currently Unused |

#### PLSQL\_PROFILER\_RAWDATA

The PLSQL\_PROFILER\_RAWDATA table contains the statistical information that is found in the PLSQL\_PROFILER\_DATA view, as well as the performance statistics returned by the DRITA counters and timers.

| Column | Data Type | Description |
| --- | --- | --- |
| runid | INTEGER | The run identifier (plsql\_profiler\_runnumber). |
| sourcecode | TEXT | The individual line of profiled code. |
| func\_oid | OID | Object ID of the unit that contains the current line. |
| line\_number | INTEGER | Current line number of the profiled workload. |
| exec\_count | BIGINT | The number of times that the line was executed. |
| time\_total | DOUBLE PRECISION | The amount of time spent executing the line. |
| time\_shortest | DOUBLE PRECISION | The minimum execution time for the line. |
| time\_longest | DOUBLE PRECISION | The maximum execution time for the line. |
| tuples\_returned | BIGINT | Currently Unused |
| num\_scans | BIGINT | Currently Unused |
| tuples\_fetched | BIGINT | Currently Unused |
| tuples\_inserted | BIGINT | Currently Unused |
| tuples\_updated | BIGINT | Currently Unused |
| tuples\_deleted | BIGINT | Currently Unused |
| blocks\_fetched | BIGINT | Currently Unused |
| blocks\_hit | BIGINT | Currently Unused |
| wal\_write | BIGINT | The server has waited for a write to the write-ahead log buffer (expect this value to be high). |
| wal\_flush | BIGINT | The server has waited for the write-ahead log to flush to disk. |
| wal\_file\_sync | BIGINT | The server has waited for the write-ahead log to sync to disk (related to the wal\_sync\_method parameter which, by default, is 'fsync' - better performance can be gained by changing this parameter to open\_sync). |
| buffer\_free\_list\_lock\_acquire | BIGINT | The server has waited for the short-term lock that synchronizes access to the list of free buffers (in shared memory). |
| shmem\_index\_lock\_acquire | BIGINT | The server has waited for the short-term lock that synchronizes access to the shared-memory map. |
| oid\_gen\_lock\_acquire | BIGINT | The server has waited for the short-term lock that synchronizes access to the next available OID (object ID). |
| xid\_gen\_lock\_acquire | BIGINT | The server has waited for the short-term lock that synchronizes access to the next available transaction ID. |
| proc\_array\_lock\_acquire | BIGINT | The server has waited for the short-term lock that synchronizes access to the process array |
| sinval\_lock\_acquire | BIGINT | The server has waited for the short-term lock that synchronizes access to the cache invalidation state. |
| freespace\_lock\_acquire | BIGINT | The server has waited for the short-term lock that synchronizes access to the freespace map. |
| wal\_insert\_lock\_acquire | BIGINT | The server has waited for the short-term lock that synchronizes write access to the write-ahead log. A high number may indicate that WAL buffers are sized too small. |
| wal\_write\_lock\_acquire | BIGINT | The server has waited for the short-term lock that synchronizes write-ahead log flushes. |
| control\_file\_lock\_acquire | BIGINT | The server has waited for the short-term lock that synchronizes write access to the control file (this should usually be a low number). |
| checkpoint\_lock\_acquire | BIGINT | A server process has waited for the short-term lock that prevents simultaneous checkpoints. |
| clog\_control\_lock\_acquire | BIGINT | The server has waited for the short-term lock that synchronizes access to the commit log. |
| subtrans\_control\_lock\_acquire | BIGINT | The server has waited for the short-term lock that synchronizes access to the subtransaction log. |
| multi\_xact\_gen\_lock\_acquire | BIGINT | The server has waited for the short-term lock that synchronizes access to the next available multi-transaction ID (when a SELECT...FOR SHARE statement executes). |
| multi\_xact\_offset\_lock\_acquire | BIGINT | The server has waited for the short-term lock that synchronizes access to the multi-transaction offset file (when a SELECT...FOR SHARE statement executes). |
| multi\_xact\_member\_lock\_acquire | BIGINT | The server has waited for the short-term lock that synchronizes access to the multi-transaction member file (when a SELECT...FOR SHARE statement executes). |
| rel\_cache\_init\_lock\_acquire | BIGINT | The server has waited for the short-term lock that prevents simultaneous relation-cache loads/unloads. |
| bgwriter\_communication\_lock\_acquire | BIGINT | The bgwriter (background writer) process has waited for the short-term lock that synchronizes messages between the bgwriter and a backend process. |
| two\_phase\_state\_lock\_acquire | BIGINT | The server has waited for the short-term lock that synchronizes access to the list of prepared transactions. |
| tablespace\_create\_lock\_acquire | BIGINT | The server has waited for the short-term lock that prevents simultaneous CREATE TABLESPACE or DROP TABLESPACE commands. |
| btree\_vacuum\_lock\_acquire | BIGINT | The server has waited for the short-term lock that synchronizes access to the next available vacuum cycle ID. |
| add\_in\_shmem\_lock\_acquire | BIGINT | Currently Unused |
| autovacuum\_lock\_acquire | BIGINT | The server has waited for the short-term lock that synchronizes access to the shared autovacuum state. |
| autovacuum\_schedule\_lock\_acquire | BIGINT | The server has waited for the short-term lock that synchronizes access to the autovacuum schedule. |
| syncscan\_lock\_acquire | BIGINT | The server has waited for the short-term lock that coordinates synchronous scans. |
| icache\_lock\_acquire | BIGINT | The server has waited for the short-term lock that synchronizes access to InfiniteCache state |
| breakpoint\_lock\_acquire | BIGINT | The server has waited for the short-term lock that synchronizes access to the debugger breakpoint list. |
| lwlock\_acquire | BIGINT | The server has waited for a short-term lock that has not been described elsewhere in this section. |
| db\_file\_read | BIGINT | A server process has waited for the completion of a read (from disk). |
| db\_file\_write | BIGINT | A server process has waited for the completion of a write (to disk). |
| db\_file\_sync | BIGINT | A server process has waited for the operating system to flush all changes to disk. |
| db\_file\_extend | BIGINT | A server process has waited for the operating system while adding a new page to the end of a file. |
| sql\_parse | BIGINT | Currently Unused |
| query\_plan | BIGINT | The server has generated a query plan. |
| infinitecache\_read | BIGINT | The server has waited for an Infinite Cache read request. |
| infinitecache\_write | BIGINT | The server has waited for an Infinite Cache write request. |
| wal\_write\_time | BIGINT | The amount of time that the server has waited for a write to the write-ahead log buffer (expect this value to be high). |
| wal\_flush\_time | BIGINT | The amount of time that the server has waited for the write-ahead log to flush to disk. |
| wal\_file\_sync\_time | BIGINT | The amount of time that the server has waited for the write-ahead log to sync to disk (related to the wal\_sync\_method parameter which, by default, is 'fsync' - better performance can be gained by changing this parameter to open\_sync). |
| buffer\_free\_list\_lock\_acquire\_time | BIGINT | The amount of time that the server has waited for the short-term lock that synchronizes access to the list of free buffers (in shared memory). |
| shmem\_index\_lock\_acquire\_time | BIGINT | The amount of time that the server has waited for the short-term lock that synchronizes access to the shared-memory map. |
| oid\_gen\_lock\_acquire\_time | BIGINT | The amount of time that the server has waited for the short-term lock that synchronizes access to the next available OID (object ID). |
| xid\_gen\_lock\_acquire\_time | BIGINT | The amount of time that the server has waited for the short-term lock that synchronizes access to the next available transaction ID. |
| proc\_array\_lock\_acquire\_time | BIGINT | The amount of time that the server has waited for the short-term lock that synchronizes access to the process array. |
| sinval\_lock\_acquire\_time | BIGINT | The amount of time that the server has waited for the short-term lock that synchronizes access to the cache invalidation state. |
| freespace\_lock\_acquire\_time | BIGINT | The amount of time that the server has waited for the short-term lock that synchronizes access to the freespace map. |
| wal\_insert\_lock\_acquire\_time | BIGINT | The amount of time that the server has waited for the short-term lock that synchronizes write access to the write-ahead log. A high number may indicate that WAL buffers are sized too small. |
| wal\_write\_lock\_acquire\_time | BIGINT | The amount of time that the server has waited for the short-term lock that synchronizes write-ahead log flushes. |
| control\_file\_lock\_acquire\_time | BIGINT | The amount of time that the server has waited for the short-term lock that synchronizes write access to the control file (this should usually be a low number). |
| checkpoint\_lock\_acquire\_time | BIGINT | The amount of time that the server process has waited for the short-term lock that prevents simultaneous checkpoints. |
| clog\_control\_lock\_acquire\_time | BIGINT | The amount of time that the server has waited for the short-term lock that synchronizes access to the commit log. |
| subtrans\_control\_lock\_acquire\_time | BIGINT | The amount of time that the server has waited for the short-term lock that synchronizes access to the subtransaction log. |
| multi\_xact\_gen\_lock\_acquire\_time | BIGINT | The amount of time that the server has waited for the short-term lock that synchronizes access to the next available multi-transaction ID (when a SELECT...FOR SHARE statement executes). |
| multi\_xact\_offset\_lock\_acquire\_time | BIGINT | The amount of time that the server has waited for the short-term lock that synchronizes access to the multi-transaction offset file (when a SELECT...FOR SHARE statement executes). |
| multi\_xact\_member\_lock\_acquire\_time | BIGINT | The amount of time that the server has waited for the short-term lock that synchronizes access to the multi-transaction member file (when a SELECT...FOR SHARE statement executes). |
| rel\_cache\_init\_lock\_acquire\_time | BIGINT | The amount of time that the server has waited for the short-term lock that prevents simultaneous relation-cache loads/unloads. |
| bgwriter\_communication\_lock\_acquire\_time | BIGINT | The amount of time that the bgwriter (background writer) process has waited for the short-term lock that synchronizes messages between the bgwriter and a backend process. |
| two\_phase\_state\_lock\_acquire\_time | BIGINT | The amount of time that the server has waited for the short-term lock that synchronizes access to the list of prepared transactions. |
| tablespace\_create\_lock\_acquire\_time | BIGINT | The amount of time that the server has waited for the short-term lock that prevents simultaneous CREATE TABLESPACE or DROP TABLESPACE commands. |
| btree\_vacuum\_lock\_acquire\_time | BIGINT | The amount of time that the server has waited for the short-term lock that synchronizes access to the next available vacuum cycle ID. |
| add\_in\_shmem\_lock\_acquire\_time | BIGINT | Obsolete/unused |
| autovacuum\_lock\_acquire\_time | BIGINT | The amount of time that the server has waited for the short-term lock that synchronizes access to the shared autovacuum state. |
| autovacuum\_schedule\_lock\_acquire\_time | BIGINT | The amount of time that the server has waited for the short-term lock that synchronizes access to the autovacuum schedule. |
| syncscan\_lock\_acquire\_time | BIGINT | The amount of time that the server has waited for the short-term lock that coordinates synchronous scans. |
| icache\_lock\_acquire\_time | BIGINT | The amount of time that the server has waited for the short-term lock that synchronizes access to InfiniteCache state |
| breakpoint\_lock\_acquire\_time | BIGINT | The amount of time that the server has waited for the short-term lock that synchronizes access to the debugger breakpoint list. |
| lwlock\_acquire\_time | BIGINT | The amount of time that the server has waited for a short-term lock that has not been described elsewhere in this section. |
| db\_file\_read\_time | BIGINT | The amount of time that the server process has waited for the completion of a read (from disk). |
| db\_file\_write\_time | BIGINT | The amount of time that the server process has waited for the completion of a write (to disk). |
| db\_file\_sync\_time | BIGINT | The amount of time that the server process has waited for the operating system to flush all changes to disk. |
| db\_file\_extend\_time | BIGINT | The amount of time that the server process has waited for the operating system while adding a new page to the end of a file. |
| sql\_parse\_time | BIGINT | The amount of time that the server has parsed a SQL statement. |
| query\_plan\_time | BIGINT | The amount of time that the server has computed the execution plan for a SQL statement. |
| infinitecache\_read\_time | BIGINT | The amount of time that the server has waited for an Infinite Cache read request. |
| infinitecache\_write\_time | BIGINT | The amount of time that the server has waited for an Infinite Cache write request. |
| totalwaits | BIGINT | The total number of event waits. |
| totalwaittime | BIGINT | The total time spent waiting for an event. |

## Dynamic Runtime Instrumentation Tools Architecture (DRITA)

The Dynamic Runtime Instrumentation Tools Architecture (DRITA) allows a DBA to query catalog views to determine the *wait events* that affect the performance of individual sessions or the system as a whole. DRITA records the number of times each event occurs as well as the time spent waiting; you can use this information to diagnose performance problems. DRITA offers this functionality, while consuming minimal system resources.

DRITA compares *snapshots* to evaluate the performance of a system. A snapshot is a saved set of system performance data at a given point in time. Each snapshot is identified by a unique ID number; you can use snapshot ID numbers with DRITA reporting functions to return system performance statistics.

### Configuring and Using DRITA

Advanced Server's postgresql.conf file includes a configuration parameter named timed\_statistics that controls the collection of timing data. The valid parameter values are TRUE or FALSE; the default value is FALSE.

This is a dynamic parameter which can be modified in the postgresql.conf file, or while a session is in progress. To enable DRITA, you must either:

Modify the postgresql.conf file, setting the timed\_statistics parameter to TRUE.

or

Connect to the server with the EDB-PSQL client, and invoke the command:

SET timed\_statistics = TRUE

After modifying the timed\_statistics parameter, take a starting snapshot. A snapshot captures the current state of each timer and event counter. The server will compare the starting snapshot to a later snapshot to gauge system performance.

Use the edbsnap() function to take the beginning snapshot:

edb=# SELECT \* FROM edbsnap();

edbsnap

----------------------

Statement processed.

(1 row)

Then, run the workload that you would like to evaluate; when the workload has completed (or at a strategic point during the workload), take another snapshot:

edb=# SELECT \* FROM edbsnap();

edbsnap

----------------------

Statement processed.

(1 row)

You can capture multiple snapshots during a session. Then, use the DRITA functions and reports to manage and compare the snapshots to evaluate performance information.

## DRITA Functions

You can use DRITA functions to gather wait information and manage snapshots. DRITA functions are fully supported by Advanced Server 9.5 whether your installation is made compatible with Oracle databases or is made in PostgreSQL-compatible mode.

### get\_snaps()

The get\_snaps() function returns a list of the current snapshots. The signature is:

get\_snaps()

The following example demonstrates using the get\_snaps() function to display a list of snapshots:

edb=# SELECT \* FROM get\_snaps();

get\_snaps

------------------------------

1 11-FEB-10 10:41:05.668852

2 11-FEB-10 10:42:27.26154

3 11-FEB-10 10:45:48.999992

4 11-FEB-10 11:01:58.345163

5 11-FEB-10 11:05:14.092683

6 11-FEB-10 11:06:33.151002

7 11-FEB-10 11:11:16.405664

8 11-FEB-10 11:13:29.458405

9 11-FEB-10 11:23:57.595916

10 11-FEB-10 11:29:02.214014

11 11-FEB-10 11:31:44.244038

(11 rows)

The first column in the result list displays the snapshot identifier; the second column displays the date and time that the snapshot was captured.

### sys\_rpt()

The sys\_rpt() function returns system wait information. The signature is:

sys\_rpt(*beginning\_id*, *ending\_id*, *top\_n*)

Parameters

*beginning*\_*id*

*beginning*\_*id* is an integer value that represents the beginning session identifier.

*ending*\_*id*

*ending*\_*id* is an integer value that represents the ending session identifier.

*top\_n*

*top\_n* represents the number of rows to return

This example demonstrates a call to the sys\_rpt()function:

edb=# SELECT \* FROM sys\_rpt(9, 10, 10);

sys\_rpt

-----------------------------------------------------------------------------

WAIT NAME COUNT WAIT TIME % WAIT

---------------------------------------------------------------------------

wal write 21250 104.723772 36.31

db file read 121407 72.143274 25.01

wal flush 84185 51.652495 17.91

wal file sync 712 29.482206 10.22

infinitecache write 84178 15.814444 5.48

db file write 84177 14.447718 5.01

infinitecache read 672 0.098691 0.03

db file extend 190 0.040386 0.01

query plan 52 0.024400 0.01

wal insert lock acquire 4 0.000837 0.00

(12 rows)

The information displayed in the result set includes:

| Column Name | Description |
| --- | --- |
| WAIT NAME | The name of the wait. |
| COUNT | The number of times that the wait event occurred. |
| WAIT TIME | The time of the wait event in milliseconds. |
| % WAIT | The percentage of the total wait time used by this wait for this session. |

### sess\_rpt()

The sess\_rpt() function returns session wait information. The signature is:

sess\_rpt(*beginning\_id*, *ending\_id, top\_n*)

Parameters

*beginning*\_*id*

*beginning\_id* is an integer value that represents the beginning session identifier.

*ending\_id*

*ending\_id* is an integer value that represents the ending session identifier.

*top\_n*

*top\_n* represents the number of rows to return

The following example demonstrates a call to the sess\_rpt()function:

SELECT \* FROM sess\_rpt(18, 19, 10);

sess\_rpt

-----------------------------------------------------------------------------

ID USER WAIT NAME COUNT TIME(ms) %WAIT SES %WAIT ALL

----------------------------------------------------------------------------

17373 enterprise db file read 30 0.175713 85.24 85.24

17373 enterprise query plan 18 0.014930 7.24 7.24

17373 enterprise wal flush 6 0.004067 1.97 1.97

17373 enterprise wal write 1 0.004063 1.97 1.97

17373 enterprise wal file sync 1 0.003664 1.78 1.78

17373 enterprise infinitecache read 38 0.003076 1.49 1.49

17373 enterprise infinitecache write 5 0.000548 0.27 0.27

17373 enterprise db file extend 190 0.04.386 0.03 0.03

17373 enterprise db file write 5 0.000082 0.04 0.04

(11 rows)

The information displayed in the result set includes:

| Column Name | Description |
| --- | --- |
| ID | The processID of the session. |
| USER | The name of the user incurring the wait. |
| WAIT NAME | The name of the wait event. |
| COUNT | The number of times that the wait event occurred. |
| TIME (ms) | The length of the wait event in milliseconds. |
| % WAIT SES | The percentage of the total wait time used by this wait for this session. |
| % WAIT ALL | The percentage of the total wait time used by this wait (for all sessions). |

### sessid\_rpt()

The sessid\_rpt() function returns session ID information for a specified backend. The signature is:

sessid\_rpt(*beginning\_id*, *ending\_id, backend\_id*)

Parameters

*beginning\_id*

*beginning\_id* is an integer value that represents the beginning session identifier.

*ending\_id*

*ending\_id* is an integer value that represents the ending session identifier.

*backend\_id*

*backend\_id* is an integer value that represents the backend identifier.

The following code sample demonstrates a call to sessid\_rpt():

SELECT \* FROM sessid\_rpt(18, 19, 17373);

sessid\_rpt

-----------------------------------------------------------------------------

ID USER WAIT NAME COUNT TIME(ms) %WAIT SES %WAIT ALL

----------------------------------------------------------------------------

17373 enterprise db file read 30 0.175713 85.24 85.24

17373 enterprise query plan 18 0.014930 7.24 7.24

17373 enterprise wal flush 6 0.004067 1.97 1.97

17373 enterprise wal write 1 0.004063 1.97 1.97

17373 enterprise wal file sync 1 0.003664 1.78 1.78

17373 enterprise infinitecache read 38 0.003076 1.49 1.49

17373 enterprise infinitecache write 5 0.000548 0.27 0.27

17373 enterprise db file extend 190 0.040386 0.03 0.03

17373 enterprise db file write 5 0.000082 0.04 0.04

(11 rows)

The information displayed in the result set includes:

| Column Name | Description |
| --- | --- |
| ID | The process ID of the wait. |
| USER | The name of the user that owns the session. |
| WAIT NAME | The name of the wait event. |
| COUNT | The number of times that the wait event occurred. |
| TIME (ms) | The length of the wait in milliseconds. |
| % WAIT SES | The percentage of the total wait time used by this wait for this session. |
| % WAIT ALL | The percentage of the total wait time used by this wait (for all sessions). |

### sesshist\_rpt()

The sesshist\_rpt() function returns session wait information for a specified backend. The signature is:

sesshist\_rpt(*snapshot\_id*, *session\_id*)

Parameters

*snapshot\_id*

*snapshot\_id* is an integer value that identifies the snapshot.

*session\_id*

*session\_id* is an integer value that represents the session.

The following example demonstrates a call to the sesshist\_rpt()function:

edb=# SELECT \* FROM sesshist\_rpt (9, 5531);

sesshist\_rpt

----------------------------------------------------------------------------

ID USER SEQ WAIT NAME

ELAPSED(ms) File Name # of Blk Sum of Blks

----------------------------------------------------------------------------

5531 enterprise 1 db file read

18546 14309 session\_waits\_pk 1 1

5531 enterprise 2 infinitecache read

125 14309 session\_waits\_pk 1 1

5531 enterprise 3 db file read

376 14304 edb$session\_waits 0 1

5531 enterprise 4 infinitecache read

166 14304 edb$session\_waits 0 1

5531 enterprise 5 db file read

7978 1260 pg\_authid 0 1

5531 enterprise 6 infinitecache read

154 1260 pg\_authid 0 1

5531 enterprise 7 db file read

628 14302 system\_waits\_pk 1 1

5531 enterprise 8 infinitecache read

463 14302 system\_waits\_pk 1 1

5531 enterprise 9 db file read

3446 14297 edb$system\_waits 0 1

5531 enterprise 10 infinitecache read

187 14297 edb$system\_waits 0 1

5531 enterprise 11 db file read

14750 14295 snap\_pk 1 1

5531 enterprise 12 infinitecache read

416 14295 snap\_pk 1 1

5531 enterprise 13 db file read

7139 14290 edb$snap 0 1

5531 enterprise 14 infinitecache read

158 14290 edb$snap 0 1

5531 enterprise 15 db file read

27287 14288 snapshot\_num\_seq 0 1

5531 enterprise 16 infinitecache read

(17 rows)

The information displayed in the result set includes:

| Column Name | Description |
| --- | --- |
| ID | The system-assigned identifier of the wait. |
| USER | The name of the user that incurred the wait. |
| SEQ | The sequence number of the wait event. |
| WAIT NAME | The name of the wait event. |
| ELAPSED (ms) | The length of the wait event in milliseconds. |
| File | The relfilenode number of the file. |
| Name | If available, the name of the file name related to the wait event. |
| # of Blk | The block number read or written for a specific instance of the event . |
| Sum of Blks | The number of blocks read. |

### purgesnap()

The purgesnap() function purges a range of snapshots from the snapshot tables. The signature is:

purgesnap(*beginning\_id*, *ending\_id*)

Parameters

*beginning*\_*id*

*beginning*\_*id* is an integer value that represents the beginning session identifier.

*ending*\_*id*

*ending*\_*id* is an integer value that represents the ending session identifier.

purgesnap()removes all snapshots between *beginning*\_*id* and *ending\_id* (inclusive):

SELECT \* FROM purgesnap(6, 9);

purgesnap

------------------------------------

Snapshots in range 6 to 9 deleted.

(1 row)

A call to the get\_snaps() function after executing the example shows that snapshots 6 through 9 have been purged from the snapshot tables:

edb=# SELECT \* FROM get\_snaps();

get\_snaps

------------------------------

1 11-FEB-10 10:41:05.668852

2 11-FEB-10 10:42:27.26154

3 11-FEB-10 10:45:48.999992

4 11-FEB-10 11:01:58.345163

5 11-FEB-10 11:05:14.092683

10 11-FEB-10 11:29:02.214014

11 11-FEB-10 11:31:44.244038

(7 rows)

### truncsnap()

Use the truncsnap() function to delete all records from the snapshot table. The signature is:

truncsnap()

For example:

SELECT \* FROM truncsnap();

truncsnap

----------------------

Snapshots truncated.

(1 row)

A call to the get\_snaps() function after calling the truncsnap() function shows that all records have been removed from the snapshot tables:

SELECT \* FROM get\_snaps();

get\_snaps

-----------

(0 rows)

## Simulating Statspack AWR Reports

The functions described in this section return information comparable to the information contained in an Oracle Statspack/AWR (Automatic Workload Repository) report. When taking a snapshot, performance data from system catalog tables is saved into history tables. The reporting functions listed below report on the differences between two given snapshots.

* stat\_db\_rpt()
* stat\_tables\_rpt()
* statio\_tables\_rpt()
* stat\_indexes\_rpt()
* statio\_indexes\_rpt()

The reporting functions can be executed individually or you can execute all five functions by calling the edbreport() function.

### edbreport()

The edbreport() function includes data from the other reporting functions, plus additional system information. The signature is:

edbreport(beginning\_id, ending\_id)

Parameters

beginning\_id

beginning\_id is an integer value that represents the beginning session identifier.

ending\_id

ending\_id is an integer value that represents the ending session identifier.

The call to the edbreport() function returns a composite report that contains system information and the reports returned by the other statspack functions. :

edb=# SELECT \* FROM edbreport(9, 10);

edbreport

-----------------------------------------------------------------------------

EnterpriseDB Report for database edb 23-AUG-15

Version: EnterpriseDB 9.5.0.0 on i686-pc-linux-gnu

Begin snapshot: 9 at 23-AUG-15 13:45:07.165123

End snapshot: 10 at 23-AUG-15 13:45:35.653036

Size of database edb is 155 MB

Tablespace: pg\_default Size: 179 MB Owner: enterprisedb

Tablespace: pg\_global Size: 435 kB Owner: enterprisedb

Schema: pg\_toast\_temp\_1 Size: 0 bytes Owner: enterprisedb

Schema: public Size: 0 bytes Owner: enterprisedb

Schema: enterprisedb Size: 143 MB Owner: enterprisedb

Schema: pgagent Size: 192 kB Owner: enterprisedb

Schema: dbms\_job\_procedure Size: 0 bytes Owner: enterprisedb

The information displayed in the report introduction includes the database name and version, the current date, the beginning and ending snapshot date and times, database and tablespace details and schema information.

Top 10 Relations by pages

TABLE RELPAGES

----------------------------------------------------------------------------

pgbench\_accounts 15874

pg\_proc 102

edb$statio\_all\_indexes 73

edb$stat\_all\_indexes 73

pg\_attribute 67

pg\_depend 58

edb$statio\_all\_tables 49

edb$stat\_all\_tables 47

pgbench\_tellers 37

pg\_description 32

The information displayed in the Top 10 Relations by pages section includes:

| Column Name | Description |
| --- | --- |
| TABLE | The name of the table. |
| RELPAGES | The number of pages in the table. |

Top 10 Indexes by pages

INDEX RELPAGES

----------------------------------------------------------------------------

pgbench\_accounts\_pkey 2198

pg\_depend\_depender\_index 32

pg\_depend\_reference\_index 31

pg\_proc\_proname\_args\_nsp\_index 30

pg\_attribute\_relid\_attnam\_index 23

pg\_attribute\_relid\_attnum\_index 17

pg\_description\_o\_c\_o\_index 15

edb$statio\_idx\_pk 11

edb$stat\_idx\_pk 11

pg\_proc\_oid\_index 9

The information displayed in the Top 10 Indexes by pages section includes:

| Column Name | Description |
| --- | --- |
| INDEX | The name of the index. |
| RELPAGES | The number of pages in the index. |

Top 10 Relations by DML

SCHEMA RELATION UPDATES DELETES INSERTS

---------------------------------------------------------------------------

enterprisedb pgbench\_accounts 10400 0 1000000

enterprisedb pgbench\_tellers 10400 0 100

enterprisedb pgbench\_branches 10400 0 10

enterprisedb pgbench\_history 0 0 10400

pgagent pga\_jobclass 0 0 6

pgagent pga\_exception 0 0 0

pgagent pga\_job 0 0 0

pgagent pga\_jobagent 0 0 0

pgagent pga\_joblog 0 0 0

pgagent pga\_jobstep 0 0 0

The information displayed in the Top 10 Relations by DML section includes:

| Column Name | Description |
| --- | --- |
| SCHEMA | The name of the schema in which the table resides. |
| RELATION | The name of the table. |
| UPDATES | The number of UPDATES performed on the table. |
| DELETES | The number of DELETES performed on the table. |
| INSERTS | The number of INSERTS performed on the table. |

DATA from pg\_stat\_database

DATABASE NUMBACKENDS XACT COMMIT XACT ROLLBACK BLKS READ BLKS HIT BLKS ICACHE HIT HIT RATIO ICACHE HIT RATIO

----------------------------------------------------------------------------

edb 0 142 0 78 10446   
 0 99.26 0.00

DATA from pg\_buffercache not included because pg\_buffercache is not installed

The information displayed in the DATA from pg\_stat\_database section of the report includes:

| Column Name | Description |
| --- | --- |
| DATABASE | The name of the database. |
| NUMBACKENDS | Number of backends currently connected to this database. This is the only column in this view that returns a value reflecting current state; all other columns return the accumulated values since the last reset. |
| XACT COMMIT | Number of transactions in this database that have been committed. |
| XACT ROLLBACK | Number of transactions in this database that have been rolled back. |
| BLKS READ | Number of disk blocks read in this database. |
| BLKS HIT | Number of times disk blocks were found already in the buffer cache (when a read was not necessary). |
| BLKS ICACHE HIT | The number of blocks found in Infinite Cache. |
| HIT RATIO | The percentage of times that a block was found in the shared buffer cache. |
| ICACHE HIT RATIO | The percentage of times that a block was found in Infinite Cache. |

DATA from pg\_stat\_all\_tables ordered by seq scan

SCHEMA RELATION SEQ SCAN REL TUP READ IDX SCAN IDX TUP READ INS UPD DEL

----------------------------------------------------------------------------

pg\_catalog pg\_class 16 7162 546 319 0 1 0

pg\_catalog pg\_am 13 13 0 0 0 0 0

pg\_catalog pg\_database 4 16 42 42 0 0 0

pg\_catalog pg\_index 4 660 145 149 0 0 0

pg\_catalog pg\_namespace 4 100 49 49 0 0 0

sys edb$snap 1 9 0 0 1 0 0

pg\_catalog pg\_authid 1 1 25 25 0 0 0

sys edb$session\_wait\_history 0 0 0 0 50 0 0

sys edb$session\_waits 0 0 0 0 2 0 0

sys edb$stat\_all\_indexes 0 0 0 0 165 0 0

The information displayed in the DATA from pg\_stat\_all\_tables ordered by seq scan section includes:

| Column Name | Description |
| --- | --- |
| SCHEMA | The name of the schema in which the table resides. |
| RELATION | The name of the table. |
| SEQ SCAN | The number of sequential scans initiated on this table.. |
| REL TUP READ | The number of tuples read in the table. |
| IDX SCAN | The number of index scans initiated on the table. |
| IDX TUP READ | The number of index tuples read. |
| INS | The number of rows inserted. |
| UPD | The number of rows updated. |
| DEL | The number of rows deleted. |

DATA from pg\_stat\_all\_tables ordered by rel tup read

SCHEMA RELATION SEQ SCAN REL TUP READ IDX SCAN IDX TUP READ INS UPD DEL

----------------------------------------------------------------------------

pg\_catalog pg\_class 16 7162 546 319 0 1 0

pg\_catalog pg\_index 4 660 145 149 0 0 0

pg\_catalog pg\_namespace 4 100 49 49 0 0 0

pg\_catalog pg\_database 4 16 42 42 0 0 0

pg\_catalog pg\_am 13 13 0 0 0 0 0

sys edb$snap 1 9 0 0 1 0 0

pg\_catalog pg\_authid 1 1 25 25 0 0 0

sys edb$session\_wait\_history 0 0 0 0 50 0 0

sys edb$session\_waits 0 0 0 0 2 0 0

sys edb$stat\_all\_indexes 0 0 0 0 165 0 0

The information displayed in the DATA from pg\_stat\_all\_tables ordered by rel tup read section includes:

| Column Name | Description |
| --- | --- |
| SCHEMA | The name of the schema in which the table resides. |
| RELATION | The name of the table. |
| SEQ SCAN | The number of sequential scans performed on the table. |
| REL TUP READ | The number of tuples read from the table. |
| IDX SCAN | The number of index scans performed on the table. |
| IDX TUP READ | The number of index tuples read. |
| INS | The number of rows inserted. |
| UPD | The number of rows updated. |
| DEL | The number of rows deleted. |

DATA from pg\_statio\_all\_tables

SCHEMA RELATION HEAP HEAP HEAP IDX IDX

READ HIT ICACHE READ HIT

HIT

IDX TOAST TOAST TOAST TIDX TIDX TIDX

ICACHE READ HIT ICACHE READ HIT ICACHE

HIT HIT HIT

-----------------------------------------------------------------------------

public pgbench\_accounts 92766 67215 288 59 32126

9 0 0 0 0 0 0

pg\_catalog pg\_class 0 296 0 3 16

0 0 0 0 0 0 0

sys edb$stat\_all\_indexes 8 125 0 4 233

0 0 0 0 0 0 0

sys edb$statio\_all\_index 8 125 0 4 233

0 0 0 0 0 0 0

sys edb$stat\_all\_tables 6 91 0 2 174

0 0 0 0 0 0 0

sys edb$statio\_all\_table 6 91 0 2 174

0 0 0 0 0 0 0

pg\_catalog pg\_namespace 3 72 0 0 0

0 0 0 0 0 0 0

sys edb$session\_wait\_his 1 24 0 4 47

0 0 0 0 0 0 0

pg\_catalog pg\_opclass 3 13 0 2 0

0 0 0 0 0 0 0

pg\_catalog pg\_trigger 0 12 0 1 15

0 0 0 0 0 0 0

The information displayed in the Data from pg\_statio\_all\_tables section includes:

| Column Name | Description |
| --- | --- |
| SCHEMA | The name of the schema in which the table resides. |
| RELATION | The name of the table. |
| HEAP READ | The number of heap blocks read. |
| HEAP HIT | The number of heap blocks hit. |
| HEAP ICACHE HIT | The number of heap blocks in Infinite Cache. |
| IDX READ | The number of index blocks read. |
| IDX HIT | The number of index blocks hit. |
| IDX ICACHE HIT | The number of index blocks in Infinite Cache. |
| TOAST READ | The number of toast blocks read. |
| TOAST HIT | The number of toast blocks hit. |
| TOAST ICACHE HIT | The number of toast blocks in Infinite Cache. |
| TIDX READ | The number of toast index blocks read. |
| TIDX HIT | The number of toast index blocks hit. |
| TIDX ICACHE HIT | The number of toast index blocks in Infinite Cache. |

DATA from pg\_stat\_all\_indexes

SCHEMA RELATION INDEX IDX SCAN IDX TUP READ IDX TUP FETCH

----------------------------------------------------------------------------

pg\_catalog pg\_attribute pg\_attribute\_relid\_attnum\_index 427 907 907

pg\_catalog pg\_class pg\_class\_relname\_nsp\_index 289 62 62

pg\_catalog pg\_class pg\_class\_oid\_index 257 257 257

pg\_catalog pg\_statistic pg\_statistic\_relid\_att\_inh\_index 207 196 196

enterprisedb pgbench\_accounts pgbench\_accounts\_pkey 200 255 200

pg\_catalog pg\_cast pg\_cast\_source\_target\_index 199 50 50

pg\_catalog pg\_proc pg\_proc\_oid\_index 116 116 116

pg\_catalog edb\_partition edb\_partition\_partrelid\_index 112 0 0

pg\_catalog edb\_policy edb\_policy\_object\_name\_index 112 0 0

enterprisedb pgbench\_branches pgbench\_branches\_pkey 101 110 0

The information displayed in the DATA from pg\_stat\_all\_indexes section includes:

| Column Name | Description |
| --- | --- |
| SCHEMA | The name of the schema in which the index resides. |
| RELATION | The name of the table on which the index is defined. |
| INDEX | The name of the index. |
| IDX SCAN | The number of indexes scans initiated on this index. |
| IDX TUP READ | Number of index entries returned by scans on this index |
| IDX TUP FETCH | Number of live table rows fetched by simple index scans using this index. |

DATA from pg\_statio\_all\_indexes

SCHEMA RELATION INDEX IDX BLKS READ IDX BLKS HIT IDX BLKS ICACHE HIT

----------------------------------------------------------------------------

pg\_catalog pg\_attribute pg\_attribute\_relid\_attnum\_index 0 867 0

enterprisedb pgbench\_accounts pgbench\_accounts\_pkey 1 778 0

pg\_catalog pg\_class pg\_class\_relname\_nsp\_index 0 590 0

pg\_catalog pg\_class pg\_class\_oid\_index 0 527 0

pg\_catalog pg\_statistic pg\_statistic\_relid\_att\_inh\_index 0 441 0

sys edb$stat\_all\_indexes edb$stat\_idx\_pk 1 332 0

sys edb$statio\_all\_indexes edb$statio\_idx\_pk 1 332 0

pg\_catalog pg\_proc pg\_proc\_oid\_index 0 244 0

sys edb$stat\_all\_tables edb$stat\_tab\_pk 0 241 0

sys edb$statio\_all\_tables edb$statio\_tab\_pk 0 241 0

The information displayed in the DATA from pg\_statio\_all\_indexes section includes:

| Column Name | Description |
| --- | --- |
| SCHEMA | The name of the schema in which the index resides. |
| RELATION | The name of the table on which the index is defined. |
| INDEX | The name of the index. |
| IDX BLKS READ | The number of index blocks read. |
| IDX BLKS HIT | The number of index blocks hit. |
| IDX BLKS ICACHE HIT | The number of index blocks in Infinite Cache that were hit. |

System Wait Information

WAIT NAME COUNT WAIT TIME % WAIT

---------------------------------------------------------------------------

query plan 0 0.000407 100.00

db file read 0 0.000000 0.00

The information displayed in the System Wait Information section includes:

| Column Name | Description |
| --- | --- |
| WAIT NAME | The name of the wait. |
| COUNT | The number of times that the wait event occurred. |
| WAIT TIME | The length of the wait time in milliseconds. |
| % WAIT | The percentage of the total wait time used by this wait for this session. |

Database Parameters from postgresql.conf

PARAMETER SETTING CONTEXT MINVAL MAXVAL

----------------------------------------------------------------------------

allow\_system\_table\_mods off postmaster

application\_name psql user

archive\_command (disabled) sighup

archive\_mode off postmaster

archive\_timeout 0 sighup 0 2147483647

array\_nulls on user

authentication\_timeout 60 sighup 1 600

autovacuum on sighup

autovacuum\_analyze\_scale\_factor 0.1 sighup 0 100

autovacuum\_analyze\_threshold 50 sighup 0 2147483647

autovacuum\_freeze\_max\_age 200000000 postmaster 100000000 2000000000

autovacuum\_max\_workers 3 postmaster 1 8388607

autovacuum\_naptime 60 sighup 1 2147483

autovacuum\_vacuum\_cost\_delay 20 ...

The information displayed in the Database Parameters from postgresql.conf section includes:

| Column Name | Description |
| --- | --- |
| PARAMETER | The name of the parameter. |
| SETTING | The current value assigned to the parameter. |
| CONTEXT | The context required to set the parameter value. |
| MINVAL | The minimum value allowed for the parameter. |
| MAXVAL | The maximum value allowed for the parameter. |

### stat\_db\_rpt()

The signature is:

stat\_db\_rpt(*beginning\_id*, *ending\_id*)

Parameters

beginning\_id

beginning\_id is an integer value that represents the beginning session identifier.

ending\_id

ending\_id is an integer value that represents the ending session identifier.

The following example demonstrates the stat\_db\_rpt() function:

SELECT \* FROM stat\_db\_rpt(9, 10);

stat\_db\_rpt

-----------------------------------------------------------------------------

DATA from pg\_stat\_database

DATABASE NUMBACKENDS XACT COMMIT XACT ROLLBACK BLKS READ BLKS HIT

BLKS ICACHE HIT HIT RATIO ICACHE HIT RATIO

-----------------------------------------------------------------------------

edb 1 21 0 92928 101217

301 52.05 0.15

The information displayed in the DATA from pg\_stat\_database section of the report includes:

| Column Name | Description |
| --- | --- |
| DATABASE | The name of the database. |
| NUMBACKENDS | Number of backends currently connected to this database. This is the only column in this view that returns a value reflecting current state; all other columns return the accumulated values since the last reset. |
| XACT COMMIT | The number of transactions in this database that have been committed. |
| XACT ROLLBACK | The number of transactions in this database that have been rolled back. |
| BLKS READ | The number of blocks read. |
| BLKS HIT | The number of blocks hit. |
| BLKS ICACHE HIT | The number of blocks in Infinite Cache that were hit. |
| HIT RATIO | The percentage of times that a block was found in the shared buffer cache. |
| ICACHE HIT RATIO | The percentage of times that a block was found in Infinite Cache. |

### stat\_tables\_rpt()

The signature is:

function\_name(*beginning\_id*, *ending\_id*, *top\_n, scope*)

Parameters

beginning\_id

beginning\_id is an integer value that represents the beginning session identifier.

ending\_id

ending\_id is an integer value that represents the ending session identifier.

top\_n

top\_n represents the number of rows to return

scope

scope determines which tables the function returns statistics about. Specify SYS, USER or ALL:

* SYS indicates that the function should return information about system defined tables. A table is considered a system table if it is stored in one of the following schemas: pg\_catalog, information\_schema, sys, or dbo.
* USER indicates that the function should return information about user-defined tables.
* ALL specifies that the function should return information about all tables.

The stat\_tables\_rpt() function returns a two-part report. The first portion of the report contains:

SELECT \* FROM stat\_tables\_rpt(18, 19, 10, 'ALL');

stat\_tables\_rpt

-----------------------------------------------------------------------------

DATA from pg\_stat\_all\_tables ordered by seq scan

SCHEMA RELATION

SEQ SCAN REL TUP READ IDX SCAN IDX TUP READ INS UPD DEL

-----------------------------------------------------------------------------

pg\_catalog pg\_class

8 2952 78 65 0 0 0

pg\_catalog pg\_index

4 448 23 28 0 0 0

pg\_catalog pg\_namespace

4 76 1 1 0 0 0

pg\_catalog pg\_database

3 6 0 0 0 0 0

pg\_catalog pg\_authid

2 1 0 0 0 0 0

sys edb$snap

1 15 0 0 1 0 0

public accounts

0 0 0 0 0 0 0

public branches

0 0 0 0 0 0 0

sys edb$session\_wait\_history

0 0 0 0 25 0 0

sys edb$session\_waits

0 0 0 0 10 0 0

The information displayed in the DATA from pg\_stat\_all\_tables ordered by seq scan section includes:

| Column Name | Description |
| --- | --- |
| SCHEMA | The name of the schema in which the table resides. |
| RELATION | The name of the table. |
| SEQ SCAN | The number of sequential scans on the table. |
| REL TUP READ | The number of tuples read from the table. |
| IDX SCAN | The number of index scans performed on the table. |
| IDX TUP READ | The number of index tuples read from the table. |
| INS | The number of rows inserted. |
| UPD | The number of rows updated. |
| DEL | The number of rows deleted. |

The second portion of the report contains:

DATA from pg\_stat\_all\_tables ordered by rel tup read

SCHEMA RELATION

SEQ SCAN REL TUP READ IDX SCAN IDX TUP READ INS UPD DEL

-----------------------------------------------------------------------------

pg\_catalog pg\_class

8 2952 78 65 0 0 0

pg\_catalog pg\_index

4 448 23 28 0 0 0

pg\_catalog pg\_namespace

4 76 1 1 0 0 0

sys edb$snap

1 15 0 0 1 0 0

pg\_catalog pg\_database

3 6 0 0 0 0 0

pg\_catalog pg\_authid

2 1 0 0 0 0 0

public accounts

0 0 0 0 0 0 0

public branches

0 0 0 0 0 0 0

sys edb$session\_wait\_history

0 0 0 0 25 0 0

sys edb$session\_waits

0 0 0 0 10 0 0

(29 rows)

The information displayed in the DATA from pg\_stat\_all\_tables ordered by rel tup read section includes:

| Column Name | Description |
| --- | --- |
| SCHEMA | The name of the schema in which the table resides. |
| RELATION | The name of the table. |
| SEQ SCAN | The number of sequential scans performed on the table. |
| REL TUP READ | The number of tuples read from the table. |
| IDX SCAN | The number of index scans performed on the table. |
| IDX TUP READ | The number of live rows fetched by index scans. |
| INS | The number of rows inserted. |
| UPD | The number of rows updated. |
| DEL | The number of rows deleted. |

### statio\_tables\_rpt()

The signature is:

statio\_tables\_rpt(*beginning\_id*, *ending\_id*, *top\_n*, *scope*)

Parameters

beginning\_id

beginning\_id is an integer value that represents the beginning session identifier.

ending\_id

ending\_id is an integer value that represents the ending session identifier.

top\_n

top\_n represents the number of rows to return

scope

scope determines which tables the function returns statistics about. Specify SYS, USER or ALL:

* SYS indicates that the function should return information about system defined tables. A table is considered a system table if it is stored in one of the following schemas: pg\_catalog, information\_schema, sys, or dbo.
* USER indicates that the function should return information about user-defined tables.
* ALL specifies that the function should return information about all tables.

The statio\_tables\_rpt() function returns a report that contains:

edb=# SELECT \* FROM statio\_tables\_rpt(9, 10, 10, 'SYS');

statio\_tables\_rpt -----------------------------------------------------------------------------

DATA from pg\_statio\_all\_tables

SCHEMA RELATION HEAP HEAP HEAP IDX IDX

READ HIT ICACHE READ HIT

HIT

IDX TOAST TOAST TOAST TIDX TIDX TIDX

ICACHE READ HIT ICACHE READ HIT ICACHE

HIT HIT HIT

-----------------------------------------------------------------------------

public pgbench\_accounts 92766 67215 288 59 32126

9 0 0 0 0 0 0

pg\_catalog pg\_class 0 296 0 3 16

0 0 0 0 0 0 0

sys edb$stat\_all\_indexes 8 125 0 4 233

0 0 0 0 0 0 0

sys edb$statio\_all\_index 8 125 0 4 233

0 0 0 0 0 0 0

sys edb$stat\_all\_tables 6 91 0 2 174

0 0 0 0 0 0 0

sys edb$statio\_all\_table 6 91 0 2 174

0 0 0 0 0 0 0

pg\_catalog pg\_namespace 3 72 0 0 0

0 0 0 0 0 0 0

sys edb$session\_wait\_his 1 24 0 4 47

0 0 0 0 0 0 0

pg\_catalog pg\_opclass 3 13 0 2 0

0 0 0 0 0 0 0

pg\_catalog pg\_trigger 0 12 0 1 15

0 0 0 0 0 0 0

(16 rows)

The information displayed in the Data from pg\_statio\_all\_tables section includes:

| Column Name | Description |
| --- | --- |
| SCHEMA | The name of the schema in which the relation resides. |
| RELATION | The name of the relation. |
| HEAP READ | The number of heap blocks read. |
| HEAP HIT | The number of heap blocks hit. |
| HEAP ICACHE HIT | The number of heap blocks in Infinite Cache. |
| IDX READ | The number of index blocks read. |
| IDX HIT | The number of index blocks hit. |
| IDX ICACHE HIT | The number of index blocks in Infinite Cache. |
| TOAST READ | The number of toast blocks read. |
| TOAST HIT | The number of toast blocks hit. |
| TOAST ICACHE HIT | The number of toast blocks in Infinite Cache. |
| TIDX READ | The number of toast index blocks read. |
| TIDX HIT | The number of toast index blocks hit. |
| TIDX ICACHE HIT | The number of toast index blocks in Infinite Cache. |

### stat\_indexes\_rpt()

The signature is:

stat\_indexes\_rpt(*beginning\_id*, *ending\_id, top\_n, scope*)

Parameters

beginning\_id

beginning\_id is an integer value that represents the beginning session identifier.

ending\_id

ending\_id is an integer value that represents the ending session identifier.

top\_n

top\_n represents the number of rows to return

scope

scope determines which tables the function returns statistics about. Specify SYS, USER or ALL:

* SYS indicates that the function should return information about system defined tables. A table is considered a system table if it is stored in one of the following schemas: pg\_catalog, information\_schema, sys, or dbo.
* USER indicates that the function should return information about user-defined tables.
* ALL specifies that the function should return information about all tables.

The stat\_indexes\_rpt() function returns a report that contains:

edb=# SELECT \* FROM stat\_indexes\_rpt(9, 10, 10, 'ALL');

stat\_indexes\_rpt

-----------------------------------------------------------------------------

DATA from pg\_stat\_all\_indexes

SCHEMA RELATION INDEX

IDX SCAN IDX TUP READ IDX TUP FETCH

-----------------------------------------------------------------------------

pg\_catalog pg\_cast pg\_cast\_source\_target\_index

30 7 7

pg\_catalog pg\_class pg\_class\_oid\_index

15 15 15

pg\_catalog pg\_trigger pg\_trigger\_tgrelid\_tgname\_index

12 12 12

pg\_catalog pg\_attribute pg\_attribute\_relid\_attnum\_index

7 31 31

pg\_catalog pg\_statistic pg\_statistic\_relid\_att\_index

7 0 0

pg\_catalog pg\_database pg\_database\_oid\_index

5 5 5

pg\_catalog pg\_proc pg\_proc\_oid\_index

5 5 5

pg\_catalog pg\_operator pg\_operator\_oprname\_l\_r\_n\_index

3 1 1

pg\_catalog pg\_type pg\_type\_typname\_nsp\_index

3 1 1

pg\_catalog pg\_amop pg\_amop\_opr\_fam\_index

2 3 3

(14 rows)

The information displayed in the DATA from pg\_stat\_all\_indexes section includes:

| Column Name | Description |
| --- | --- |
| SCHEMA | The name of the schema in which the relation resides. |
| RELATION | The name of the relation. |
| INDEX | The name of the index. |
| IDX SCAN | The number of indexes scanned. |
| IDX TUP READ | The number of index tuples read. |
| IDX TUP FETCH | The number of index tuples fetched. |

### statio\_indexes\_rpt()

The signature is:

statio\_indexes\_rpt(*beginning\_id*, *ending\_id, top\_n, scope*)

Parameters

beginning\_id

beginning\_id is an integer value that represents the beginning session identifier.

ending\_id

ending\_id is an integer value that represents the ending session identifier.

top\_n

top\_n represents the number of rows to return

scope

scope determines which tables the function returns statistics about. Specify SYS, USER or ALL:

* SYS indicates that the function should return information about system defined tables. A table is considered a system table if it is stored in one of the following schemas: pg\_catalog, information\_schema, sys, or dbo.
* USER indicates that the function should return information about user-defined tables.
* ALL specifies that the function should return information about all tables.

The statio\_indexes\_rpt()function returns a report that contains:

edb=# SELECT \* FROM statio\_indexes\_rpt(9, 10, 10, 'SYS');

statio\_indexes\_rpt

-----------------------------------------------------------------------------

DATA from pg\_statio\_all\_indexes

SCHEMA RELATION INDEX

IDX BLKS READ IDX BLKS HIT IDX BLKS ICACHE HIT

-----------------------------------------------------------------------------

public pgbench\_accounts pgbench\_accounts\_pkey

59 32126 9

sys edb$stat\_all\_indexes edb$stat\_idx\_pk

4 233 0

sys edb$statio\_all\_indexes edb$statio\_idx\_pk

4 233 0

sys edb$stat\_all\_tables edb$stat\_tab\_pk

2 174 0

sys edb$statio\_all\_tables edb$statio\_tab\_pk

2 174 0

sys edb$session\_wait\_history session\_waits\_hist\_pk

4 47 0

pg\_catalog pg\_cast pg\_cast\_source\_target\_index

1 29 0

pg\_catalog pg\_trigger pg\_trig\_tgrelid\_tgname\_index

1 15 0

pg\_catalog pg\_class pg\_class\_oid\_index

1 14 0

pg\_catalog pg\_statistic pg\_statistic\_relid\_att\_index

2 12 0

(14 rows)

The information displayed in the DATA from pg\_statio\_all\_indexes report includes:

| Column Name | Description |
| --- | --- |
| SCHEMA | The name of the schema in which the relation resides. |
| RELATION | The name of the table on which the index is defined. |
| INDEX | The name of the index. |
| IDX BLKS READ | The number of index blocks read. |
| IDX BLKS HIT | The number of index blocks hit. |
| IDX BLKS ICACHE HIT | The number of index blocks in Infinite Cache that were hit. |

## Performance Tuning Recommendations

To use DRITA reports for performance tuning, review the top five events in a given report, looking for any event that takes a disproportionately large percentage of resources. In a streamlined system, user I/O will probably make up the largest number of waits. Waits should be evaluated in the context of CPU usage and total time; an event may not be significant if it takes 2 minutes out of a total measurement interval of 2 hours, if the rest of the time is consumed by CPU time. The component of response time (CPU "work" time or other "wait" time) that consumes the highest percentage of overall time should be evaluated.

When evaluating events, watch for:

|  |  |
| --- | --- |
| Event type | Description |
| Checkpoint waits | Checkpoint waits may indicate that checkpoint parameters need to be adjusted. |
| WAL-related waits | WAL-related waits may indicate wal\_buffers are under-sized. |
| SQL Parse waits | If the number of waits is high, try to use prepared statements. |
| db file random reads | If high, check that appropriate indexes and statistics exist. |
| db file random writes | If high, may need to decrease bgwriter\_delay. |
| btree random lock acquires | May indicate indexes are being rebuilt. Schedule index builds during less active time. |

Performance reviews should also include careful scrutiny of the hardware, the operating system, the network and the application SQL statements.

## Event Descriptions

|  |  |
| --- | --- |
| Event Name | Description |
| add in shmem lock acquire | Obsolete/unused |
| bgwriter communication lock acquire | The bgwriter (background writer) process has waited for the short-term lock that synchronizes messages between the bgwriter and a backend process. |
| btree vacuum lock acquire | The server has waited for the short-term lock that synchronizes access to the next available vacuum cycle ID. |
| buffer free list lock acquire | The server has waited for the short-term lock that synchronizes access to the list of free buffers (in shared memory). |
| checkpoint lock acquire: | A server process has waited for the short-term lock that prevents simultaneous checkpoints. |
| checkpoint start lock acquire | The server has waited for the short-term lock that synchronizes access to the bgwriter checkpoint schedule. |
| clog control lock acquire | The server has waited for the short-term lock that synchronizes access to the commit log. |
| control file lock acquire | The server has waited for the short-term lock that synchronizes write access to the control file (this should usually be a low number). |
| db file extend | A server process has waited for the operating system while adding a new page to the end of a file. |
| db file read | A server process has waited for the completion of a read (from disk). |
| db file write | A server process has waited for the completion of a write (to disk). |
| db file sync | A server process has waited for the operating system to flush all changes to disk. |
| first buf mapping lock acquire | The server has waited for a short-term lock that synchronizes access to the shared-buffer mapping table. |
| freespace lock acquire | The server has waited for the short-term lock that synchronizes access to the freespace map. |
| Infinite Cache read | The server has waited for an Infinite Cache read request. |
| Infinite Cache write | The server has waited for an Infinite Cache write request. |
| lwlock acquire | The server has waited for a short-term lock that has not been described elsewhere in this section. |
| multi xact gen lock acquire | The server has waited for the short-term lock that synchronizes access to the next available multi-transaction ID (when a SELECT...FOR SHARE statement executes). |
| multi xact member lock acquire | The server has waited for the short-term lock that synchronizes access to the multi-transaction member file (when a SELECT...FOR SHARE statement executes). |
| multi xact offset lock acquire | The server has waited for the short-term lock that synchronizes access to the multi-transaction offset file (when a SELECT...FOR SHARE statement executes). |
| oid gen lock acquire | The server has waited for the short-term lock that synchronizes access to the next available OID (object ID). |
| query plan | The server has computed the execution plan for a SQL statement. |
| rel cache init lock acquire | The server has waited for the short-term lock that prevents simultaneous relation-cache loads/unloads. |
| shmem index lock acquire | The server has waited for the short-term lock that synchronizes access to the shared-memory map. |
| sinval lock acquire | The server has waited for the short-term lock that synchronizes access to the cache invalidation state. |
| sql parse | The server has parsed a SQL statement. |
| subtrans control lock acquire | The server has waited for the short-term lock that synchronizes access to the subtransaction log. |
| tablespace create lock acquire | The server has waited for the short-term lock that prevents simultaneous CREATE TABLESPACE or DROP TABLESPACE commands. |
| two phase state lock acquire | The server has waited for the short-term lock that synchronizes access to the list of prepared transactions. |
| wal insert lock acquire | The server has waited for the short-term lock that synchronizes write access to the write-ahead log. A high number may indicate that WAL buffers are sized too small. |
| wal write lock acquire | The server has waited for the short-term lock that synchronizes write-ahead log flushes. |
| wal file sync | The server has waited for the write-ahead log to sync to disk (related to the wal\_sync\_method parameter which, by default, is 'fsync' - better performance can be gained by changing this parameter to open\_sync). |
| wal flush | The server has waited for the write-ahead log to flush to disk. |
| wal write | The server has waited for a write to the write-ahead log buffer (expect this value to be high). |
| xid gen lock acquire | The server has waited for the short-term lock that synchronizes access to the next available transaction ID. |

## Catalog Views

The following DRITA catalog views provide access to performance information relating to system waits.

### edb$system\_waits

The edb$system\_waits view summarizes the number of waits and the total wait time per session for each wait named. It also displays the average and max wait times. The following example shows the result of a SELECT statement on the edb$system\_waits view:

select \* from sys.edb$system\_waits;

 edb\_id | dbname |wait\_name | wait\_count |avg\_wait | max\_wait | totalwait

--------+--------+-----------+------------+---------+----------+----------

      1 | edb    |db fileread|        301 |0.011516 | 0.629986 | 2.742500

      1 | edb    |wal flush  |        26 |0.010364 | 0.085380 | 0.269452

      1 | edb    |wal write  |        26 |0.010355 | 0.085371 | 0.269232

      1 | edb    |query plan |        277 |0.001367 | 0.049425 | 0.192442

      2 | edb    |wal flush |        28 |0.040443 | 0.095150 | 0.431984

      2 | edb    |wal write |        28 |0.040434 | 0.095093 | 0.431698

      2 | edb    |query plan |        299 |0.001479 | 0.049425 | 0.262596

edb$system\_waits summarizes the following information:

| Column Name | Type | Description |
| --- | --- | --- |
| edb\_id | BIGINT | Wait identifier. |
| dbname | NAME | Name of the database in which the wait occurs. |
| wait\_name | TEXT | Name of the wait event. |
| wait\_count | BIGINT | Number of times the wait event has occurred. |
| avg\_wait | NUMERIC | Average wait time in milliseconds. |
| max\_wait | NUMERIC(50,6) | Maximum wait time in milliseconds. |
| totalwait | NUMERIC(50,6) | Total wait time in milliseconds. |

### edb$session\_waits

The edb$session\_waits view summarizes the number of waits and the total wait time per session for each wait named and identified by backend ID. It also displays the average and max wait times. The following code sample shows the result of a SELECT statement on the edb$session\_waits view:

SELECT \* FROM sys.edb$session\_waits;

edb\_id | dbname | backend\_id |  wait\_name    | wait\_count | avg\_wait\_time | max\_wait\_time| total\_wait\_time |  usename    |  current\_query

--------+--------+------------+---------------+------------+---------------+--------------+-----------------+--------------+---------------------------

1 | edb | 22935 | db file read  | 175 | 0.008399 |  
 0.629986 | 1.469887 | enterprisedb | <IDLE>

1 | edb | 22988 | db file read  | 116 | 0.009556 |  
 0.040627 | 1.108438 | enterprisedb | select \* from edbsnap();

1 | edb | 22988 | wal flush | 26 | 0.010364 |  
 0.085380 | 0.269452 | enterprisedb | select \* from edbsnap();

(3 rows)

edb$session\_waits summarizes the following information:

| Column Name | Type | Description |
| --- | --- | --- |
| edb\_id | BIGINT | Wait identifier. |
| dbname | NAME | Name of the database in which the wait occurs. |
| backend\_id | BIGINT | The backend ID of the process. |
| wait\_name | TEXT | Name of the wait event. |
| wait\_count | BIGINT | Number of times the wait event has occurred. |
| avg\_wait\_time | NUMERIC(50,6) | Average wait time in milliseconds. |
| max\_wait\_time | NUMERIC | Maximum wait time in milliseconds. |
| total\_wait\_time | NUMERIC | Total wait time in milliseconds. |
| use\_name | NAME | The name of the user invoking the query. |
| current\_query | TEXT | The query that is currently executing. |

### edb$session\_wait\_history

The edb$session\_wait\_history view contains the last 25 wait events for each backend ID active during the session. The following code sample shows the result of a SELECT statement on the edb$session\_wait\_history view:

SELECT \* FROM sys.edb$session\_wait\_history;

edb\_id | dbname | backend\_id | seq |   wait\_name   | elapsed | p1 | p2 | p3

--------+--------+------------+-----+---------------+---------+----+----+----

1 | edb    |      22935 |   1 | query plan   |      54 |  0 |  0 | 0

1 | edb    |      22935 |   2 | db file read |    1116 |2689| 8 | 1

1 | edb    |      22935 |   3 | db file read |     983 |1255| 32 |  1

1 | edb    |      22935 |   4 | db file read  |   13717 |2691| 19 |  1

1 | edb    |      22935 |   5 | query plan    |      75 | 0|  0 |  0

1 | edb    |      22935 |   6 | db file read |   11053 |1255|  7 | 1

1 | edb    |      22935 |   7 | db file read  |     404 |2689|  4 |  1

(7 rows)

The edb$session\_wait\_history view includes the following information:

| Column Name | Type | Description |
| --- | --- | --- |
| edb\_id | BIGINT | Wait identifier. |
| dbname | TEXT | Name of the database in which the wait occurs. |
| backend\_id | BIGINT | The session identifier of the process in which the wait occurs. |
| seq | BIGINT | The sequence number of the event (value 1 through 25). |
| wait\_name | TEXT | Name of the wait event. |
| elapsed | BIGINT | Elapsed time in milliseconds. |
| p1 | BIGINT | Wait specific – see table below. |
| p2 | BIGINT | Wait specific – see table below. |
| p3 | BIGINT | Wait specific – see table below. |

The values contained in the p1, p2, and p3 columns are wait-specific. The following waits include information in those columns:

|  |  |  |  |
| --- | --- | --- | --- |
| Wait Name | p1 | p2 | p3 |
| wal file sync | 0 means Fsync  1 means Fdatasync  2 means open  3 means Fsync writethrough  4 means open dsync  For more information, please see the documentation for WAL\_SYNC\_METHOD | unused | unused |
| Infinite Cache write | The Infinite Cache node ID that was written | The file ID from pg\_class.relfilenode | The block number that was written |
| Infinite Cache read | The file ID from pg\_class.relfilenode | The block number that was written | unused |
| db file extend | The file ID from pg\_class.relfilenode | The block number that was extended | Skip Fsync;  1 if True, 0 if False |
| db file read | The file ID from pg\_class.relfilenode | The block number that was read | unused |
| db file write | The file ID from pg\_class.relfilenode | The block number that was written | unused |

For all other event types, the p1, p2, and p3 columns are unused.

# Built-In Utility Packages

This chapter describes the built-in packages that are provided with Advanced Server. For certain packages, non-superusers must be explicitly granted the EXECUTE privilege on the package before using any of the package’s functions or procedures. For most of the built-in packages, EXECUTE privilege has been granted to PUBLIC by default. See the GRANT command for granting privileges.

All built-in packages are owned by the special sys user which must be specified when granting or revoking privileges on built-in packages:

GRANT EXECUTE ON PACKAGE SYS.UTL\_FILE TO john;

Note: When executing a built-in package procedure that has no IN OUT or OUT parameters from within a PL/pgSQL function or trigger, the PERFORM statement must be used as illustrated by the following example:

PERFORM DBMS\_ALERT.SIGNAL('dept\_alert', 'Alert message');

This differs from the manner in which a procedure is executed from within an SPL anonymous block, procedure, function, or trigger as shown by the following example:

DBMS\_ALERT.SIGNAL('dept\_alert', 'Alert message');

Within an SPL program, the package-qualified procedure name is specified without the PERFORM statement.

When executing a built-in package procedure that has a single IN OUT or OUT parameter from within a PL/pgSQL function or trigger, a variable with a data type compatible with the IN OUT or OUT parameter must be assigned the result of the evaluated function as illustrated by the following example:

DECLARE

v\_item VARCHAR(100);

BEGIN

.

.

.

v\_item := DBMS\_PIPE.UNPACK\_MESSAGE(v\_item);

When executing a built-in package procedure that has more than one IN OUT or OUT parameters from within a PL/pgSQL function or trigger, a variable of type RECORD must be assigned the result of the evaluated function. The returned values of the individual IN OUT and OUT parameters can then be referenced from the individual fields of the record in the form, record\_name.parameter\_name where record\_name is the RECORD type variable name and parameter\_name is the name of an IN OUT or OUT parameter declared in the built-in package procedure parameter declaration as illustrated by the following example:

DECLARE

v\_name VARCHAR2(30);

v\_msg VARCHAR2(80);

v\_status INTEGER;

v\_timeout NUMBER(3) := 120;

v\_waitany RECORD;

BEGIN

.

.

.

v\_waitany := DBMS\_ALERT.WAITANY(v\_name,v\_msg,v\_status,v\_timeout);

RAISE INFO 'Alert name : %', v\_waitany.name;

RAISE INFO 'Alert msg : %', v\_waitany.message;

RAISE INFO 'Alert status : %', v\_waitany.status;

See Section 9.1.5 for the parameter declarations of the DBMS\_ALERT.WAITANY procedure.

## DBMS\_ALERT

The DBMS\_ALERT package provides the capability to register for, send, and receive alerts.

Table 7-9‑1 DBMS\_ALERT Functions/Procedures

| Function/Procedure | Return Type | Description |
| --- | --- | --- |
| REGISTER(name) | n/a | Register to be able to receive alerts named, name. |
| REMOVE(name) | n/a | Remove registration for the alert named, name. |
| REMOVEALL | n/a | Remove registration for all alerts. |
| SIGNAL(name, message) | n/a | Signals the alert named, name, with message. |
| WAITANY(name OUT, message OUT, status OUT, timeout) | n/a | Wait for any registered alert to occur. |
| WAITONE(name, message OUT, status OUT, timeout) | n/a | Wait for the specified alert, name, to occur. |

Advanced Server allows a maximum of 500 concurrent alerts. You can use the dbms\_alert.max\_alerts GUC variable (located in the postgresql.conf file) to specify the maximum number of concurrent alerts allowed on a system.

To set a value for the dbms\_alert.max\_alerts variable, open the postgresql.conf file (located by default in /opt/PostgresPlus/9.5AS/data) with your choice of editor, and edit the dbms\_alert.max\_alerts parameter as shown:

dbms\_alert.max\_alerts = alert\_count

alert\_count

alert\_count specifies the maximum number of concurrent alerts. By default, the value of dbms\_alert.max\_alerts is 100. To disable this feature, set dbms\_alert.max\_alerts to 0.

For the dbms\_alert.max\_alerts GUC to function correctly, the custom\_variable\_classes parameter must contain dbms\_alerts:

custom\_variable\_classes = 'dbms\_alert, …'

After editing the postgresql.conf file parameters, you must restart the server for the changes to take effect.

### REGISTER

The REGISTER procedure enables the current session to be notified of the specified alert.

REGISTER(name VARCHAR2)

Parameters

name

Name of the alert to be registered.

Examples

The following anonymous block registers for an alert named, alert\_test, then waits for the signal.

DECLARE

v\_name VARCHAR2(30) := 'alert\_test';

v\_msg VARCHAR2(80);

v\_status INTEGER;

v\_timeout NUMBER(3) := 120;

BEGIN

DBMS\_ALERT.REGISTER(v\_name);

DBMS\_OUTPUT.PUT\_LINE('Registered for alert ' || v\_name);

DBMS\_OUTPUT.PUT\_LINE('Waiting for signal...');

DBMS\_ALERT.WAITONE(v\_name,v\_msg,v\_status,v\_timeout);

DBMS\_OUTPUT.PUT\_LINE('Alert name : ' || v\_name);

DBMS\_OUTPUT.PUT\_LINE('Alert msg : ' || v\_msg);

DBMS\_OUTPUT.PUT\_LINE('Alert status : ' || v\_status);

DBMS\_OUTPUT.PUT\_LINE('Alert timeout: ' || v\_timeout || ' seconds');

DBMS\_ALERT.REMOVE(v\_name);

END;

Registered for alert alert\_test

Waiting for signal...

### REMOVE

The REMOVE procedure unregisters the session for the named alert.

REMOVE(name VARCHAR2)

Parameters

name

Name of the alert to be unregistered.

### REMOVEALL

The REMOVEALL procedure unregisters the session for all alerts.

REMOVEALL

### SIGNAL

The SIGNAL procedure signals the occurrence of the named alert.

SIGNAL(name VARCHAR2, message VARCHAR2)

Parameters

name

Name of the alert.

message

Information to pass with this alert.

Examples

The following anonymous block signals an alert for alert\_test.

DECLARE

v\_name VARCHAR2(30) := 'alert\_test';

BEGIN

DBMS\_ALERT.SIGNAL(v\_name,'This is the message from ' || v\_name);

DBMS\_OUTPUT.PUT\_LINE('Issued alert for ' || v\_name);

END;

Issued alert for alert\_test

### WAITANY

The WAITANY procedure waits for any of the registered alerts to occur.

WAITANY(name OUT VARCHAR2, message OUT VARCHAR2,

status OUT INTEGER, timeout NUMBER)

Parameters

name

Variable receiving the name of the alert.

message

Variable receiving the message sent by the SIGNAL procedure.

status

Status code returned by the operation. Possible values are: 0 – alert occurred; 1 – timeout occurred.

timeout

Time to wait for an alert in seconds.

Examples

The following anonymous block uses the WAITANY procedure to receive an alert named, alert\_test or any\_alert:

DECLARE

v\_name VARCHAR2(30);

v\_msg VARCHAR2(80);

v\_status INTEGER;

v\_timeout NUMBER(3) := 120;

BEGIN

DBMS\_ALERT.REGISTER('alert\_test');

DBMS\_ALERT.REGISTER('any\_alert');

DBMS\_OUTPUT.PUT\_LINE('Registered for alert alert\_test and any\_alert');

DBMS\_OUTPUT.PUT\_LINE('Waiting for signal...');

DBMS\_ALERT.WAITANY(v\_name,v\_msg,v\_status,v\_timeout);

DBMS\_OUTPUT.PUT\_LINE('Alert name : ' || v\_name);

DBMS\_OUTPUT.PUT\_LINE('Alert msg : ' || v\_msg);

DBMS\_OUTPUT.PUT\_LINE('Alert status : ' || v\_status);

DBMS\_OUTPUT.PUT\_LINE('Alert timeout: ' || v\_timeout || ' seconds');

DBMS\_ALERT.REMOVEALL;

END;

Registered for alert alert\_test and any\_alert

Waiting for signal...

An anonymous block in a second session issues a signal for any\_alert:

DECLARE

v\_name VARCHAR2(30) := 'any\_alert';

BEGIN

DBMS\_ALERT.SIGNAL(v\_name,'This is the message from ' || v\_name);

DBMS\_OUTPUT.PUT\_LINE('Issued alert for ' || v\_name);

END;

Issued alert for any\_alert

Control returns to the first anonymous block and the remainder of the code is executed:

Registered for alert alert\_test and any\_alert

Waiting for signal...

Alert name : any\_alert

Alert msg : This is the message from any\_alert

Alert status : 0

Alert timeout: 120 seconds

### WAITONE

The WAITONE procedure waits for the specified registered alert to occur.

WAITONE(name VARCHAR2, message OUT VARCHAR2,

status OUT INTEGER, timeout NUMBER)

Parameters

name

Name of the alert.

message

Variable receiving the message sent by the SIGNAL procedure.

status

Status code returned by the operation. Possible values are: 0 – alert occurred; 1 – timeout occurred.

timeout

Time to wait for an alert in seconds.

Examples

The following anonymous block is similar to the one used in the WAITANY example except the WAITONE procedure is used to receive the alert named, alert\_test.

DECLARE

v\_name VARCHAR2(30) := 'alert\_test';

v\_msg VARCHAR2(80);

v\_status INTEGER;

v\_timeout NUMBER(3) := 120;

BEGIN

DBMS\_ALERT.REGISTER(v\_name);

DBMS\_OUTPUT.PUT\_LINE('Registered for alert ' || v\_name);

DBMS\_OUTPUT.PUT\_LINE('Waiting for signal...');

DBMS\_ALERT.WAITONE(v\_name,v\_msg,v\_status,v\_timeout);

DBMS\_OUTPUT.PUT\_LINE('Alert name : ' || v\_name);

DBMS\_OUTPUT.PUT\_LINE('Alert msg : ' || v\_msg);

DBMS\_OUTPUT.PUT\_LINE('Alert status : ' || v\_status);

DBMS\_OUTPUT.PUT\_LINE('Alert timeout: ' || v\_timeout || ' seconds');

DBMS\_ALERT.REMOVE(v\_name);

END;

Registered for alert alert\_test

Waiting for signal...

Signal sent for alert\_test sent by an anonymous block in a second session:

DECLARE

v\_name VARCHAR2(30) := 'alert\_test';

BEGIN

DBMS\_ALERT.SIGNAL(v\_name,'This is the message from ' || v\_name);

DBMS\_OUTPUT.PUT\_LINE('Issued alert for ' || v\_name);

END;

Issued alert for alert\_test

First session is alerted, control returns to the anonymous block, and the remainder of the code is executed:

Registered for alert alert\_test

Waiting for signal...

Alert name : alert\_test

Alert msg : This is the message from alert\_test

Alert status : 0

Alert timeout: 120 seconds

### Comprehensive Example

The following example uses two triggers to send alerts when the dept table or the emp table is changed. An anonymous block listens for these alerts and displays messages when an alert is received.

The following are the triggers on the dept and emp tables:

CREATE OR REPLACE FUNCTION dept\_alert\_trig() RETURNS TRIGGER

AS $$

DECLARE

v\_action VARCHAR(25);

BEGIN

IF TG\_OP = 'INSERT' THEN

v\_action := ' added department(s) ';

ELSIF TG\_OP = 'UPDATE' THEN

v\_action := ' updated department(s) ';

ELSIF TG\_OP = 'DELETE' THEN

v\_action := ' deleted department(s) ';

END IF;

PERFORM DBMS\_ALERT.SIGNAL('dept\_alert',USER || v\_action || 'on ' ||

TO\_CHAR(CURRENT\_TIMESTAMP, 'DD-MON-YY HH24:MI:SS'));

RETURN NULL;

END;

$$ LANGUAGE 'plpgsql';

CREATE TRIGGER dept\_alert\_trig

AFTER INSERT OR UPDATE OR DELETE ON dept

FOR EACH STATEMENT EXECUTE PROCEDURE dept\_alert\_trig();

CREATE OR REPLACE FUNCTION emp\_alert\_trig() RETURNS TRIGGER

AS $$

DECLARE

v\_action VARCHAR(25);

BEGIN

IF TG\_OP = 'INSERT' THEN

v\_action := ' added employee(s) ';

ELSIF TG\_OP = 'UPDATE' THEN

v\_action := ' updated employee(s) ';

ELSIF TG\_OP = 'DELETE' THEN

v\_action := ' deleted employee(s) ';

END IF;

PERFORM DBMS\_ALERT.SIGNAL('emp\_alert',USER || v\_action || 'on ' ||

TO\_CHAR(CURRENT\_TIMESTAMP, 'DD-MON-YY HH24:MI:SS'));

RETURN NULL;

END;

$$ LANGUAGE 'plpgsql';

CREATE TRIGGER emp\_alert\_trig

AFTER INSERT OR UPDATE OR DELETE ON emp

FOR EACH STATEMENT EXECUTE PROCEDURE emp\_alert\_trig();

The following anonymous block is executed in a session while updates to the dept and emp tables occur in other sessions:

DECLARE

v\_dept\_alert VARCHAR2(30) := 'dept\_alert';

v\_emp\_alert VARCHAR2(30) := 'emp\_alert';

v\_name VARCHAR2(30);

v\_msg VARCHAR2(80);

v\_status INTEGER;

v\_timeout NUMBER(3) := 60;

BEGIN

DBMS\_ALERT.REGISTER(v\_dept\_alert);

DBMS\_ALERT.REGISTER(v\_emp\_alert);

DBMS\_OUTPUT.PUT\_LINE('Registered for alerts dept\_alert and emp\_alert');

DBMS\_OUTPUT.PUT\_LINE('Waiting for signal...');

LOOP

DBMS\_ALERT.WAITANY(v\_name,v\_msg,v\_status,v\_timeout);

EXIT WHEN v\_status != 0;

DBMS\_OUTPUT.PUT\_LINE('Alert name : ' || v\_name);

DBMS\_OUTPUT.PUT\_LINE('Alert msg : ' || v\_msg);

DBMS\_OUTPUT.PUT\_LINE('Alert status : ' || v\_status);

DBMS\_OUTPUT.PUT\_LINE('------------------------------------' ||

'-------------------------');

END LOOP;

DBMS\_OUTPUT.PUT\_LINE('Alert status : ' || v\_status);

DBMS\_ALERT.REMOVEALL;

END;

Registered for alerts dept\_alert and emp\_alert

Waiting for signal...

Note: In the following sessions with users mary and john, PSQL is executed in AUTOCOMMIT off mode. This affects the number of alerts displayed by multiple SQL statements against the same table by the same user. If the PSQL default AUTCOMMIT on mode were used instead, two alerts for user mary on the emp table would be displayed instead of one since there would be two INSERT statements in two separate transactions.

\set AUTOCOMMIT off

The following changes are made by user, mary:

INSERT INTO dept VALUES (50,'FINANCE','CHICAGO');

INSERT INTO emp (empno,ename,deptno) VALUES (9001,'JONES',50);

INSERT INTO emp (empno,ename,deptno) VALUES (9002,'ALICE',50);

COMMIT;

The following change is made by user, john:

INSERT INTO dept VALUES (60,'HR','LOS ANGELES');

COMMIT;

The following is the output displayed by the anonymous block receiving the signals from the triggers:

Registered for alerts dept\_alert and emp\_alert

Waiting for signal...

Alert name : dept\_alert

Alert msg : mary added department(s) on 05-FEB-14 14:45:16

Alert status : 0

-------------------------------------------------------------

Alert name : emp\_alert

Alert msg : mary added employee(s) on 05-FEB-14 14:45:16

Alert status : 0

-------------------------------------------------------------

Alert name : dept\_alert

Alert msg : john added department(s) on 05-FEB-14 14:45:31

Alert status : 0

-------------------------------------------------------------

Alert status : 1

EDB-SPL Procedure successfully completed

## DBMS\_CRYPTO

The DBMS\_CRYPTO package provides functions and procedures that allow you to encrypt or decrypt RAW, BLOB or CLOB data. You can also use DBMS\_CRYPTO functions to generate cryptographically strong random values.

Table 7.7.2 DBMS\_CRYPTO Functions and Procedures

| Function/Procedure | Return Type | Description |
| --- | --- | --- |
| DECRYPT(src, typ, key, iv) | RAW | Decrypts RAW data. |
| DECRYPT(dst INOUT, src, typ, key, iv) | N/A | Decrypts BLOB data. |
| DECRYPT(dst INOUT, src, typ, key, iv) | N/A | Decrypts CLOB data. |
| ENCRYPT(src, typ, key, iv) | RAW | Encrypts RAW data. |
| ENCRYPT(dst INOUT, src, typ, key, iv) | N/A | Encrypts BLOB data. |
| ENCRYPT(dst INOUT, src, typ, key, iv) | N/A | Encrypts CLOB data. |
| HASH(src, typ) | RAW | Applies a hash algorithm to RAW data. |
| HASH(src) | RAW | Applies a hash algorithm to CLOB data. |
| MAC(src, typ, key) | RAW | Returns the hashed MAC value of the given RAW data using the specified hash algorithm and key. |
| MAC(src, typ, key) | RAW | Returns the hashed MAC value of the given CLOB data using the specified hash algorithm and key. |
| RANDOMBYTES(number\_bytes) | RAW | Returns a specified number of cryptographically strong random bytes. |
| RANDOMINTEGER() | INTEGER | Returns a random INTEGER. |
| RANDOMNUMBER() | NUMBER | Returns a random NUMBER. |

DBMS\_CRYPTO functions and procedures support the following error messages:

ORA-28239 - DBMS\_CRYPTO.KeyNull

ORA-28829 - DBMS\_CRYPTO.CipherSuiteNull

ORA-28827 - DBMS\_CRYPTO.CipherSuiteInvalid

Advanced Server will *not* return error ORA-28233 if you re-encrypt previously encrypted information.

Please note that RAW and BLOB are synonyms for the PostgreSQL BYTEA data type, and CLOB is a synonym for TEXT.

### DECRYPT

The DECRYPT function or procedure decrypts data using a user-specified cipher algorithm, key and optional initialization vector. The signature of the DECRYPT function is:

DECRYPT  
 (src IN RAW, typ IN INTEGER, key IN RAW, iv IN RAW   
 DEFAULT NULL) RETURN RAW

The signature of the DECRYPT procedure is:

DECRYPT  
 (dst INOUT BLOB, src IN BLOB, typ IN INTEGER, key IN RAW,   
 iv IN RAW DEFAULT NULL)

or

DECRYPT  
 (dst INOUT CLOB, src IN CLOB, typ IN INTEGER, key IN RAW,   
 iv IN RAW DEFAULT NULL)

When invoked as a procedure, DECRYPT returns BLOB or CLOB data to a user-specified BLOB.

Parameters

dst

dstspecifies the name of a BLOB to which the output of the DECRYPT procedure will be written. The DECRYPT procedure will overwrite any existing data currently in dst.

src

srcspecifies the source data that will be decrypted. If you are invoking DECRYPT as a function, specify RAW data; if invoking DECRYPT as a procedure, specify BLOB or CLOB data.

typ

typspecifies the block cipher type and any modifiers. This should match the type specified when the src was encrypted. Advanced Server supports the following block cipher algorithms, modifiers and cipher suites:

|  |  |
| --- | --- |
| Block Cipher Algorithms | |
| ENCRYPT\_DES | CONSTANT INTEGER := 1; |
| ENCRYPT\_3DES | CONSTANT INTEGER := 3; |
| ENCRYPT\_AES | CONSTANT INTEGER := 4; |
| ENCRYPT\_AES128 | CONSTANT INTEGER := 6; |
| Block Cipher Modifiers | |
| CHAIN\_CBC | CONSTANT INTEGER := 256; |
| CHAIN\_ECB | CONSTANT INTEGER := 768; |
| Block Cipher Padding Modifiers | |
| PAD\_PKCS5 | CONSTANT INTEGER := 4096; |
| PAD\_NONE | CONSTANT INTEGER := 8192; |
| Block Cipher Suites | |
| DES\_CBC\_PKCS5 | CONSTANT INTEGER := ENCRYPT\_DES + CHAIN\_CBC + PAD\_PKCS5; |
| DES3\_CBC\_PKCS5 | CONSTANT INTEGER := ENCRYPT\_3DES + CHAIN\_CBC + PAD\_PKCS5; |
| AES\_CBC\_PKCS5 | CONSTANT INTEGER := ENCRYPT\_AES + CHAIN\_CBC + PAD\_PKCS5; |

key

keyspecifies the user-defined decryption key. This should match the key specified when the *src* was encrypted.

iv

iv(optional) specifies an initialization vector. If an initialization vector was specified when the src was encrypted, you must specify an initialization vector when decrypting the src. The default is NULL.

Examples

The following example uses the DBMS\_CRYPTO.DECRYPT function to decrypt an encrypted password retrieved from the passwords table:

CREATE TABLE passwords

(

principal VARCHAR(90) PRIMARY KEY, -- username

ciphertext RAW(9) -- encrypted password

);

CREATE OR REPLACE FUNCTION get\_password (

username VARCHAR2

) RETURNS RAW

AS $$

DECLARE

typ INTEGER := 4353; -- DBMS\_CRYPTO.DES\_CBC\_PKCS5

key RAW(128) := 'my secret key';

iv RAW(100) := 'my initialization vector';

password RAW(2048);

BEGIN

SELECT ciphertext INTO password FROM passwords WHERE principal = username;

RETURN dbms\_crypto.decrypt(password, typ, key, iv);

END;

$$ LANGUAGE 'plpgsql';

Note that when calling DECRYPT, you must pass the same cipher type, key value and initialization vector that was used when ENCRYPTING the target.

### ENCRYPT

The ENCRYPT function or procedure uses a user-specified algorithm, key, and optional initialization vector to encrypt RAW, BLOB or CLOB data. The signature of the ENCRYPT function is:

ENCRYPT  
 (*src* IN RAW, *typ* IN INTEGER, *key* IN RAW,   
 *iv* IN RAW DEFAULT NULL) RETURN RAW

The signature of the ENCRYPT procedure is:

ENCRYPT  
 (*dst* INOUT BLOB, *src* IN BLOB, *typ* IN INTEGER, *key* IN RAW,   
 *iv* IN RAW DEFAULT NULL)

or

ENCRYPT  
 (*dst* INOUT BLOB, *src* IN CLOB, *typ* IN INTEGER, *key* IN RAW,   
 *iv* IN RAW DEFAULT NULL)

When invoked as a procedure, ENCRYPT returns BLOB or CLOB data to a user-specified BLOB.

Parameters

dst

*dst* specifies the name of a BLOB to which the output of the ENCRYPT procedure will be written. The ENCRYPT procedure will overwrite any existing data currently in *dst*.

src

*src* specifies the source data that will be encrypted. If you are invoking ENCRYPT as a function, specify RAW data; if invoking ENCRYPT as a procedure, specify BLOB or CLOB data.

typ

*typ* specifies the block cipher type that will be used by ENCRYPT, and any modifiers. Advanced Server supports the block cipher algorithms, modifiers and cipher suites listed below:

|  |  |
| --- | --- |
| Block Cipher Algorithms | |
| ENCRYPT\_DES | CONSTANT INTEGER := 1; |
| ENCRYPT\_3DES | CONSTANT INTEGER := 3; |
| ENCRYPT\_AES | CONSTANT INTEGER := 4; |
| ENCRYPT\_AES128 | CONSTANT INTEGER := 6; |
| Block Cipher Modifiers | |
| CHAIN\_CBC | CONSTANT INTEGER := 256; |
| CHAIN\_ECB | CONSTANT INTEGER := 768; |
| Block Cipher Padding Modifiers | |
| PAD\_PKCS5 | CONSTANT INTEGER := 4096; |
| PAD\_NONE | CONSTANT INTEGER := 8192; |
| Block Cipher Suites | |
| DES\_CBC\_PKCS5 | CONSTANT INTEGER := ENCRYPT\_DES + CHAIN\_CBC + PAD\_PKCS5; |
| DES3\_CBC\_PKCS5 | CONSTANT INTEGER := ENCRYPT\_3DES + CHAIN\_CBC + PAD\_PKCS5; |
| AES\_CBC\_PKCS5 | CONSTANT INTEGER := ENCRYPT\_AES + CHAIN\_CBC + PAD\_PKCS5; |

key

*key* specifies the encryption key.

iv

*iv* (optional) specifies an initialization vector. By default, iv is NULL.

Examples

The following example uses the DBMS\_CRYPTO.DES\_CBC\_PKCS5 Block Cipher Suite (a pre-defined set of algorithms and modifiers) to encrypt a value retrieved from the passwords table:

CREATE TABLE passwords

(

principal VARCHAR(90) PRIMARY KEY, -- username

ciphertext RAW(9) -- encrypted password

);

CREATE OR REPLACE FUNCTION set\_password (

username VARCHAR,

cleartext RAW

) RETURNS VOID

AS $$

DECLARE

typ INTEGER := 4353; -- DBMS\_CRYPTO.DES\_CBC\_PKCS5

key RAW(128) := 'my secret key';

iv RAW(100) := 'my initialization vector';

encrypted RAW(2048);

BEGIN

encrypted := dbms\_crypto.encrypt(cleartext, typ, key, iv);

UPDATE passwords SET ciphertext = encrypted WHERE principal = username;

RETURN;

END;

$$ LANGUAGE 'plpgsql';

ENCRYPT uses a key value of my secret key and an initialization vector of my initialization vector when encrypting the password; specify the same key and initialization vector when decrypting the password.

### HASH

The HASH function uses a user-specified algorithm to return the hash value of a RAW or CLOB value. The HASH function is available in three forms:

HASH  
 (*src* IN RAW, *typ* IN INTEGER) RETURN RAW

HASH  
 (*src* IN CLOB, *typ* IN INTEGER) RETURN RAW

Parameters

src

*src* specifies the value for which the hash value will be generated. You can specify a RAW, a BLOB, or a CLOB value.

typ

*typ* specifies the HASH function type. Advanced Server supports the HASH function types listed below:

|  |  |
| --- | --- |
| HASH Functions | |
| HASH\_MD4 | CONSTANT INTEGER := 1; |
| HASH\_MD5 | CONSTANT INTEGER := 2; |
| HASH\_SH1 | CONSTANT INTEGER := 3; |

Examples

The following example uses DBMS\_CRYPTO.HASH to find the md5 hash value of the string, cleartext source:

DECLARE

typ INTEGER := DBMS\_CRYPTO.HASH\_MD5;

hash\_value RAW(100);

BEGIN

hash\_value := DBMS\_CRYPTO.HASH('cleartext source', typ);

END;

### MAC

The MAC function uses a user-specified MAC function to return the hashed MAC value of a RAW or CLOB value. The MAC function is available in three forms:

MAC  
 (src IN RAW, typ IN INTEGER, key IN RAW) RETURN RAW

MAC  
 (src IN CLOB, typ IN INTEGER, key IN RAW) RETURN RAW

Parameters

src

*src* specifies the value for which the MAC value will be generated. Specify a RAW, BLOB, or CLOB value.

typ

*typ* specifies the MAC function used. Advanced Server supports the MAC functions listed below.

|  |  |
| --- | --- |
| MAC Functions | |
| HMAC\_MD5 | CONSTANT INTEGER := 1; |
| HMAC\_SH1 | CONSTANT INTEGER := 2; |

key

*key* specifies the key that will be used to calculate the hashed MAC value.

Examples

The following example finds the hashed MAC value of the string cleartext source:

DECLARE

typ INTEGER := DBMS\_CRYPTO.HMAC\_MD5;

key RAW(100) := 'my secret key';

mac\_value RAW(100);

BEGIN

mac\_value := DBMS\_CRYPTO.MAC('cleartext source', typ, key);

END;

DBMS\_CRYPTO.MAC uses a key value of my secret key when calculating the MAC value of cleartext source.

### RANDOMBYTES

The RANDOMBYTES function returns a RAW value of the specified length, containing cryptographically random bytes. The signature is:

RANDOMBYTES  
 (*number*\_*bytes* IN INTEGER) RETURNS RAW

Parameters

number\_bytes

*number\_bytes* specifies the number of random bytes to be returned

Examples

The following example uses RANDOMBYTES to return a value that is 1024 bytes long:

DECLARE

result RAW(1024);

BEGIN

result := DBMS\_CRYPTO.RANDOMBYTES(1024);

END;

### RANDOMINTEGER

The RANDOMINTEGER() function returns a random INTEGER between 0 and 268,435,455. The signature is:

RANDOMINTEGER() RETURNS INTEGER

Examples

The following example uses the RANDOMINTEGER function to return a cryptographically strong random INTEGER value:

DECLARE

result INTEGER;

BEGIN

result := DBMS\_CRYPTO.RANDOMINTEGER();

DBMS\_OUTPUT.PUT\_LINE(result);

END;

### RANDOMNUMBER

The RANDOMNUMBER() function returns a random NUMBER between 0 and 268,435,455. The signature is:

RANDOMNUMBER() RETURNS NUMBER

Examples

The following example uses the RANDOMNUMBER function to return a cryptographically strong random number:

DECLARE

result NUMBER;

BEGIN

result := DBMS\_CRYPTO.RANDOMNUMBER();

DBMS\_OUTPUT.PUT\_LINE(result);

END;

## DBMS\_JOB

The DBMS\_JOB package provides for the creation, scheduling, and managing of jobs. A job runs a stored procedure which has been previously stored in the database. The SUBMIT procedure is used to create and store a job definition. A job identifier is assigned to a job along with its associated stored procedure and the attributes describing when and how often the job is to be run.

This package relies on the pgAgent scheduler. By default, the Advanced Server installer installs pgAgent, but you must start the pgAgent service manually prior to using DBMS\_JOB. See the readme file, README-pgagent.txt, located in the POSTGRES\_PLUS\_HOME/doc directory for information on starting pgAgent. If you attempt to use this package to schedule a job after un-installing pgAgent, DBMS\_JOB will throw an error. DBMS\_JOB verifies that pgAgent is installed, but does not verify that the service is running.

Table ‑ DBMS\_JOB Functions/Procedures

| Function/Procedure | Function or Procedure | Return Type | Description |
| --- | --- | --- | --- |
| BROKEN(job, broken [, next\_date ]) | Procedure | n/a | Specify that a given job is either broken or not broken. |
| CHANGE(job, what, next\_date, interval, instance, force) | Procedure | n/a | Change the job’s parameters. |
| INTERVAL(job, interval) | Procedure | n/a | Set the execution frequency by means of a date function that is recalculated each time the job is run. This value becomes the next date/time for execution. |
| NEXT\_DATE(job, next\_date) | Procedure | n/a | Set the next date/time the job is to be run. |
| REMOVE(job) | Procedure | n/a | Delete the job definition from the database. |
| RUN(job) | Procedure | n/a | Forces execution of a job even if it is marked broken. |
| SUBMIT(job OUT, what [, next\_date [, interval [, no\_parse ]]]) | Procedure | n/a | Creates a job and stores its definition in the database. |
| WHAT(job, what) | Procedure | n/a | Change the stored procedure run by a job. |

When and how often a job is run is dependent upon two interacting parameters – next\_date and interval. The next\_date parameter is a date/time value that specifies the next date/time when the job is to be executed. The interval parameter is a string that contains a date function that evaluates to a date/time value.

Just prior to any execution of the job, the expression in the interval parameter is evaluated. The resulting value replaces the next\_date value stored with the job. The job is then executed. In this manner, the expression in interval is repeatedly re-evaluated prior to each job execution, supplying the next\_date date/time for the next execution.

The following examples use the following stored procedure, job\_proc, which simply inserts a timestamp into table, jobrun, containing a single VARCHAR2 column.

CREATE TABLE jobrun (

runtime VARCHAR2(40)

);

CREATE OR REPLACE PROCEDURE job\_proc

IS

BEGIN

INSERT INTO jobrun VALUES ('job\_proc run at ' || TO\_CHAR(SYSDATE,

'yyyy-mm-dd hh24:mi:ss'));

END;

### BROKEN

The BROKEN procedure sets the state of a job to either broken or not broken. A broken job cannot be executed except by using the RUN procedure.

BROKEN(job BINARY\_INTEGER, broken BOOLEAN [, next\_date DATE ])

Parameters

job

Identifier of the job to be set as broken or not broken.

broken

If set to TRUE the job’s state is set to broken. If set to FALSE the job’s state is set to not broken. Broken jobs cannot be run except by using the RUN procedure.

next\_date

Date/time when the job is to be run. The default is SYSDATE.

Examples

Set the state of a job with job identifier 104 to broken:

BEGIN

DBMS\_JOB.BROKEN(104,true);

END;

Change the state back to not broken:

BEGIN

DBMS\_JOB.BROKEN(104,false);

END;

### CHANGE

The CHANGE procedure modifies certain job attributes including the stored procedure to be run, the next date/time the job is to be run, and how often it is to be run.

CHANGE(job BINARY\_INTEGER what VARCHAR2, next\_date DATE,

interval VARCHAR2, instance BINARY\_INTEGER, force BOOLEAN)

Parameters

job

Identifier of the job to modify.

what

Stored procedure name. Set this parameter to null if the existing value is to remain unchanged.

next\_date

Date/time when the job is to be run next. Set this parameter to null if the existing value is to remain unchanged.

interval

Date function that when evaluated, provides the next date/time the job is to run. Set this parameter to null if the existing value is to remain unchanged.

instance

This argument is ignored, but is included for compatibility.

force

This argument is ignored, but is included for compatibility.

Examples

Change the job to run next on December 13, 2007. Leave other parameters unchanged.

BEGIN

DBMS\_JOB.CHANGE(104,NULL,TO\_DATE('13-DEC-07','DD-MON-YY'),NULL, NULL,

NULL);

END;

### INTERVAL

The INTERVAL procedure sets the frequency of how often a job is to be run.

INTERVAL(job BINARY\_INTEGER, interval VARCHAR2)

Parameters

job

Identifier of the job to modify.

interval

Date function that when evaluated, provides the next date/time the job is to be run.

Examples

Change the job to run once a week:

BEGIN

DBMS\_JOB.INTERVAL(104,'SYSDATE + 7');

END;

### NEXT\_DATE

The NEXT\_DATE procedure sets the date/time of when the job is to be run next.

NEXT\_DATE(job BINARY\_INTEGER, next\_date DATE)

Parameters

job

Identifier of the job whose next run date is to be set.

next\_date

Date/time when the job is to be run next.

Examples

Change the job to run next on December 14, 2007:

BEGIN

DBMS\_JOB.NEXT\_DATE(104, TO\_DATE('14-DEC-07','DD-MON-YY'));

END;

### REMOVE

The REMOVE procedure deletes the specified job from the database. The job must be resubmitted using the SUBMIT procedure in order to have it executed again. Note that the stored procedure that was associated with the job is not deleted.

REMOVE(job BINARY\_INTEGER)

Parameters

job

Identifier of the job that is to be removed from the database.

Examples

Remove a job from the database:

BEGIN

DBMS\_JOB.REMOVE(104);

END;

### RUN

The RUN procedure forces the job to be run, even if its state is broken.

RUN(job BINARY\_INTEGER)

Parameters

job

Identifier of the job to be run.

Examples

Force a job to be run.

BEGIN

DBMS\_JOB.RUN(104);

END;

### SUBMIT

The SUBMIT procedure creates a job definition and stores it in the database. A job consists of a job identifier, the stored procedure to be executed, when the job is to be first run, and a date function that calculates the next date/time the job is to be run.

SUBMIT(job OUT BINARY\_INTEGER, what VARCHAR2

[, next\_date DATE [, interval VARCHAR2 [, no\_parse BOOLEAN ]]])

Parameters

job

Identifier assigned to the job.

what

Name of the stored procedure to be executed by the job.

next\_date

Date/time when the job is to be run next. The default is SYSDATE.

interval

Date function that when evaluated, provides the next date/time the job is to run. If interval is set to null, then the job is run only once. Null is the default.

no\_parse

If set to TRUE, do not syntax-check the stored procedure upon job creation – check only when the job first executes. If set to FALSE, check the procedure upon job creation. The default is FALSE.

Note: The no\_parse option is not supported in this implementation of SUBMIT(). It is included for compatibility only.

Examples

The following example creates a job using stored procedure, job\_proc. The job will execute immediately and run once a day thereafter as set by the interval parameter, SYSDATE + 1.

DECLARE

jobid INTEGER;

BEGIN

DBMS\_JOB.SUBMIT(jobid,'job\_proc;',SYSDATE,

'SYSDATE + 1');

DBMS\_OUTPUT.PUT\_LINE('jobid: ' || jobid);

END;

jobid: 104

The job immediately executes procedure, job\_proc, populating table, jobrun, with a row:

SELECT \* FROM jobrun;

runtime

-------------------------------------

job\_proc run at 2007-12-11 11:43:25

(1 row)

### WHAT

The WHAT procedure changes the stored procedure that the job will execute.

WHAT(job BINARY\_INTEGER, what VARCHAR2)

Parameters

job

Identifier of the job for which the stored procedure is to be changed.

what

Name of the stored procedure to be executed.

Examples

Change the job to run the list\_emp procedure:

BEGIN

DBMS\_JOB.WHAT(104,'list\_emp;');

END;

## DBMS\_LOB

The DBMS\_LOB package provides the capability to operate on large objects.

Table ‑ DBMS\_LOB Functions/Procedures

| Function/Procedure | Function or Procedure | Return Type | Description |
| --- | --- | --- | --- |
| APPEND(dest\_lob IN OUT, src\_lob) | Procedure | n/a | Appends one large object to another. |
| COMPARE(lob\_1, lob\_2 [, amount [, offset\_1 [, offset\_2 ]]]) | Function | INTEGER | Compares two large objects. |
| CONVERTOBLOB(dest\_lob IN OUT, src\_clob, amount, dest\_offset IN OUT, src\_offset IN OUT, blob\_csid, lang\_context IN OUT, warning OUT) | Procedure | n/a | Converts character data to binary. |
| CONVERTTOCLOB(dest\_lob IN OUT, src\_blob, amount, dest\_offset IN OUT, src\_offset IN OUT, blob\_csid, lang\_context IN OUT, warning OUT) | Procedure | n/a | Converts binary data to character. |
| COPY(dest\_lob IN OUT, src\_lob, amount [, dest\_offset [, src\_offset ]]) | Procedure | n/a | Copies one large object to another. |
| ERASE(lob\_loc IN OUT, amount IN OUT [, offset ]) | Procedure | n/a | Erase a large object. |
| GET\_STORAGE\_LIMIT(lob\_loc) | Function | INTEGER | Get the storage limit for large objects. |
| GETLENGTH(lob\_loc) | Function | INTEGER | Get the length of the large object. |
| INSTR(lob\_loc, pattern [, offset [, nth ]]) | Function | INTEGER | Get the position of the nth occurrence of a pattern in the large object starting at offset. |
| READ(lob\_loc, amount IN OUT, offset, buffer OUT) | Procedure | n/a | Read a large object. |
| SUBSTR(lob\_loc [, amount [, offset ]]) | Function | RAW, VARCHAR2 | Get part of a large object. |
| TRIM(lob\_loc IN OUT, newlen) | Procedure | n/a | Trim a large object to the specified length. |
| WRITE(lob\_loc IN OUT, amount, offset, buffer) | Procedure | n/a | Write data to a large object. |
| WRITEAPPEND(lob\_loc IN OUT, amount, buffer) | Procedure | n/a | Write data from the buffer to the end of a large object. |

The following table lists the public variables available in the package.

Table ‑ DBMS\_LOB Public Variables

| Public Variables | Data Type | Value |
| --- | --- | --- |
| compress off | INTEGER | 0 |
| compress\_on | INTEGER | 1 |
| deduplicate\_off | INTEGER | 0 |
| deduplicate\_on | INTEGER | 4 |
| default\_csid | INTEGER | 0 |
| default\_lang\_ctx | INTEGER | 0 |
| encrypt\_off | INTEGER | 0 |
| encrypt\_on | INTEGER | 1 |
| file\_readonly | INTEGER | 0 |
| lobmaxsize | INTEGER | 1073741823 |
| lob\_readonly | INTEGER | 0 |
| lob\_readwrite | INTEGER | 1 |
| no\_warning | INTEGER | 0 |
| opt\_compress | INTEGER | 1 |
| opt\_deduplicate | INTEGER | 4 |
| opt\_encrypt | INTEGER | 2 |
| warn\_inconvertible\_char | INTEGER | 1 |

In the following sections, lengths and offsets are measured in bytes if the large objects are BLOBs. Lengths and offsets are measured in characters if the large objects are CLOBs.

### APPEND

The APPEND procedure provides the capability to append one large object to another. Both large objects must be of the same type.

APPEND(dest\_lob IN OUT { BLOB | CLOB }, src\_lob { BLOB | CLOB })

Parameters

dest\_lob

Large object locator for the destination object. Must be the same data type as src\_lob.

src\_lob

Large object locator for the source object. Must be the same data type as dest\_lob.

### COMPARE

The COMPARE procedure performs an exact byte-by-byte comparison of two large objects for a given length at given offsets. The large objects being compared must be the same data type.

status INTEGER COMPARE(lob\_1 { BLOB | CLOB },

lob\_2 { BLOB | CLOB }

[, amount INTEGER [, offset\_1 INTEGER [, offset\_2 INTEGER ]]])

Parameters

lob\_1

Large object locator of the first large object to be compared. Must be the same data type as lob\_2.

lob\_2

Large object locator of the second large object to be compared. Must be the same data type as lob\_1.

amount

If the data type of the large objects is BLOB, then the comparison is made for amount bytes. If the data type of the large objects is CLOB, then the comparison is made for amount characters. The default is the maximum size of a large object.

offset\_1

Position within the first large object to begin the comparison. The first byte/character is offset 1. The default is 1.

offset\_2

Position within the second large object to begin the comparison. The first byte/character is offset 1. The default is 1.

status

Zero if both large objects are exactly the same for the specified length for the specified offsets. Non-zero, if the objects are not the same. NULL if amount, offset\_1, or offset\_2 are less than zero.

### CONVERTTOBLOB

The CONVERTTOBLOB procedure provides the capability to convert character data to binary.

CONVERTTOBLOB(dest\_lob IN OUT BLOB, src\_clob CLOB,

amount INTEGER, dest\_offset IN OUT INTEGER,

src\_offset IN OUT INTEGER, blob\_csid NUMBER,

lang\_context IN OUT INTEGER, warning OUT INTEGER)

Parameters

dest\_lob

BLOB large object locator to which the character data is to be converted.

src\_clob

CLOB large object locator of the character data to be converted.

amount

Number of characters of src\_clob to be converted.

dest\_offset IN

Position in bytes in the destination BLOB where writing of the source CLOB should begin. The first byte is offset 1.

dest\_offset OUT

Position in bytes in the destination BLOB after the write operation completes. The first byte is offset 1.

src\_offset IN

Position in characters in the source CLOB where conversion to the destination BLOB should begin. The first character is offset 1.

src\_offset OUT

Position in characters in the source CLOB after the conversion operation completes. The first character is offset 1.

blob\_csid

Character set ID of the converted, destination BLOB.

lang\_context IN

Language context for the conversion. The default value of 0 is typically used for this setting.

lang\_context OUT

Language context after the conversion completes.

warning

0 if the conversion was successful, 1 if an inconvertible character was encountered.

### CONVERTTOCLOB

The CONVERTTOCLOB procedure provides the capability to convert binary data to character.

CONVERTTOCLOB(dest\_lob IN OUT CLOB, src\_blob BLOB,

amount INTEGER, dest\_offset IN OUT INTEGER,

src\_offset IN OUT INTEGER, blob\_csid NUMBER,

lang\_context IN OUT INTEGER, warning OUT INTEGER)

Parameters

dest\_lob

CLOB large object locator to which the binary data is to be converted.

src\_blob

BLOB large object locator of the binary data to be converted.

amount

Number of bytes of src\_blob to be converted.

dest\_offset IN

Position in characters in the destination CLOB where writing of the source BLOB should begin. The first character is offset 1.

dest\_offset OUT

Position in characters in the destination CLOB after the write operation completes. The first character is offset 1.

src\_offset IN

Position in bytes in the source BLOB where conversion to the destination CLOB should begin. The first byte is offset 1.

src\_offset OUT

Position in bytes in the source BLOB after the conversion operation completes. The first byte is offset 1.

blob\_csid

Character set ID of the converted, destination CLOB.

lang\_context IN

Language context for the conversion. The default value of 0 is typically used for this setting.

lang\_context OUT

Language context after the conversion completes.

warning

0 if the conversion was successful, 1 if an inconvertible character was encountered.

### COPY

The COPY procedure provides the capability to copy one large object to another. The source and destination large objects must be the same data type.

COPY(dest\_lob IN OUT { BLOB | CLOB }, src\_lob { BLOB | CLOB },

amount INTEGER

[, dest\_offset INTEGER [, src\_offset INTEGER ]])

Parameters

dest\_lob

Large object locator of the large object to which src\_lob is to be copied. Must be the same data type as src\_lob.

src\_lob

Large object locator of the large object to be copied to dest\_lob. Must be the same data type as dest\_lob.

amount

Number of bytes/characters of src\_lob to be copied.

dest\_offset

Position in the destination large object where writing of the source large object should begin. The first position is offset 1. The default is 1.

src\_offset

Position in the source large object where copying to the destination large object should begin. The first position is offset 1. The default is 1.

### ERASE

The ERASE procedure provides the capability to erase a portion of a large object. To erase a large object means to replace the specified portion with zero-byte fillers for BLOBs or with spaces for CLOBs. The actual size of the large object is not altered.

ERASE(lob\_loc IN OUT { BLOB | CLOB }, amount IN OUT INTEGER

[, offset INTEGER ])

Parameters

lob\_loc

Large object locator of the large object to be erased.

amount IN

Number of bytes/characters to be erased.

amount OUT

Number of bytes/characters actually erased. This value can be smaller than the input value if the end of the large object is reached before amount bytes/characters have been erased.

offset

Position in the large object where erasing is to begin. The first byte/character is position 1. The default is 1.

### GET\_STORAGE\_LIMIT

The GET\_STORAGE\_LIMIT function returns the limit on the largest allowable large object.

size INTEGER GET\_STORAGE\_LIMIT(lob\_loc BLOB)

size INTEGER GET\_STORAGE\_LIMIT(lob\_loc CLOB)

Parameters

size

Maximum allowable size of a large object in this database.

*lob\_loc*

This parameter is ignored, but is included for compatibility.

### GETLENGTH

The GETLENGTH function returns the length of a large object.

amount INTEGER GETLENGTH(lob\_loc BLOB)

amount INTEGER GETLENGTH(lob\_loc CLOB)

Parameters

lob\_loc

Large object locator of the large object whose length is to be obtained.

amount

Length of the large object in bytes for BLOBs or characters for CLOBs.

### INSTR

The INSTR function returns the location of the nth occurrence of a given pattern within a large object.

position INTEGER INSTR(lob\_loc { BLOB | CLOB },

pattern { RAW | VARCHAR2 } [, offset INTEGER [, nth INTEGER ]])

Parameters

lob\_loc

Large object locator of the large object in which to search for pattern.

pattern

Pattern of bytes or characters to match against the large object, lob. pattern must be RAW if lob\_loc is a BLOB. pattern must be VARCHAR2 if lob\_loc is a CLOB.

offset

Position within lob\_loc to start search for pattern. The first byte/character is position 1. The default is 1.

nth

Search for pattern, nth number of times starting at the position given by offset. The default is 1.

position

Position within the large object where pattern appears the nth time specified by nth starting from the position given by offset.

### READ

The READ procedure provides the capability to read a portion of a large object into a buffer.

READ(lob\_loc { BLOB | CLOB }, amount IN OUT BINARY\_INTEGER,

offset INTEGER, buffer OUT { RAW | VARCHAR2 })

Parameters

lob\_loc

Large object locator of the large object to be read.

amount IN

Number of bytes/characters to read.

amount OUT

Number of bytes/characters actually read. If there is no more data to be read, then amount returns 0 and a DATA\_NOT\_FOUND exception is thrown.

offset

Position to begin reading. The first byte/character is position 1.

buffer

Variable to receive the large object. If lob\_loc is a BLOB, then buffer must be RAW. If lob\_loc is a CLOB, then buffer must be VARCHAR2.

### SUBSTR

The SUBSTR function provides the capability to return a portion of a large object.

data { RAW | VARCHAR2 } SUBSTR(lob\_loc { BLOB | CLOB }

[, amount INTEGER [, offset INTEGER ]])

Parameters

lob\_loc

Large object locator of the large object to be read.

amount

Number of bytes/characters to be returned. Default is 32,767.

offset

Position within the large object to begin returning data. The first byte/character is position 1. The default is 1.

data

Returned portion of the large object to be read. If lob\_loc is a BLOB, the return data type is RAW. If lob\_loc is a CLOB, the return data type is VARCHAR2.

### TRIM

The TRIM procedure provides the capability to truncate a large object to the specified length.

TRIM(lob\_loc IN OUT { BLOB | CLOB }, newlen INTEGER)

Parameters

lob\_loc

Large object locator of the large object to be trimmed.

newlen

Number of bytes/characters to which the large object is to be trimmed.

### WRITE

The WRITE procedure provides the capability to write data into a large object. Any existing data in the large object at the specified offset for the given length is overwritten by data given in the buffer.

WRITE(lob\_loc IN OUT { BLOB | CLOB }, amount BINARY\_INTEGER,

offset INTEGER, buffer { RAW | VARCHAR2 })

Parameters

lob\_loc

Large object locator of the large object to be written.

amount

The number of bytes/characters in buffer to be written to the large object.

offset

The offset in bytes/characters from the beginning of the large object (origin is 1) for the write operation to begin.

buffer

Contains data to be written to the large object. If lob\_loc is a BLOB, then buffer must be RAW. If lob\_loc is a CLOB, then buffer must be VARCHAR2.

### WRITEAPPEND

The WRITEAPPEND procedure provides the capability to add data to the end of a large object.

WRITEAPPEND(lob\_loc IN OUT { BLOB | CLOB },

amount BINARY\_INTEGER, buffer { RAW | VARCHAR2 })

Parameters

lob\_loc

Large object locator of the large object to which data is to be appended.

amount

Number of bytes/characters from buffer to be appended the large object.

buffer

Data to be appended to the large object. If lob\_loc is a BLOB, then buffer must be RAW. If lob\_loc is a CLOB, then buffer must be VARCHAR2.

## DBMS\_LOCK

Advanced Server provides support for the DBMS\_LOCK.SLEEP procedure.

Table 7.7.2 DBMS\_LOCK Procedure

| Function/Procedure | Return Type | Description |
| --- | --- | --- |
| SLEEP(seconds) | n/a | Suspends a session for the specified number of seconds. |

### SLEEP

The SLEEP procedure suspends the current session for the specified number of seconds.

SLEEP(seconds NUMBER)

Parameters

seconds

seconds specifies the number of seconds for which you wish to suspend the session. seconds can be a fractional value; for example, enter 1.75 to specify one and three-fourths of a second.

## DBMS\_MVIEW

Use procedures in the DBMS\_MVIEW package to manage and refresh materialized views and their dependencies. Advanced Server provides support for the following DBMS\_MVIEW procedures:

Table 7.7.2 DBMS\_MVIEW Procedures

| Procedure | Return Type | Description |
| --- | --- | --- |
| GET\_MV\_DEPENDENCIES(list VARCHAR2, deplist VARCHAR2); | n/a | The GET\_MV\_DEPENDENCIES procedure returns a list of dependencies for a specified view. |
| REFRESH(list VARCHAR2, method VARCHAR2, rollback\_seg VARCHAR2 , push\_deferred\_rpc BOOLEAN, refresh\_after\_errors BOOLEAN , purge\_option NUMBER, parallelism NUMBER, heap\_size NUMBER , atomic\_refresh BOOLEAN , nested BOOLEAN); | n/a | This variation of the REFRESH procedure refreshes all views named in a comma-separated list of view names. |
| REFRESH(tab dbms\_utility.uncl\_array, method VARCHAR2, rollback\_seg VARCHAR2, push\_deferred\_rpc BOOLEAN, refresh\_after\_errors BOOLEAN, purge\_option NUMBER, parallelism NUMBER, heap\_size NUMBER, atomic\_refresh BOOLEAN, nested BOOLEAN); | n/a | This variation of the REFRESH procedure refreshes all views named in a table of dbms\_utility.uncl\_array values. |
| REFRESH\_ALL\_MVIEWS(number\_of\_failures BINARY\_INTEGER, method VARCHAR2, rollback\_seg VARCHAR2, refresh\_after\_errors BOOLEAN, atomic\_refresh BOOLEAN); | n/a | The REFRESH\_ALL\_MVIEWS procedure refreshes all materialized views. |
| REFRESH\_DEPENDENT(number\_of\_failures BINARY\_INTEGER, list VARCHAR2, method VARCHAR2, rollback\_seg VARCHAR2, refresh\_after\_errors BOOLEAN, atomic\_refresh BOOLEAN, nested BOOLEAN); | n/a | This variation of the REFRESH\_DEPENDENT procedure refreshes all views that are dependent on the views listed in a comma-separated list. |
| REFRESH\_DEPENDENT(number\_of\_failures BINARY\_INTEGER, tab dbms\_utility.uncl\_array, method VARCHAR2, rollback\_seg VARCHAR2, refresh\_after\_errors BOOLEAN, atomic\_refresh BOOLEAN, nested BOOLEAN); | n/a | This variation of the REFRESH\_DEPENDENT procedure refreshes all views that are dependent on the views listed in a table of dbms\_utility.uncl\_array values. |

### GET\_MV\_DEPENDENCIES

When given the name of a materialized view, GET\_MV\_DEPENDENCIES returns a list of items that depend on the specified view. The signature is:

GET\_MV\_DEPENDENCIES(  
list IN VARCHAR2,   
deplist OUT VARCHAR2);

Parameters

list

list specifies the name of a materialized view, or a comma-separated list of materialized view names.

deplist

deplist is a comma-separated list of schema-qualified dependencies. *deplist* is a VARCHAR2 value.

Examples

The following example:

DECLARE  
 deplist VARCHAR2(1000);  
BEGIN  
 DBMS\_MVIEW.GET\_MV\_DEPENDENCIES('public.emp\_view', deplist);  
 DBMS\_OUTPUT.PUT\_LINE('deplist: ' || deplist);  
END;

Displays a list of the dependencies on a materialized view named public.emp\_view.

### REFRESH

Use the REFRESH procedure to refresh all views specified in either a comma-separated list of view names, or a table of DBMS\_UTILITY.UNCL\_ARRAY values. The procedure has two signatures; use the first form when specifying a comma-separated list of view names:

REFRESH(  
 list IN VARCHAR2,   
 method IN VARCHAR2 DEFAULT NULL,   
 rollback\_seg IN VARCHAR2 DEFAULT NULL,   
 push\_deferred\_rpc IN BOOLEAN DEFAULT TRUE,   
refresh\_after\_errors IN BOOLEAN DEFAULT FALSE,   
 purge\_option IN NUMBER DEFAULT 1,   
 parallelism IN NUMBER DEFAULT 0,   
 heap\_size IN NUMBER DEFAULT 0,   
 atomic\_refresh IN BOOLEAN DEFAULT TRUE,   
 nested IN BOOLEAN DEFAULT FALSE);

Use the second form to specify view names in a table of DBMS\_UTILITY.UNCL\_ARRAY values:

REFRESH(  
 tab IN OUT DBMS\_UTILITY.UNCL\_ARRAY,   
 method IN VARCHAR2 DEFAULT NULL,   
 rollback\_seg IN VARCHAR2 DEFAULT NULL,   
 push\_deferred\_rpc IN BOOLEAN DEFAULT TRUE,   
 refresh\_after\_errors IN BOOLEAN DEFAULT FALSE,   
 purge\_option IN NUMBER DEFAULT 1,   
 parallelism IN NUMBER DEFAULT 0,   
 heap\_size IN NUMBER DEFAULT 0,   
 atomic\_refresh IN BOOLEAN DEFAULT TRUE,   
 nested IN BOOLEAN DEFAULT FALSE);

Parameters

list

list is a VARCHAR2 value that specifies the name of a materialized view, or a comma-separated list of materialized view names. The names may be schema-qualified.

tab

tab is a table of DBMS\_UTILITY.UNCL\_ARRAY values that specify the name (or names) of a materialized view.

method

method is a VARCHAR2 value that specifies the refresh method that will be applied to the specified view (or views). The only supported method is C; this performs a complete refresh of the view.

rollback\_seg

rollback\_seg is accepted for compatibility and ignored. The default is NULL.

push\_deferred\_rpc

push\_deferred\_rpc is accepted for compatibility and ignored. The default is TRUE.

refresh\_after\_errors

refresh\_after\_errors is accepted for compatibility and ignored. The default is FALSE.

purge\_option

purge\_option is accepted for compatibility and ignored. The default is 1.

parallelism

parallelism is accepted for compatibility and ignored. The default is 0.

heap\_size IN NUMBER DEFAULT 0,

heap\_size is accepted for compatibility and ignored. The default is 0.

atomic\_refresh

atomic\_refresh is accepted for compatibility and ignored. The default is TRUE.

nested

nested is accepted for compatibility and ignored. The default is FALSE.

Examples

The following example uses DBMS\_MVIEW.REFRESH to perform a COMPLETE refresh on the public.emp\_view materialized view:

EXEC DBMS\_MVIEW.REFRESH(list => 'public.emp\_view', method => 'C');

### REFRESH\_ALL\_MVIEWS

Use the REFRESH\_ALL\_MVIEWS procedure to refresh any materialized views that have not been refreshed since the table or view on which the view depends has been modified. The signature is:

REFRESH\_ALL\_MVIEWS(  
 number\_of\_failures OUT BINARY\_INTEGER,   
 method IN VARCHAR2 DEFAULT NULL,   
 rollback\_seg IN VARCHAR2 DEFAULT NULL,   
 refresh\_after\_errors IN BOOLEAN DEFAULT FALSE,   
 atomic\_refreshIN BOOLEAN DEFAULT TRUE);

Parameters

number\_of\_failures

number\_of\_failures is a BINARY\_INTEGER that specifies the number of failures that occurred during the refresh operation.

method

method is a VARCHAR2 value that specifies the refresh method that will be applied to the specified view (or views). The only supported method is C; this performs a complete refresh of the view.

rollback\_seg

rollback\_seg is accepted for compatibility and ignored. The default is NULL.

refresh\_after\_errors

refresh\_after\_errors is accepted for compatibility and ignored. The default is FALSE.

atomic\_refresh

atomic\_refresh is accepted for compatibility and ignored. The default is TRUE.

Examples

The following example performs a COMPLETE refresh on all materialized views:

DECLARE  
 errors INTEGER;  
BEGIN  
 DBMS\_MVIEW.REFRESH\_ALL\_MVIEWS(errors, method => 'C');  
END;

Upon completion, errors contains the number of failures.

### REFRESH\_DEPENDENT

Use the REFRESH\_DEPENDENT procedure to refresh all material views that are dependent on the views specified in the call to the procedure. You can specify a comma-separated list or provide the view names in a table of DBMS\_UTILITY.UNCL\_ARRAY values.

Use the first form of the procedure to refresh all material views that are dependent on the views specified in a comma-separated list:

REFRESH\_DEPENDENT(  
 number\_of\_failures OUT BINARY\_INTEGER,  
list IN VARCHAR2,  
method IN VARCHAR2 DEFAULT NULL,  
rollback\_seg IN VARCHAR2 DEFAULT NULL  
 refresh\_after\_errors IN BOOLEAN DEFAULT FALSE,  
 atomic\_refresh IN BOOLEAN DEFAULT TRUE,  
nested IN BOOLEAN DEFAULT FALSE);

Use the second form of the procedure to refresh all material views that are dependent on the views specified in a table of DBMS\_UTILITY.UNCL\_ARRAY values:

REFRESH\_DEPENDENT(  
 number\_of\_failures OUT BINARY\_INTEGER,   
 tab IN DBMS\_UTILITY.UNCL\_ARRAY,   
 method IN VARCHAR2 DEFAULT NULL,   
 rollback\_seg IN VARCHAR2 DEFAULT NULL,  
 refresh\_after\_errors IN BOOLEAN DEFAULT FALSE,   
 atomic\_refresh IN BOOLEAN DEFAULT TRUE,   
 nested IN BOOLEAN DEFAULT FALSE);

Parameters

number\_of\_failures

number\_of\_failures is a BINARY\_INTEGER that contains the number of failures that occurred during the refresh operation.

list

list is a VARCHAR2 value that specifies the name of a materialized view, or a comma-separated list of materialized view names. The names may be schema-qualified.

tab

tab is a table of DBMS\_UTILITY.UNCL\_ARRAY values that specify the name (or names) of a materialized view.

method

method is a VARCHAR2 value that specifies the refresh method that will be applied to the specified view (or views). The only supported method is C; this performs a complete refresh of the view.

rollback\_seg

rollback\_seg is accepted for compatibility and ignored. The default is NULL.

refresh\_after\_errors

refresh\_after\_errors is accepted for compatibility and ignored. The default is FALSE.

atomic\_refresh

atomic\_refresh is accepted for compatibility and ignored. The default is TRUE.

nested

nested is accepted for compatibility and ignored. The default is FALSE.

Examples

The following example performs a COMPLETE refresh on all materialized views dependent on a materialized view named emp\_view that resides in the public schema:

DECLARE  
 errors INTEGER;  
BEGIN  
 DBMS\_MVIEW.REFRESH\_DEPENDENT(errors, list => 'public.emp\_view', method => 'C');  
END;

Upon completion, errors contains the number of failures.

## DBMS\_OUTPUT

The DBMS\_OUTPUT package provides the capability to send messages (lines of text) to a message buffer, or get messages from the message buffer. A message buffer is local to a single session. Use the DBMS\_PIPE package to send messages between sessions.

The procedures and functions available in the DBMS\_OUTPUT package are listed in the following table.

Table 7‑9‑5 DBMS\_OUTPUT Functions/Procedures

| Function/Procedure | Return Type | Description |
| --- | --- | --- |
| DISABLE | n/a | Disable the capability to send and receive messages. |
| ENABLE(buffer\_size) | n/a | Enable the capability to send and receive messages. |
| GET\_LINE(line OUT, status OUT) | n/a | Get a line from the message buffer. |
| GET\_LINES(lines OUT, numlines IN OUT) | n/a | Get multiple lines from the message buffer. |
| NEW\_LINE | n/a | Puts an end-of-line character sequence. |
| PUT(item) | n/a | Puts a partial line without an end-of-line character sequence. |
| PUT\_LINE(item) | n/a | Puts a complete line with an end-of-line character sequence. |
| SERVEROUTPUT(stdout) | n/a | Direct messages from PUT, PUT\_LINE, or NEW\_LINE to either standard output or the message buffer. |

The following table lists the public variables available in the DBMS\_OUTPUT package.

Table 7‑9‑6 DBMS\_OUTPUT Public Variables

| Public Variables | Data Type | Value | Description |
| --- | --- | --- | --- |
| chararr | TABLE |  | For message lines. |

### CHARARR

The CHARARR is for storing multiple message lines.

TYPE chararr IS TABLE OF VARCHAR2(32767) INDEX BY BINARY\_INTEGER;

### DISABLE

The DISABLE procedure clears out the message buffer. Any messages in the buffer at the time the DISABLE procedure is executed will no longer be accessible. Any messages subsequently sent with the PUT, PUT\_LINE, or NEW\_LINE procedures are discarded. No error is returned to the sender when the PUT, PUT\_LINE, or NEW\_LINE procedures are executed and messages have been disabled.

Use the ENABLE procedure or SERVEROUTPUT(TRUE) procedure to re-enable the sending and receiving of messages.

DISABLE

Examples

This anonymous block disables the sending and receiving messages in the current session.

BEGIN

DBMS\_OUTPUT.DISABLE;

END;

### ENABLE

The ENABLE procedure enables the capability to send messages to the message buffer or retrieve messages from the message buffer. Running SERVEROUTPUT(TRUE) also implicitly performs the ENABLE procedure.

The destination of a message sent with PUT, PUT\_LINE, or NEW\_LINE depends upon the state of SERVEROUTPUT.

* If the last state of SERVEROUTPUT is TRUE, the message goes to standard output of the command line.
* If the last state of SERVEROUTPUT is FALSE, the message goes to the message buffer.

ENABLE [ (buffer\_size INTEGER) ]

Parameters

buffer\_size

Maximum length of the message buffer in bytes. If a buffer\_size of less than 2000 is specified, the buffer size is set to 2000.

Examples

The following anonymous block enables messages. Setting SERVEROUTPUT(TRUE) forces them to standard output.

BEGIN

DBMS\_OUTPUT.ENABLE;

DBMS\_OUTPUT.SERVEROUTPUT(TRUE);

DBMS\_OUTPUT.PUT\_LINE('Messages enabled');

END;

Messages enabled

The same effect could have been achieved by simply using SERVEROUTPUT(TRUE).

BEGIN

DBMS\_OUTPUT.SERVEROUTPUT(TRUE);

DBMS\_OUTPUT.PUT\_LINE('Messages enabled');

END;

Messages enabled

The following anonymous block enables messages, but setting SERVEROUTPUT(FALSE) directs messages to the message buffer.

BEGIN

DBMS\_OUTPUT.ENABLE;

DBMS\_OUTPUT.SERVEROUTPUT(FALSE);

DBMS\_OUTPUT.PUT\_LINE('Message sent to buffer');

END;

### GET\_LINE

The GET\_LINE procedure provides the capability to retrieve a line of text from the message buffer. Only text that has been terminated by an end-of-line character sequence is retrieved – that is complete lines generated using PUT\_LINE, or by a series of PUT calls followed by a NEW\_LINE call.

GET\_LINE(line OUT VARCHAR2, status OUT INTEGER)

Parameters

line

Variable receiving the line of text from the message buffer.

status

0 if a line was returned from the message buffer, 1 if there was no line to return.

Examples

The following anonymous block writes the emp table out to the message buffer as a comma-delimited string for each row.

EXEC DBMS\_OUTPUT.SERVEROUTPUT(FALSE);

DECLARE

v\_emprec VARCHAR2(120);

CURSOR emp\_cur IS SELECT \* FROM emp ORDER BY empno;

BEGIN

DBMS\_OUTPUT.ENABLE;

FOR i IN emp\_cur LOOP

v\_emprec := i.empno || ',' || i.ename || ',' || i.job || ',' ||

NVL(LTRIM(TO\_CHAR(i.mgr,'9999')),'') || ',' || i.hiredate ||

',' || i.sal || ',' ||

NVL(LTRIM(TO\_CHAR(i.comm,'9990.99')),'') || ',' || i.deptno;

DBMS\_OUTPUT.PUT\_LINE(v\_emprec);

END LOOP;

END;

The following anonymous block reads the message buffer and inserts the messages written by the prior example into a table named messages. The rows in messages are then displayed.

CREATE TABLE messages (

status INTEGER,

msg VARCHAR2(100)

);

DECLARE

v\_line VARCHAR2(100);

v\_status INTEGER := 0;

BEGIN

DBMS\_OUTPUT.GET\_LINE(v\_line,v\_status);

WHILE v\_status = 0 LOOP

INSERT INTO messages VALUES(v\_status, v\_line);

DBMS\_OUTPUT.GET\_LINE(v\_line,v\_status);

END LOOP;

END;

SELECT msg FROM messages;

msg

-----------------------------------------------------------------

7369,SMITH,CLERK,7902,17-DEC-80 00:00:00,800.00,,20

7499,ALLEN,SALESMAN,7698,20-FEB-81 00:00:00,1600.00,300.00,30

7521,WARD,SALESMAN,7698,22-FEB-81 00:00:00,1250.00,500.00,30

7566,JONES,MANAGER,7839,02-APR-81 00:00:00,2975.00,,20

7654,MARTIN,SALESMAN,7698,28-SEP-81 00:00:00,1250.00,1400.00,30

7698,BLAKE,MANAGER,7839,01-MAY-81 00:00:00,2850.00,,30

7782,CLARK,MANAGER,7839,09-JUN-81 00:00:00,2450.00,,10

7788,SCOTT,ANALYST,7566,19-APR-87 00:00:00,3000.00,,20

7839,KING,PRESIDENT,,17-NOV-81 00:00:00,5000.00,,10

7844,TURNER,SALESMAN,7698,08-SEP-81 00:00:00,1500.00,0.00,30

7876,ADAMS,CLERK,7788,23-MAY-87 00:00:00,1100.00,,20

7900,JAMES,CLERK,7698,03-DEC-81 00:00:00,950.00,,30

7902,FORD,ANALYST,7566,03-DEC-81 00:00:00,3000.00,,20

7934,MILLER,CLERK,7782,23-JAN-82 00:00:00,1300.00,,10

(14 rows)

### GET\_LINES

The GET\_LINES procedure provides the capability to retrieve one or more lines of text from the message buffer into a collection. Only text that has been terminated by an end-of-line character sequence is retrieved – that is complete lines generated using PUT\_LINE, or by a series of PUT calls followed by a NEW\_LINE call.

GET\_LINES(lines OUT CHARARR, numlines IN OUT INTEGER)

Parameters

lines

Table receiving the lines of text from the message buffer. See CHARARR for a description of lines.

numlines IN

Number of lines to be retrieved from the message buffer.

numlines OUT

Actual number of lines retrieved from the message buffer. If the output value of numlines is less than the input value, then there are no more lines left in the message buffer.

Examples

The following example uses the GET\_LINES procedure to store all rows from the emp table that were placed on the message buffer, into an array.

EXEC DBMS\_OUTPUT.SERVEROUTPUT(FALSE);

DECLARE

v\_emprec VARCHAR2(120);

CURSOR emp\_cur IS SELECT \* FROM emp ORDER BY empno;

BEGIN

DBMS\_OUTPUT.ENABLE;

FOR i IN emp\_cur LOOP

v\_emprec := i.empno || ',' || i.ename || ',' || i.job || ',' ||

NVL(LTRIM(TO\_CHAR(i.mgr,'9999')),'') || ',' || i.hiredate ||

',' || i.sal || ',' ||

NVL(LTRIM(TO\_CHAR(i.comm,'9990.99')),'') || ',' || i.deptno;

DBMS\_OUTPUT.PUT\_LINE(v\_emprec);

END LOOP;

END;

DECLARE

v\_lines DBMS\_OUTPUT.CHARARR;

v\_numlines INTEGER := 14;

v\_status INTEGER := 0;

BEGIN

DBMS\_OUTPUT.GET\_LINES(v\_lines,v\_numlines);

FOR i IN 1..v\_numlines LOOP

INSERT INTO messages VALUES(v\_numlines, v\_lines(i));

END LOOP;

END;

SELECT msg FROM messages;

msg

-----------------------------------------------------------------

7369,SMITH,CLERK,7902,17-DEC-80 00:00:00,800.00,,20

7499,ALLEN,SALESMAN,7698,20-FEB-81 00:00:00,1600.00,300.00,30

7521,WARD,SALESMAN,7698,22-FEB-81 00:00:00,1250.00,500.00,30

7566,JONES,MANAGER,7839,02-APR-81 00:00:00,2975.00,,20

7654,MARTIN,SALESMAN,7698,28-SEP-81 00:00:00,1250.00,1400.00,30

7698,BLAKE,MANAGER,7839,01-MAY-81 00:00:00,2850.00,,30

7782,CLARK,MANAGER,7839,09-JUN-81 00:00:00,2450.00,,10

7788,SCOTT,ANALYST,7566,19-APR-87 00:00:00,3000.00,,20

7839,KING,PRESIDENT,,17-NOV-81 00:00:00,5000.00,,10

7844,TURNER,SALESMAN,7698,08-SEP-81 00:00:00,1500.00,0.00,30

7876,ADAMS,CLERK,7788,23-MAY-87 00:00:00,1100.00,,20

7900,JAMES,CLERK,7698,03-DEC-81 00:00:00,950.00,,30

7902,FORD,ANALYST,7566,03-DEC-81 00:00:00,3000.00,,20

7934,MILLER,CLERK,7782,23-JAN-82 00:00:00,1300.00,,10

(14 rows)

### NEW\_LINE

The NEW\_LINE procedure writes an end-of-line character sequence in the message buffer.

NEW\_LINE

Parameters

The NEW\_LINE procedure expects no parameters.

### PUT

The PUT procedure writes a string to the message buffer. No end-of-line character sequence is written at the end of the string. Use the NEW\_LINE procedure to add an end-of-line character sequence.

PUT(item VARCHAR2)

Parameters

item

Text written to the message buffer.

Examples

The following example uses the PUT procedure to display a comma-delimited list of employees from the emp table.

DECLARE

CURSOR emp\_cur IS SELECT \* FROM emp ORDER BY empno;

BEGIN

FOR i IN emp\_cur LOOP

DBMS\_OUTPUT.PUT(i.empno);

DBMS\_OUTPUT.PUT(',');

DBMS\_OUTPUT.PUT(i.ename);

DBMS\_OUTPUT.PUT(',');

DBMS\_OUTPUT.PUT(i.job);

DBMS\_OUTPUT.PUT(',');

DBMS\_OUTPUT.PUT(i.mgr);

DBMS\_OUTPUT.PUT(',');

DBMS\_OUTPUT.PUT(i.hiredate);

DBMS\_OUTPUT.PUT(',');

DBMS\_OUTPUT.PUT(i.sal);

DBMS\_OUTPUT.PUT(',');

DBMS\_OUTPUT.PUT(i.comm);

DBMS\_OUTPUT.PUT(',');

DBMS\_OUTPUT.PUT(i.deptno);

DBMS\_OUTPUT.NEW\_LINE;

END LOOP;

END;

7369,SMITH,CLERK,7902,17-DEC-80 00:00:00,800.00,,20

7499,ALLEN,SALESMAN,7698,20-FEB-81 00:00:00,1600.00,300.00,30

7521,WARD,SALESMAN,7698,22-FEB-81 00:00:00,1250.00,500.00,30

7566,JONES,MANAGER,7839,02-APR-81 00:00:00,2975.00,,20

7654,MARTIN,SALESMAN,7698,28-SEP-81 00:00:00,1250.00,1400.00,30

7698,BLAKE,MANAGER,7839,01-MAY-81 00:00:00,2850.00,,30

7782,CLARK,MANAGER,7839,09-JUN-81 00:00:00,2450.00,,10

7788,SCOTT,ANALYST,7566,19-APR-87 00:00:00,3000.00,,20

7839,KING,PRESIDENT,,17-NOV-81 00:00:00,5000.00,,10

7844,TURNER,SALESMAN,7698,08-SEP-81 00:00:00,1500.00,0.00,30

7876,ADAMS,CLERK,7788,23-MAY-87 00:00:00,1100.00,,20

7900,JAMES,CLERK,7698,03-DEC-81 00:00:00,950.00,,30

7902,FORD,ANALYST,7566,03-DEC-81 00:00:00,3000.00,,20

7934,MILLER,CLERK,7782,23-JAN-82 00:00:00,1300.00,,10

### PUT\_LINE

The PUT\_LINE procedure writes a single line to the message buffer including an end-of-line character sequence.

PUT\_LINE(item VARCHAR2)

Parameters

item

Text to be written to the message buffer.

Examples

The following example uses the PUT\_LINE procedure to display a comma-delimited list of employees from the emp table.

DECLARE

v\_emprec VARCHAR2(120);

CURSOR emp\_cur IS SELECT \* FROM emp ORDER BY empno;

BEGIN

FOR i IN emp\_cur LOOP

v\_emprec := i.empno || ',' || i.ename || ',' || i.job || ',' ||

NVL(LTRIM(TO\_CHAR(i.mgr,'9999')),'') || ',' || i.hiredate ||

',' || i.sal || ',' ||

NVL(LTRIM(TO\_CHAR(i.comm,'9990.99')),'') || ',' || i.deptno;

DBMS\_OUTPUT.PUT\_LINE(v\_emprec);

END LOOP;

END;

7369,SMITH,CLERK,7902,17-DEC-80 00:00:00,800.00,,20

7499,ALLEN,SALESMAN,7698,20-FEB-81 00:00:00,1600.00,300.00,30

7521,WARD,SALESMAN,7698,22-FEB-81 00:00:00,1250.00,500.00,30

7566,JONES,MANAGER,7839,02-APR-81 00:00:00,2975.00,,20

7654,MARTIN,SALESMAN,7698,28-SEP-81 00:00:00,1250.00,1400.00,30

7698,BLAKE,MANAGER,7839,01-MAY-81 00:00:00,2850.00,,30

7782,CLARK,MANAGER,7839,09-JUN-81 00:00:00,2450.00,,10

7788,SCOTT,ANALYST,7566,19-APR-87 00:00:00,3000.00,,20

7839,KING,PRESIDENT,,17-NOV-81 00:00:00,5000.00,,10

7844,TURNER,SALESMAN,7698,08-SEP-81 00:00:00,1500.00,0.00,30

7876,ADAMS,CLERK,7788,23-MAY-87 00:00:00,1100.00,,20

7900,JAMES,CLERK,7698,03-DEC-81 00:00:00,950.00,,30

7902,FORD,ANALYST,7566,03-DEC-81 00:00:00,3000.00,,20

7934,MILLER,CLERK,7782,23-JAN-82 00:00:00,1300.00,,10

### SERVEROUTPUT

The SERVEROUTPUT procedure provides the capability to direct messages to standard output of the command line or to the message buffer. Setting SERVEROUTPUT(TRUE) also performs an implicit execution of ENABLE.

In PSQL, SERVEROUTPUT(TRUE) is the default setting.

SERVEROUTPUT(stdout BOOLEAN)

Parameters

stdout

Set to TRUE if subsequent PUT, PUT\_LINE, or NEW\_LINE commands are to send text directly to standard output of the command line. Set to FALSE if text is to be sent to the message buffer.

Examples

The following anonymous block sends the first message to the command line and the second message to the message buffer.

BEGIN

DBMS\_OUTPUT.SERVEROUTPUT(TRUE);

DBMS\_OUTPUT.PUT\_LINE('This message goes to the command line');

DBMS\_OUTPUT.SERVEROUTPUT(FALSE);

DBMS\_OUTPUT.PUT\_LINE('This message goes to the message buffer');

END;

This message goes to the command line

If within the same session, the following anonymous block is executed, the message stored in the message buffer from the prior example is flushed and displayed on the command line as well as the new message.

BEGIN

DBMS\_OUTPUT.SERVEROUTPUT(TRUE);

DBMS\_OUTPUT.PUT\_LINE('Flush messages from the buffer');

END;

This message goes to the message buffer

Flush messages from the buffer

## DBMS\_PIPE

The DBMS\_PIPE package provides the capability to send messages through a pipe within or between sessions connected to the same database cluster.

The procedures and functions available in the DBMS\_PIPE package are listed in the following table.

Table 7‑9‑7 DBMS\_PIPE Functions/Procedures

| Function/Procedure | Return Type | Description |
| --- | --- | --- |
| CREATE\_PIPE(pipename [, maxpipesize ] [, private ]) | INTEGER | Explicitly create a private pipe if private is “true” (the default) or a public pipe if private is “false”. |
| NEXT\_ITEM\_TYPE | INTEGER | Determine the data type of the next item in a received message. |
| PACK\_MESSAGE(item) | n/a | Place item in the session’s local message buffer. |
| PURGE(pipename) | n/a | Remove unreceived messages from the specified pipe. |
| RECEIVE\_MESSAGE(pipename [, timeout ]) | INTEGER | Get a message from a specified pipe. |
| REMOVE\_PIPE(pipename) | INTEGER | Delete an explicitly created pipe. |
| RESET\_BUFFER | n/a | Reset the local message buffer. |
| SEND\_MESSAGE(pipename [, timeout ] [, maxpipesize ]) | INTEGER | Send a message on a pipe. |
| UNIQUE\_SESSION\_NAME | VARCHAR2 | Obtain a unique session name. |
| UNPACK\_MESSAGE(item OUT) | n/a | Retrieve the next data item from a message into a type-compatible variable, item. |

Pipes are categorized as implicit or explicit. An implicit pipe is created if a reference is made to a pipe name that was not previously created by the CREATE\_PIPE function. For example, if the SEND\_MESSAGE function is executed using a non-existent pipe name, a new implicit pipe is created with that name. An explicit pipe is created using the CREATE\_PIPE function whereby the first parameter specifies the pipe name for the new pipe.

Pipes are also categorized as private or public. A private pipe can only be accessed by the user who created the pipe. Even a superuser cannot access a private pipe that was created by another user. A public pipe can be accessed by any user who has access to the DBMS\_PIPE package.

A public pipe can only be created by using the CREATE\_PIPE function with the third parameter set to FALSE. The CREATE\_PIPE function can be used to create a private pipe by setting the third parameter to TRUE or by omitting the third parameter. All implicit pipes are private.

The individual data items or “lines” of a message are first built-in a local message buffer, unique to the current session. The PACK\_MESSAGE procedure builds the message in the session’s local message buffer. The SEND\_MESSAGE function is then used to send the message through the pipe.

Receipt of a message involves the reverse operation. The RECEIVE\_MESSAGE function is used to get a message from the specified pipe. The message is written to the session’s local message buffer. The UNPACK\_MESSAGE procedure is then used to transfer the message data items from the message buffer to program variables. If a pipe contains multiple messages, RECEIVE\_MESSAGE gets the messages in FIFO (first-in-first-out) order.

Each session maintains separate message buffers for messages created with the PACK\_MESSAGE procedure and messages retrieved by the RECEIVE\_MESSAGE function. Thus messages can be both built and received in the same session. However, if consecutive RECEIVE\_MESSAGE calls are made, only the message from the last RECEIVE\_MESSAGE call will be preserved in the local message buffer.

### CREATE\_PIPE

The CREATE\_PIPE function creates an explicit public pipe or an explicit private pipe with a specified name.

status INTEGER CREATE\_PIPE(pipename VARCHAR2

[, maxpipesize INTEGER ] [, private BOOLEAN ])

Parameters

pipename

Name of the pipe.

maxpipesize

Maximum capacity of the pipe in bytes. Default is 8192 bytes.

private

Create a public pipe if set to FALSE. Create a private pipe if set to TRUE. This is the default.

status

Status code returned by the operation. 0 indicates successful creation.

Examples

The following example creates a private pipe named messages:

DECLARE

v\_status INTEGER;

BEGIN

v\_status := DBMS\_PIPE.CREATE\_PIPE('messages');

DBMS\_OUTPUT.PUT\_LINE('CREATE\_PIPE status: ' || v\_status);

END;

CREATE\_PIPE status: 0

The following example creates a public pipe named mailbox:

DECLARE

v\_status INTEGER;

BEGIN

v\_status := DBMS\_PIPE.CREATE\_PIPE('mailbox',8192,FALSE);

DBMS\_OUTPUT.PUT\_LINE('CREATE\_PIPE status: ' || v\_status);

END;

CREATE\_PIPE status: 0

### NEXT\_ITEM\_TYPE

The NEXT\_ITEM\_TYPE function returns an integer code identifying the data type of the next data item in a message that has been retrieved into the session’s local message buffer. As each item is moved off of the local message buffer with the UNPACK\_MESSAGE procedure, the NEXT\_ITEM\_TYPE function will return the data type code for the next available item. A code of 0 is returned when there are no more items left in the message.

typecode INTEGER NEXT\_ITEM\_TYPE

Parameters

typecode

Code identifying the data type of the next data item as shown in Table 7‑9‑8.

Table 7‑9‑8 NEXT\_ITEM\_TYPE Data Type Codes

| Type Code | Data Type |
| --- | --- |
| 0 | No more data items |
| 9 | NUMBER |
| 11 | VARCHAR2 |
| 13 | DATE |
| 23 | RAW |

Examples

The following example shows a pipe packed with a NUMBER item, a VARCHAR2 item, a DATE item, and a RAW item. A second anonymous block then uses the NEXT\_ITEM\_TYPE function to display the type code of each item.

DECLARE

v\_number NUMBER := 123;

v\_varchar VARCHAR2(20) := 'Character data';

v\_date DATE := SYSDATE;

v\_raw RAW(4) := '21222324';

v\_status INTEGER;

BEGIN

DBMS\_PIPE.PACK\_MESSAGE(v\_number);

DBMS\_PIPE.PACK\_MESSAGE(v\_varchar);

DBMS\_PIPE.PACK\_MESSAGE(v\_date);

DBMS\_PIPE.PACK\_MESSAGE(v\_raw);

v\_status := DBMS\_PIPE.SEND\_MESSAGE('datatypes');

DBMS\_OUTPUT.PUT\_LINE('SEND\_MESSAGE status: ' || v\_status);

EXCEPTION

WHEN OTHERS THEN

DBMS\_OUTPUT.PUT\_LINE('SQLERRM: ' || SQLERRM);

DBMS\_OUTPUT.PUT\_LINE('SQLCODE: ' || SQLCODE);

END;

SEND\_MESSAGE status: 0

DECLARE

v\_number NUMBER;

v\_varchar VARCHAR2(20);

v\_date DATE;

v\_timestamp TIMESTAMP;

v\_raw RAW(4);

v\_status INTEGER;

BEGIN

v\_status := DBMS\_PIPE.RECEIVE\_MESSAGE('datatypes');

DBMS\_OUTPUT.PUT\_LINE('RECEIVE\_MESSAGE status: ' || v\_status);

DBMS\_OUTPUT.PUT\_LINE('----------------------------------');

v\_status := DBMS\_PIPE.NEXT\_ITEM\_TYPE;

DBMS\_OUTPUT.PUT\_LINE('NEXT\_ITEM\_TYPE: ' || v\_status);

DBMS\_PIPE.UNPACK\_MESSAGE(v\_number);

DBMS\_OUTPUT.PUT\_LINE('NUMBER Item : ' || v\_number);

DBMS\_OUTPUT.PUT\_LINE('----------------------------------');

v\_status := DBMS\_PIPE.NEXT\_ITEM\_TYPE;

DBMS\_OUTPUT.PUT\_LINE('NEXT\_ITEM\_TYPE: ' || v\_status);

DBMS\_PIPE.UNPACK\_MESSAGE(v\_varchar);

DBMS\_OUTPUT.PUT\_LINE('VARCHAR2 Item : ' || v\_varchar);

DBMS\_OUTPUT.PUT\_LINE('----------------------------------');

v\_status := DBMS\_PIPE.NEXT\_ITEM\_TYPE;

DBMS\_OUTPUT.PUT\_LINE('NEXT\_ITEM\_TYPE: ' || v\_status);

DBMS\_PIPE.UNPACK\_MESSAGE(v\_date);

DBMS\_OUTPUT.PUT\_LINE('DATE Item : ' || v\_date);

DBMS\_OUTPUT.PUT\_LINE('----------------------------------');

v\_status := DBMS\_PIPE.NEXT\_ITEM\_TYPE;

DBMS\_OUTPUT.PUT\_LINE('NEXT\_ITEM\_TYPE: ' || v\_status);

DBMS\_PIPE.UNPACK\_MESSAGE(v\_raw);

DBMS\_OUTPUT.PUT\_LINE('RAW Item : ' || v\_raw);

DBMS\_OUTPUT.PUT\_LINE('----------------------------------');

v\_status := DBMS\_PIPE.NEXT\_ITEM\_TYPE;

DBMS\_OUTPUT.PUT\_LINE('NEXT\_ITEM\_TYPE: ' || v\_status);

DBMS\_OUTPUT.PUT\_LINE('---------------------------------');

EXCEPTION

WHEN OTHERS THEN

DBMS\_OUTPUT.PUT\_LINE('SQLERRM: ' || SQLERRM);

DBMS\_OUTPUT.PUT\_LINE('SQLCODE: ' || SQLCODE);

END;

RECEIVE\_MESSAGE status: 0

----------------------------------

NEXT\_ITEM\_TYPE: 9

NUMBER Item : 123

----------------------------------

NEXT\_ITEM\_TYPE: 11

VARCHAR2 Item : Character data

----------------------------------

NEXT\_ITEM\_TYPE: 13

DATE Item : 02-OCT-07 11:11:43

----------------------------------

NEXT\_ITEM\_TYPE: 23

RAW Item : 21222324

----------------------------------

NEXT\_ITEM\_TYPE: 0

### PACK\_MESSAGE

The PACK\_MESSAGE procedure places an item of data in the session’s local message buffer. PACK\_MESSAGE must be executed at least once before issuing a SEND\_MESSAGE call.

PACK\_MESSAGE(item { DATE | NUMBER | VARCHAR2 | RAW })

Use the UNPACK\_MESSAGE procedure to obtain data items once the message is retrieved using a RECEIVE\_MESSAGE call.

Parameters

item

An expression evaluating to any of the acceptable parameter data types. The value is added to the session’s local message buffer.

### PURGE

The PURGE procedure removes the unreceived messages from a specified implicit pipe.

PURGE(pipename VARCHAR2)

Use the REMOVE\_PIPE function to delete an explicit pipe.

Parameters

pipename

Name of the pipe.

Examples

Two messages are sent on a pipe:

DECLARE

v\_status INTEGER;

BEGIN

DBMS\_PIPE.PACK\_MESSAGE('Message #1');

v\_status := DBMS\_PIPE.SEND\_MESSAGE('pipe');

DBMS\_OUTPUT.PUT\_LINE('SEND\_MESSAGE status: ' || v\_status);

DBMS\_PIPE.PACK\_MESSAGE('Message #2');

v\_status := DBMS\_PIPE.SEND\_MESSAGE('pipe');

DBMS\_OUTPUT.PUT\_LINE('SEND\_MESSAGE status: ' || v\_status);

END;

SEND\_MESSAGE status: 0

SEND\_MESSAGE status: 0

Receive the first message and unpack it:

DECLARE

v\_item VARCHAR2(80);

v\_status INTEGER;

BEGIN

v\_status := DBMS\_PIPE.RECEIVE\_MESSAGE('pipe',1);

DBMS\_OUTPUT.PUT\_LINE('RECEIVE\_MESSAGE status: ' || v\_status);

DBMS\_PIPE.UNPACK\_MESSAGE(v\_item);

DBMS\_OUTPUT.PUT\_LINE('Item: ' || v\_item);

END;

RECEIVE\_MESSAGE status: 0

Item: Message #1

Purge the pipe:

EXEC DBMS\_PIPE.PURGE('pipe');

Try to retrieve the next message. The RECEIVE\_MESSAGE call returns status code 1 indicating it timed out because no message was available.

DECLARE

v\_item VARCHAR2(80);

v\_status INTEGER;

BEGIN

v\_status := DBMS\_PIPE.RECEIVE\_MESSAGE('pipe',1);

DBMS\_OUTPUT.PUT\_LINE('RECEIVE\_MESSAGE status: ' || v\_status);

END;

RECEIVE\_MESSAGE status: 1

### RECEIVE\_MESSAGE

The RECEIVE\_MESSAGE function obtains a message from a specified pipe.

status INTEGER RECEIVE\_MESSAGE(pipename VARCHAR2

[, timeout INTEGER ])

Parameters

pipename

Name of the pipe.

timeout

Wait time (seconds). Default is 86400000 (1000 days).

status

Status code returned by the operation.

The possible status codes are:

Table 7‑9‑9 RECEIVE\_MESSAGE Status Codes

| Status Code | Description |
| --- | --- |
| 0 | Success |
| 1 | Time out |
| 2 | Message too large .for the buffer |

### REMOVE\_PIPE

The REMOVE\_PIPE function deletes an explicit private or explicit public pipe.

status INTEGER REMOVE\_PIPE(pipename VARCHAR2)

Use the REMOVE\_PIPE function to delete explicitly created pipes – i.e., pipes created with the CREATE\_PIPE function.

Parameters

pipename

Name of the pipe.

status

Status code returned by the operation. A status code of 0 is returned even if the named pipe is non-existent.

Examples

Two messages are sent on a pipe:

DECLARE

v\_status INTEGER;

BEGIN

v\_status := DBMS\_PIPE.CREATE\_PIPE('pipe');

DBMS\_OUTPUT.PUT\_LINE('CREATE\_PIPE status : ' || v\_status);

DBMS\_PIPE.PACK\_MESSAGE('Message #1');

v\_status := DBMS\_PIPE.SEND\_MESSAGE('pipe');

DBMS\_OUTPUT.PUT\_LINE('SEND\_MESSAGE status: ' || v\_status);

DBMS\_PIPE.PACK\_MESSAGE('Message #2');

v\_status := DBMS\_PIPE.SEND\_MESSAGE('pipe');

DBMS\_OUTPUT.PUT\_LINE('SEND\_MESSAGE status: ' || v\_status);

END;

CREATE\_PIPE status : 0

SEND\_MESSAGE status: 0

SEND\_MESSAGE status: 0

Receive the first message and unpack it:

DECLARE

v\_item VARCHAR2(80);

v\_status INTEGER;

BEGIN

v\_status := DBMS\_PIPE.RECEIVE\_MESSAGE('pipe',1);

DBMS\_OUTPUT.PUT\_LINE('RECEIVE\_MESSAGE status: ' || v\_status);

DBMS\_PIPE.UNPACK\_MESSAGE(v\_item);

DBMS\_OUTPUT.PUT\_LINE('Item: ' || v\_item);

END;

RECEIVE\_MESSAGE status: 0

Item: Message #1

Remove the pipe:

SELECT DBMS\_PIPE.REMOVE\_PIPE('pipe') FROM DUAL;

remove\_pipe

-------------

0

(1 row)

Try to retrieve the next message. The RECEIVE\_MESSAGE call returns status code 1 indicating it timed out because the pipe had been deleted.

DECLARE

v\_item VARCHAR2(80);

v\_status INTEGER;

BEGIN

v\_status := DBMS\_PIPE.RECEIVE\_MESSAGE('pipe',1);

DBMS\_OUTPUT.PUT\_LINE('RECEIVE\_MESSAGE status: ' || v\_status);

END;

RECEIVE\_MESSAGE status: 1

### RESET\_BUFFER

The RESET\_BUFFER procedure resets a “pointer” to the session’s local message buffer back to the beginning of the buffer. This has the effect of causing subsequent PACK\_MESSAGE calls to overwrite any data items that existed in the message buffer prior to the RESET\_BUFFER call.

RESET\_BUFFER

Examples

A message to John is written to the local message buffer. It is replaced by a message to Bob by calling RESET\_BUFFER. The message is sent on the pipe.

DECLARE

v\_status INTEGER;

BEGIN

DBMS\_PIPE.PACK\_MESSAGE('Hi, John');

DBMS\_PIPE.PACK\_MESSAGE('Can you attend a meeting at 3:00, today?');

DBMS\_PIPE.PACK\_MESSAGE('If not, is tomorrow at 8:30 ok with you?');

DBMS\_PIPE.RESET\_BUFFER;

DBMS\_PIPE.PACK\_MESSAGE('Hi, Bob');

DBMS\_PIPE.PACK\_MESSAGE('Can you attend a meeting at 9:30, tomorrow?');

v\_status := DBMS\_PIPE.SEND\_MESSAGE('pipe');

DBMS\_OUTPUT.PUT\_LINE('SEND\_MESSAGE status: ' || v\_status);

END;

SEND\_MESSAGE status: 0

The message to Bob is in the received message.

DECLARE

v\_item VARCHAR2(80);

v\_status INTEGER;

BEGIN

v\_status := DBMS\_PIPE.RECEIVE\_MESSAGE('pipe',1);

DBMS\_OUTPUT.PUT\_LINE('RECEIVE\_MESSAGE status: ' || v\_status);

DBMS\_PIPE.UNPACK\_MESSAGE(v\_item);

DBMS\_OUTPUT.PUT\_LINE('Item: ' || v\_item);

DBMS\_PIPE.UNPACK\_MESSAGE(v\_item);

DBMS\_OUTPUT.PUT\_LINE('Item: ' || v\_item);

END;

RECEIVE\_MESSAGE status: 0

Item: Hi, Bob

Item: Can you attend a meeting at 9:30, tomorrow?

### SEND\_MESSAGE

The SEND\_MESSAGE function sends a message from the session’s local message buffer to the specified pipe.

status SEND\_MESSAGE(pipename VARCHAR2 [, timeout INTEGER ]

[, maxpipesize INTEGER ])

Parameters

pipename

Name of the pipe.

timeout

Wait time (seconds). Default is 86400000 (1000 days).

maxpipesize

Maximum capacity of the pipe in bytes. Default is 8192 bytes.

status

Status code returned by the operation.

The possible status codes are:

Table 7‑9‑10 SEND\_MESSAGE Status Codes

| Status Code | Description |
| --- | --- |
| 0 | Success |
| 1 | Time out |
| 3 | Function interrupted |

### UNIQUE\_SESSION\_NAME

The UNIQUE\_SESSION\_NAME function returns a name, unique to the current session.

name VARCHAR2 UNIQUE\_SESSION\_NAME

Parameters

name

Unique session name.

Examples

The following anonymous block retrieves and displays a unique session name.

DECLARE

v\_session VARCHAR2(30);

BEGIN

v\_session := DBMS\_PIPE.UNIQUE\_SESSION\_NAME;

DBMS\_OUTPUT.PUT\_LINE('Session Name: ' || v\_session);

END;

Session Name: PG$PIPE$5$2752

### UNPACK\_MESSAGE

The UNPACK\_MESSAGE procedure copies the data items of a message from the local message buffer to a specified program variable. The message must be placed in the local message buffer with the RECEIVE\_MESSAGE function before using UNPACK\_MESSAGE.

UNPACK\_MESSAGE(item OUT { DATE | NUMBER | VARCHAR2 | RAW })

Parameters

item

Type-compatible variable that receives a data item from the local message buffer.

### Comprehensive Example

The following example uses a pipe as a “mailbox”. The procedures to create the mailbox, add a multi-item message to the mailbox (up to three items), and display the full contents of the mailbox are enclosed in a package named, mailbox.

CREATE OR REPLACE PACKAGE mailbox

IS

PROCEDURE create\_mailbox;

PROCEDURE add\_message (

p\_mailbox VARCHAR2,

p\_item\_1 VARCHAR2,

p\_item\_2 VARCHAR2 DEFAULT 'END',

p\_item\_3 VARCHAR2 DEFAULT 'END'

);

PROCEDURE empty\_mailbox (

p\_mailbox VARCHAR2,

p\_waittime INTEGER DEFAULT 10

);

END mailbox;

CREATE OR REPLACE PACKAGE BODY mailbox

IS

PROCEDURE create\_mailbox

IS

v\_mailbox VARCHAR2(30);

v\_status INTEGER;

BEGIN

v\_mailbox := DBMS\_PIPE.UNIQUE\_SESSION\_NAME;

v\_status := DBMS\_PIPE.CREATE\_PIPE(v\_mailbox,1000,FALSE);

IF v\_status = 0 THEN

DBMS\_OUTPUT.PUT\_LINE('Created mailbox: ' || v\_mailbox);

ELSE

DBMS\_OUTPUT.PUT\_LINE('CREATE\_PIPE failed - status: ' ||

v\_status);

END IF;

END create\_mailbox;

PROCEDURE add\_message (

p\_mailbox VARCHAR2,

p\_item\_1 VARCHAR2,

p\_item\_2 VARCHAR2 DEFAULT 'END',

p\_item\_3 VARCHAR2 DEFAULT 'END'

)

IS

v\_item\_cnt INTEGER := 0;

v\_status INTEGER;

BEGIN

DBMS\_PIPE.PACK\_MESSAGE(p\_item\_1);

v\_item\_cnt := 1;

IF p\_item\_2 != 'END' THEN

DBMS\_PIPE.PACK\_MESSAGE(p\_item\_2);

v\_item\_cnt := v\_item\_cnt + 1;

END IF;

IF p\_item\_3 != 'END' THEN

DBMS\_PIPE.PACK\_MESSAGE(p\_item\_3);

v\_item\_cnt := v\_item\_cnt + 1;

END IF;

v\_status := DBMS\_PIPE.SEND\_MESSAGE(p\_mailbox);

IF v\_status = 0 THEN

DBMS\_OUTPUT.PUT\_LINE('Added message with ' || v\_item\_cnt ||

' item(s) to mailbox ' || p\_mailbox);

ELSE

DBMS\_OUTPUT.PUT\_LINE('SEND\_MESSAGE in add\_message failed - ' ||

'status: ' || v\_status);

END IF;

END add\_message;

PROCEDURE empty\_mailbox (

p\_mailbox VARCHAR2,

p\_waittime INTEGER DEFAULT 10

)

IS

v\_msgno INTEGER DEFAULT 0;

v\_itemno INTEGER DEFAULT 0;

v\_item VARCHAR2(100);

v\_status INTEGER;

BEGIN

v\_status := DBMS\_PIPE.RECEIVE\_MESSAGE(p\_mailbox,p\_waittime);

WHILE v\_status = 0 LOOP

v\_msgno := v\_msgno + 1;

DBMS\_OUTPUT.PUT\_LINE('\*\*\*\*\*\* Start message #' || v\_msgno ||

' \*\*\*\*\*\*');

BEGIN

LOOP

v\_status := DBMS\_PIPE.NEXT\_ITEM\_TYPE;

EXIT WHEN v\_status = 0;

DBMS\_PIPE.UNPACK\_MESSAGE(v\_item);

v\_itemno := v\_itemno + 1;

DBMS\_OUTPUT.PUT\_LINE('Item #' || v\_itemno || ': ' ||

v\_item);

END LOOP;

DBMS\_OUTPUT.PUT\_LINE('\*\*\*\*\*\*\* End message #' || v\_msgno ||

' \*\*\*\*\*\*\*');

DBMS\_OUTPUT.PUT\_LINE('\*');

v\_itemno := 0;

v\_status := DBMS\_PIPE.RECEIVE\_MESSAGE(p\_mailbox,1);

END;

END LOOP;

DBMS\_OUTPUT.PUT\_LINE('Number of messages received: ' || v\_msgno);

v\_status := DBMS\_PIPE.REMOVE\_PIPE(p\_mailbox);

IF v\_status = 0 THEN

DBMS\_OUTPUT.PUT\_LINE('Deleted mailbox ' || p\_mailbox);

ELSE

DBMS\_OUTPUT.PUT\_LINE('Could not delete mailbox - status: '

|| v\_status);

END IF;

END empty\_mailbox;

END mailbox;

The following demonstrates the execution of the procedures in mailbox. The first procedure creates a public pipe using a name generated by the UNIQUE\_SESSION\_NAME function.

EXEC mailbox.create\_mailbox;

Created mailbox: PG$PIPE$13$3940

Using the mailbox name, any user in the same database with access to the mailbox package and DBMS\_PIPE package can add messages:

EXEC mailbox.add\_message('PG$PIPE$13$3940','Hi, John','Can you attend a meeting at 3:00, today?','-- Mary');

Added message with 3 item(s) to mailbox PG$PIPE$13$3940

EXEC mailbox.add\_message('PG$PIPE$13$3940','Don''t forget to submit your report','Thanks,','-- Joe');

Added message with 3 item(s) to mailbox PG$PIPE$13$3940

Finally, the contents of the mailbox can be emptied:

EXEC mailbox.empty\_mailbox('PG$PIPE$13$3940');

\*\*\*\*\*\* Start message #1 \*\*\*\*\*\*

Item #1: Hi, John

Item #2: Can you attend a meeting at 3:00, today?

Item #3: -- Mary

\*\*\*\*\*\*\* End message #1 \*\*\*\*\*\*\*

\*

\*\*\*\*\*\* Start message #2 \*\*\*\*\*\*

Item #1: Don't forget to submit your report

Item #2: Thanks,

Item #3: Joe

\*\*\*\*\*\*\* End message #2 \*\*\*\*\*\*\*

\*

Number of messages received: 2

Deleted mailbox PG$PIPE$13$3940

## DBMS\_PROFILER

The DBMS\_PROFILER package collects and stores performance information about the PL/pgSQL and SPL statements that are executed during a performance profiling session; use the functions and procedures listed below to control the profiling tool.

For more information about the DBMS\_PROFILER built-in package (including usage examples and a reference guide to the DBMS\_PROFILER tables and views), see Section 9.9.

Table ‑ DBMS\_PROFILER Functions/Procedures

| Function/Procedure | Function or Procedure | Return Type | Description |
| --- | --- | --- | --- |
| FLUSH\_DATA | Both | Status Code or Exception | Flushes performance data collected in the current session without terminating the session (profiling continues). |
| GET\_VERSION(major OUT, minor OUT) | Procedure | n/a | Returns the version number of this package. |
| INTERNAL\_VERSION\_CHECK | Function | Status Code | Confirms that the current version of the profiler will work with the current database. |
| PAUSE\_PROFILER | Both | Status Code or Exception | Pause data collection. |
| RESUME\_PROFILER | Both | Status Code or Exception | Resume data collection. |
| START\_PROFILER(run\_comment, run\_comment1 [, run\_number OUT ]) | Both | Status Code or Exception | Start data collection. |
| STOP\_PROFILER | Both | Status Code or Exception | Stop data collection and flush performance data to the PLSQL\_PROFILER\_RAWDATA table. |

The functions within the DBMS\_PROFILER package return a status code to indicate success or failure; the DBMS\_PROFILER procedures raise an exception only if they encounter a failure. The status codes and messages returned by the functions, and the exceptions raised by the procedures are listed in the table below.

Table ‑ DBMS\_PROFILER Status Codes and Exceptions

| Status Code | Message | Exception | Description |
| --- | --- | --- | --- |
| -1 | error version | version\_mismatch | The profiler version and the database are incompatible. |
| 0 | success | n/a | The operation completed successfully. |
| 1 | error\_param | profiler\_error | The operation received an incorrect parameter. |
| 2 | error\_io | profiler\_error | The data flush operation has failed. |

### FLUSH\_DATA

The FLUSH\_DATA function/procedure flushes the data collected in the current session without terminating the profiler session. The data is flushed to the tables described in the Advanced Server Performance Features Guide. The function and procedure signatures are:

status INTEGER FLUSH\_DATA

FLUSH\_DATA

Parameters

status

Status code returned by the operation.

### GET\_VERSION

The GET\_VERSION procedure returns the version of DBMS\_PROFILER. The procedure signature is:

GET\_VERSION(major OUT INTEGER, minor OUT INTEGER)

Parameters

major

The major version number of DBMS\_PROFILER.

minor

The minor version number of DBMS\_PROFILER.

### INTERNAL\_VERSION\_CHECK

The INTERNAL\_VERSION\_CHECK function confirms that the current version of DBMS\_PROFILER will work with the current database. The function signature is:

status INTEGER INTERNAL\_VERSION\_CHECK

Parameters

status

Status code returned by the operation.

### PAUSE\_PROFILER

The PAUSE\_PROFILER function/procedure pauses a profiling session. The function and procedure signatures are:

status INTEGER PAUSE\_PROFILER

PAUSE\_PROFILER

Parameters

status

Status code returned by the operation.

### RESUME\_PROFILER

The RESUME\_PROFILER function/procedure pauses a profiling session. The function and procedure signatures are:

status INTEGER RESUME\_PROFILER

RESUME\_PROFILER

Parameters

status

Status code returned by the operation.

### START\_PROFILER

The START\_PROFILER function/procedure starts a data collection session. The function and procedure signatures are:

status INTEGER START\_PROFILER(run\_comment TEXT := SYSDATE,

run\_comment1 TEXT := '' [, run\_number OUT INTEGER ])

START\_PROFILER(run\_comment TEXT := SYSDATE,

run\_comment1 TEXT := '' [, run\_number OUT INTEGER ])

Parameters

run\_comment

A user-defined comment for the profiler session. The default value is SYSDATE.

run\_comment1

An additional user-defined comment for the profiler session. The default value is ''.

run\_number

The session number of the profiler session.

status

Status code returned by the operation.

### STOP\_PROFILER

The STOP\_PROFILER function/procedure stops a profiling session and flushes the performance information to the DBMS\_PROFILER tables and view. The function and procedure signatures are:

status INTEGER STOP\_PROFILER

STOP\_PROFILER

Parameters

status

Status code returned by the operation.

## DBMS\_RANDOM

The DBMS\_RANDOM package provides a number of methods to generate random values. The procedures and functions available in the DBMS\_RANDOM package are listed in the following table.

Table 7. DBMS\_RANDOM Functions/Procedures

| Function/Procedure | Return Type | Description |
| --- | --- | --- |
| INITIALIZE(val) | n/a | Initializes the DBMS\_RANDOM package with the specified seed value. Deprecated, but supported for backward compatibility. |
| NORMAL() | NUMBER | Returns a random NUMBER. |
| RANDOM | INTEGER | Returns a random INTEGER with a value greater than or equal to -2^31 and less than 2^31. Deprecated, but supported for backward compatibility. |
| SEED(val) | n/a | Resets the seed with the specified value. |
| SEED(val) | n/a | Resets the seed with the specified value. |
| STRING(opt, len) | VARCHAR2 | Returns a random string. |
| TERMINATE | n/a | TERMINATE has no effect. Deprecated, but supported for backward compatibility. |
| VALUE | NUMBER | Returns a random number with a value greater than or equal to 0 and less than 1, with 38 digit precision. |
| VALUE(low, high) | NUMBER | Returns a random number with a value greater than or equal to low and less than high. |

### INITIALIZE

The INITIALIZE procedure initializes the DBMS\_RANDOM package with a seed value. The signature is:

INITIALIZE(val IN INTEGER)

This procedure should be considered deprecated; it is included for backward compatibility only.

Parameters

val

val is the seed value used by the DBMS\_RANDOM package algorithm.

Example

The following code snippet demonstrates a call to the INITIALIZE procedure that initializes the DBMS\_RANDOM package with the seed value, 6475.

DBMS\_RANDOM.INITIALIZE(6475);

### NORMAL

The NORMAL function returns a random number of type NUMBER. The signature is:

result NUMBER NORMAL()

Parameters

result

result is a random value of type NUMBER.

Example

The following code snippet demonstrates a call to the NORMAL function:

x:= DBMS\_RANDOM.NORMAL();

### RANDOM

The RANDOM function returns a random INTEGER value that is greater than or equal to -2 ^31 and less than 2 ^31. The signature is:

result INTEGER RANDOM()

This function should be considered deprecated; it is included for backward compatibility only.

Parameters

result

result is a random value of type INTEGER.

Example

The following code snippet demonstrates a call to the RANDOM function. The call returns a random number:

x := DBMS\_RANDOM.RANDOM();

### SEED

The first form of the SEED procedure resets the seed value for the DBMS\_RANDOM package with an INTEGER value. The SEED procedure is available in two forms; the signature of the first form is:

SEED(val IN INTEGER)

Parameters

val

val is the seed value used by the DBMS\_RANDOM package algorithm.

Example

The following code snippet demonstrates a call to the SEED procedure; the call sets the seed value at 8495.

DBMS\_RANDOM.SEED(8495);

### SEED

The second form of the SEED procedure resets the seed value for the DBMS\_RANDOM package with a string value. The SEED procedure is available in two forms; the signature of the second form is:

SEED(val IN VARCHAR2)

Parameters

val

val is the seed value used by the DBMS\_RANDOM package algorithm.

Example

The following code snippet demonstrates a call to the SEED procedure; the call sets the seed value to abc123.

DBMS\_RANDOM.SEED('abc123');

### STRING

The STRING function returns a random VARCHAR2 string in a user-specified format. The signature of the STRING function is:

result VARCHAR2 STRING(opt IN CHAR, len IN NUMBER)

Parameters

opt

Formatting option for the returned string. option may be:

|  |  |
| --- | --- |
| **Option** | **Specifies Formatting Option** |
| u or U | Uppercase alpha string |
| l or L | Lowercase alpha string |
| a or A | Mixed case string |
| x or X | Uppercase alpha-numeric string |
| p or P | Any printable characters |

len

The length of the returned string.

result

result is a random value of type VARCHAR2.

Example

The following code snippet demonstrates a call to the STRING function; the call returns a random alpha-numeric character string that is 10 characters long.

x := DBMS\_RANDOM.STRING('X', 10);

### TERMINATE

The TERMINATE procedure has no effect. The signature is:

TERMINATE

The TERMINATE procedure should be considered deprecated; the procedure is supported for compatibility only.

### VALUE

The VALUE function returns a random NUMBER that is greater than or equal to 0, and less than 1, with 38 digit precision. The VALUE function has two forms; the signature of the first form is:

result NUMBER VALUE()

Parameters

result

result is a random value of type NUMBER.

Example

The following code snippet demonstrates a call to the VALUE function. The call returns a random NUMBER:

x := DBMS\_RANDOM.VALUE();

### VALUE

The VALUE function returns a random NUMBER with a value that is between user-specified boundaries. The VALUE function has two forms; the signature of the second form is:

result NUMBER VALUE(low IN NUMBER, high IN NUMBER)

Parameters

low

low specifies the lower boundary for the random value. The random value may be equal to low.

high

high specifies the upper boundary for the random value; the random value will be less than high.

result

result is a random value of type NUMBER.

Example

The following code snippet demonstrates a call to the VALUE function. The call returns a random NUMBER with a value that is greater than or equal to 1 and less than 100:

x := DBMS\_RANDOM.VALUE(1, 100);

## DBMS\_RLS

The DBMS\_RLS package enables the implementation of Virtual Private Database on certain Advanced Server database objects.

Table ‑ DBMS\_RLS Functions/Procedures

| Function/Procedure | Function or Procedure | Return Type | Description |
| --- | --- | --- | --- |
| ADD\_POLICY(object\_schema, object\_name, policy\_name, function\_schema, policy\_function [, statement\_types [, update\_check [, enable [, static\_policy [, policy\_type [, long\_predicate [, sec\_relevant\_cols [, sec\_relevant\_cols\_opt ]]]]]]]]) | Procedure | n/a | Add a security policy to a database object. |
| DROP\_POLICY(object\_schema, object\_name, policy\_name) | Procedure | n/a | Remove a security policy from a database object. |
| ENABLE\_POLICY(object\_schema, object\_name, policy\_name, enable) | Procedure | n/a | Enable or disable a security policy. |

Virtual Private Database is a type of fine-grained access control using security policies. Fine-grained access control in Virtual Private Database means that access to data can be controlled down to specific rows as defined by the security policy.

The rules that encode a security policy are defined in a policy function, which is an SPL function with certain input parameters and return value. The security policy is the named association of the policy function to a particular database object, typically a table.

Note: In Advanced Server, the policy function can be written in any language supported by Advanced Server such as SQL and PL/pgSQL in addition to SPL.

Note: The database objects currently supported by Advanced Server Virtual Private Database are tables. Policies cannot be applied to views or synonyms.

The advantages of using Virtual Private Database are the following:

* Provides a fine-grained level of security. Database object level privileges given by the GRANT command determine access privileges to the entire instance of a database object, while Virtual Private Database provides access control for the individual rows of a database object instance.
* A different security policy can be applied depending upon the type of SQL command (INSERT, UPDATE, DELETE, or SELECT).
* The security policy can vary dynamically for each applicable SQL command affecting the database object depending upon factors such as the session user of the application accessing the database object.
* Invocation of the security policy is transparent to all applications that access the database object and thus, individual applications do not have to be modified to apply the security policy.
* Once a security policy is enabled, it is not possible for any application (including new applications) to circumvent the security policy except by the system privilege noted by the following.
* Even superusers cannot circumvent the security policy except by the system privilege noted by the following.

Note: The only way security policies can be circumvented is if the EXEMPT ACCESS POLICY system privilege has been granted to a user. The EXEMPT ACCESS POLICY privilege should be granted with extreme care as a user with this privilege is exempted from all policies in the database.

The DBMS\_RLS package provides procedures to create policies, remove policies, enable policies, and disable policies.

The process for implementing Virtual Private Database is as follows:

* Create a policy function. The function must have two input parameters of type VARCHAR2. The first input parameter is for the schema containing the database object to which the policy is to apply and the second input parameter is for the name of that database object. The function must have a VARCHAR2 return type. The function must return a string in the form of a WHERE clause predicate. This predicate is dynamically appended as an AND condition to the SQL command that acts upon the database object. Thus, rows that do not satisfy the policy function predicate are filtered out from the SQL command result set.
* Use the ADD\_POLICY procedure to define a new policy, which is the association of a policy function with a database object. With the ADD\_POLICY procedure, you can also specify the types of SQL commands (INSERT, UPDATE, DELETE, or SELECT) to which the policy is to apply, whether or not to enable the policy at the time of its creation, and if the policy should apply to newly inserted rows or the modified image of updated rows.
* Use the ENABLE\_POLICY procedure to disable or enable an existing policy.
* Use the DROP\_POLICY procedure to remove an existing policy. The DROP\_POLICY procedure does not drop the policy function or the associated database object.

Once policies are created, they can be viewed in the catalog views ALL\_POLICIES (see Section 10.11), DBA\_POLICIES (see Section 10.35), or USER\_POLICIES (see Section 10.62).

The SYS\_CONTEXT function is often used with DBMS\_RLS. The signature is:

SYS\_CONTEXT(namespace, attribute)

Where:

namespace is a VARCHAR2; the only accepted value is USERENV. Any other value will return NULL.

attribute is a VARCHAR2. attribute may be:

|  |  |
| --- | --- |
| attribute Value | Equivalent Value |
| SESSION\_USER | pg\_catalog.session\_user |
| CURRENT\_USER | pg\_catalog.current\_user |
| CURRENT\_SCHEMA | pg\_catalog.current\_schema |
| HOST | pg\_catalog.inet\_host |
| IP\_ADDRESS | pg\_catalog.inet\_client\_addr |
| SERVER\_HOST | pg\_catalog.inet\_server\_addr |

Note: The examples used to illustrate the DBMS\_RLS package are based on a modified copy of the sample emp table provided with Advanced Server along with a role named salesmgr that is granted all privileges on the table. You can create the modified copy of the emp table named vpemp and the salesmgr role as shown by the following:

CREATE TABLE public.vpemp AS SELECT empno, ename, job, sal, comm, deptno FROM emp;

ALTER TABLE vpemp ADD authid VARCHAR2(12);

UPDATE vpemp SET authid = 'researchmgr' WHERE deptno = 20;

UPDATE vpemp SET authid = 'salesmgr' WHERE deptno = 30;

SELECT \* FROM vpemp;

empno | ename | job | sal | comm | deptno | authid

-------+--------+-----------+---------+---------+--------+-------------

7782 | CLARK | MANAGER | 2450.00 | | 10 |

7839 | KING | PRESIDENT | 5000.00 | | 10 |

7934 | MILLER | CLERK | 1300.00 | | 10 |

7369 | SMITH | CLERK | 800.00 | | 20 | researchmgr

7566 | JONES | MANAGER | 2975.00 | | 20 | researchmgr

7788 | SCOTT | ANALYST | 3000.00 | | 20 | researchmgr

7876 | ADAMS | CLERK | 1100.00 | | 20 | researchmgr

7902 | FORD | ANALYST | 3000.00 | | 20 | researchmgr

7499 | ALLEN | SALESMAN | 1600.00 | 300.00 | 30 | salesmgr

7521 | WARD | SALESMAN | 1250.00 | 500.00 | 30 | salesmgr

7654 | MARTIN | SALESMAN | 1250.00 | 1400.00 | 30 | salesmgr

7698 | BLAKE | MANAGER | 2850.00 | | 30 | salesmgr

7844 | TURNER | SALESMAN | 1500.00 | 0.00 | 30 | salesmgr

7900 | JAMES | CLERK | 950.00 | | 30 | salesmgr

(14 rows)

CREATE ROLE salesmgr WITH LOGIN PASSWORD 'password';

GRANT ALL ON vpemp TO salesmgr;

### ADD\_POLICY

The ADD\_POLICY procedure creates a new policy by associating a policy function with a database object.

You must be a superuser to execute this procedure.

ADD\_POLICY(object\_schema VARCHAR2, object\_name VARCHAR2,

policy\_name VARCHAR2, function\_schema VARCHAR2,

policy\_function VARCHAR2

[, statement\_types VARCHAR2

[, update\_check BOOLEAN

[, enable BOOLEAN

[, static\_policy BOOLEAN

[, policy\_type INTEGER

[, long\_predicate BOOLEAN

[, sec\_relevant\_cols VARCHAR2

[, sec\_relevant\_cols\_opt INTEGER ]]]]]]]])

Parameters

object\_schema

Name of the schema containing the database object to which the policy is to be applied.

object\_name

Name of the database object to which the policy is to be applied. A given database object may have more than one policy applied to it.

policy\_name

Name assigned to the policy. The combination of database object (identified by object\_schema and object\_name) and policy name must be unique within the database.

function\_schema

Name of the schema containing the policy function.

Note: The policy function may belong to a package in which case function\_schema must contain the name of the schema in which the package is defined.

policy\_function

Name of the SPL function that defines the rules of the security policy. The same function may be specified in more than one policy.

Note: The policy function may belong to a package in which case policy\_function must also contain the package name in dot notation (that is, package\_name.function\_name).

statement\_types

Comma-separated list of SQL commands to which the policy applies. Valid SQL commands are INSERT, UPDATE, DELETE, and SELECT. The default is INSERT,UPDATE,DELETE,SELECT.

Note: Advanced Server accepts INDEX as a statement type, but it is ignored. Policies are not applied to index operations in Advanced Server.

update\_check

Applies to INSERT and UPDATE SQL commands only.

When set to TRUE, the policy is applied to newly inserted rows and to the modified image of updated rows. If any of the new or modified rows do not qualify according to the policy function predicate, then the INSERT or UPDATE command throws an exception and no rows are inserted or modified by the INSERT or UPDATE command.

When set to FALSE, the policy is not applied to newly inserted rows or the modified image of updated rows. Thus, a newly inserted row may not appear in the result set of a subsequent SQL command that invokes the same policy. Similarly, rows which qualified according to the policy prior to an UPDATE command may not appear in the result set of a subsequent SQL command that invokes the same policy.

The default is FALSE.

enable

When set to TRUE, the policy is enabled and applied to the SQL commands given by the statement\_types parameter. When set to FALSE the policy is disabled and not applied to any SQL commands. The policy can be enabled using the ENABLE\_POLICY procedure. The default is TRUE.

static\_policy

The intended purpose of this parameter is when set to TRUE, the policy is static, which means the policy function is evaluated once per database object the first time it is invoked by a policy on that database object. The resulting policy function predicate string is saved in memory and reused for all invocations of that policy on that database object while the database server instance is running.

When set to FALSE, the policy is dynamic, which means the policy function is re-evaluated and the policy function predicate string regenerated for all invocations of the policy.

The default is FALSE.

Note: The setting of static\_policy is ignored by Advanced Server. Advanced Server implements only the dynamic policy, regardless of the setting of the static\_policy parameter.

policy\_type

Its intended purpose is to determine when the policy function is re-evaluated, and hence, if and when the predicate string returned by the policy function changes. The default is NULL.

Note: The setting of this parameter is ignored by Advanced Server. Advanced Server always assumes a dynamic policy.

long\_predicate

Its intended purpose is to allow predicates up to 32K bytes if set to TRUE, otherwise predicates are limited to 4000 bytes. The default is FALSE.

Note: The setting of this parameter is ignored by Advanced Server. An Advanced Server policy function can return a predicate of unlimited length for all practical purposes.

sec\_relevant\_cols

Comma-separated list of columns of object\_name. Provides column-level Virtual Private Database for the listed columns. The policy is enforced if any of the listed columns are referenced in a SQL command of a type listed in statement\_types. The policy is not enforced if no such columns are referenced.

The default is NULL, which has the same effect as if all of the database object’s columns were included in sec\_relevant\_cols.

sec\_relevant\_cols\_opt

Its intended purpose is when sec\_relevant\_cols\_opt is set to DBMS\_RLS.ALL\_ROWS (INTEGER constant of value 1), then the columns listed in sec\_relevant\_cols return NULL on all rows where the applied policy predicate is false. (If sec\_relevant\_cols\_opt is not set to DBMS\_RLS.ALL\_ROWS, these rows would not be returned at all in the result set.) The default is NULL.

Note: Advanced Server does not support the DBMS\_RLS.ALL\_ROWS functionality. Advanced Server throws an error if sec\_relevant\_cols\_opt is set to DBMS\_RLS.ALL\_ROWS (INTEGER value of 1).

Examples

This example uses the following policy function:

CREATE OR REPLACE FUNCTION verify\_session\_user (

p\_schema VARCHAR2,

p\_object VARCHAR2

)

RETURN VARCHAR2

IS

BEGIN

RETURN 'authid = SYS\_CONTEXT(''USERENV'', ''SESSION\_USER'')';

END;

This function generates the predicate authid = SYS\_CONTEXT('USERENV', 'SESSION\_USER'), which is added to the WHERE clause of any SQL command of the type specified in the ADD\_POLICY procedure.

This limits the effect of the SQL command to those rows where the content of the authid column is the same as the session user.

Note: This example uses the SYS\_CONTEXT function to return the login user name. The first parameter of the SYS\_CONTEXT function is the name of an application context while the second parameter is the name of an attribute set within the application context. USERENV is a special built-in namespace that describes the current session. Advanced Server does not support application contexts, but only this specific usage of the SYS\_CONTEXT function.

The following anonymous block calls the ADD\_POLICY procedure to create a policy named secure\_update to be applied to the vpemp table using function verify\_session\_user whenever an INSERT, UPDATE, or DELETE SQL command is given referencing the vpemp table.

DECLARE

v\_object\_schema VARCHAR2(30) := 'public';

v\_object\_name VARCHAR2(30) := 'vpemp';

v\_policy\_name VARCHAR2(30) := 'secure\_update';

v\_function\_schema VARCHAR2(30) := 'enterprisedb';

v\_policy\_function VARCHAR2(30) := 'verify\_session\_user';

v\_statement\_types VARCHAR2(30) := 'INSERT,UPDATE,DELETE';

v\_update\_check BOOLEAN := TRUE;

v\_enable BOOLEAN := TRUE;

BEGIN

DBMS\_RLS.ADD\_POLICY(

v\_object\_schema,

v\_object\_name,

v\_policy\_name,

v\_function\_schema,

v\_policy\_function,

v\_statement\_types,

v\_update\_check,

v\_enable

);

END;

After successful creation of the policy, a terminal session is started by user salesmgr. The following query shows the content of the vpemp table:

edb=# \c edb salesmgr

Password for user salesmgr:

You are now connected to database "edb" as user "salesmgr".

edb=> SELECT \* FROM vpemp;

empno | ename | job | sal | comm | deptno | authid

-------+--------+-----------+---------+---------+--------+-------------

7782 | CLARK | MANAGER | 2450.00 | | 10 |

7839 | KING | PRESIDENT | 5000.00 | | 10 |

7934 | MILLER | CLERK | 1300.00 | | 10 |

7369 | SMITH | CLERK | 800.00 | | 20 | researchmgr

7566 | JONES | MANAGER | 2975.00 | | 20 | researchmgr

7788 | SCOTT | ANALYST | 3000.00 | | 20 | researchmgr

7876 | ADAMS | CLERK | 1100.00 | | 20 | researchmgr

7902 | FORD | ANALYST | 3000.00 | | 20 | researchmgr

7499 | ALLEN | SALESMAN | 1600.00 | 300.00 | 30 | salesmgr

7521 | WARD | SALESMAN | 1250.00 | 500.00 | 30 | salesmgr

7654 | MARTIN | SALESMAN | 1250.00 | 1400.00 | 30 | salesmgr

7698 | BLAKE | MANAGER | 2850.00 | | 30 | salesmgr

7844 | TURNER | SALESMAN | 1500.00 | 0.00 | 30 | salesmgr

7900 | JAMES | CLERK | 950.00 | | 30 | salesmgr

(14 rows)

An unqualified UPDATE command (no WHERE clause) is issued by the salesmgr user:

edb=> UPDATE vpemp SET comm = sal \* .75;

UPDATE 6

Instead of updating all rows in the table, the policy restricts the effect of the update to only those rows where the authid column contains the value salesmgr as specified by the policy function predicate authid = SYS\_CONTEXT('USERENV', 'SESSION\_USER').

The following query shows that the comm column has been changed only for those rows where authid contains salesmgr. All other rows are unchanged.

edb=> SELECT \* FROM vpemp;

empno | ename | job | sal | comm | deptno | authid

-------+--------+-----------+---------+---------+--------+-------------

7782 | CLARK | MANAGER | 2450.00 | | 10 |

7839 | KING | PRESIDENT | 5000.00 | | 10 |

7934 | MILLER | CLERK | 1300.00 | | 10 |

7369 | SMITH | CLERK | 800.00 | | 20 | researchmgr

7566 | JONES | MANAGER | 2975.00 | | 20 | researchmgr

7788 | SCOTT | ANALYST | 3000.00 | | 20 | researchmgr

7876 | ADAMS | CLERK | 1100.00 | | 20 | researchmgr

7902 | FORD | ANALYST | 3000.00 | | 20 | researchmgr

7499 | ALLEN | SALESMAN | 1600.00 | 1200.00 | 30 | salesmgr

7521 | WARD | SALESMAN | 1250.00 | 937.50 | 30 | salesmgr

7654 | MARTIN | SALESMAN | 1250.00 | 937.50 | 30 | salesmgr

7698 | BLAKE | MANAGER | 2850.00 | 2137.50 | 30 | salesmgr

7844 | TURNER | SALESMAN | 1500.00 | 1125.00 | 30 | salesmgr

7900 | JAMES | CLERK | 950.00 | 712.50 | 30 | salesmgr

(14 rows)

Furthermore, since the update\_check parameter was set to TRUE in the ADD\_POLICY procedure, the following INSERT command throws an exception since the value given for the authid column, researchmgr, does not match the session user, which is salesmgr, and hence, fails the policy.

edb=> INSERT INTO vpemp VALUES (9001,'SMITH','ANALYST',3200.00,NULL,20, 'researchmgr');

ERROR: policy with check option violation

DETAIL: Policy predicate was evaluated to FALSE with the updated values

If update\_check was set to FALSE, the preceding INSERT command would have succeeded.

The following example illustrates the use of the sec\_relevant\_cols parameter to apply a policy only when certain columns are referenced in the SQL command. The following policy function is used for this example, which selects rows where the employee salary is less than 2000.

CREATE OR REPLACE FUNCTION sal\_lt\_2000 (

p\_schema VARCHAR2,

p\_object VARCHAR2

)

RETURN VARCHAR2

IS

BEGIN

RETURN 'sal < 2000';

END;

The policy is created so that it is enforced only if a SELECT command includes columns sal or comm:

DECLARE

v\_object\_schema VARCHAR2(30) := 'public';

v\_object\_name VARCHAR2(30) := 'vpemp';

v\_policy\_name VARCHAR2(30) := 'secure\_salary';

v\_function\_schema VARCHAR2(30) := 'enterprisedb';

v\_policy\_function VARCHAR2(30) := 'sal\_lt\_2000';

v\_statement\_types VARCHAR2(30) := 'SELECT';

v\_sec\_relevant\_cols VARCHAR2(30) := 'sal,comm';

BEGIN

DBMS\_RLS.ADD\_POLICY(

v\_object\_schema,

v\_object\_name,

v\_policy\_name,

v\_function\_schema,

v\_policy\_function,

v\_statement\_types,

sec\_relevant\_cols => v\_sec\_relevant\_cols

);

END;

If a query does not reference columns sal or comm, then the policy is not applied. The following query returns all 14 rows of table vpemp:

edb=# SELECT empno, ename, job, deptno, authid FROM vpemp;

empno | ename | job | deptno | authid

-------+--------+-----------+--------+-------------

7782 | CLARK | MANAGER | 10 |

7839 | KING | PRESIDENT | 10 |

7934 | MILLER | CLERK | 10 |

7369 | SMITH | CLERK | 20 | researchmgr

7566 | JONES | MANAGER | 20 | researchmgr

7788 | SCOTT | ANALYST | 20 | researchmgr

7876 | ADAMS | CLERK | 20 | researchmgr

7902 | FORD | ANALYST | 20 | researchmgr

7499 | ALLEN | SALESMAN | 30 | salesmgr

7521 | WARD | SALESMAN | 30 | salesmgr

7654 | MARTIN | SALESMAN | 30 | salesmgr

7698 | BLAKE | MANAGER | 30 | salesmgr

7844 | TURNER | SALESMAN | 30 | salesmgr

7900 | JAMES | CLERK | 30 | salesmgr

(14 rows)

If the query references the sal or comm columns, then the policy is applied to the query eliminating any rows where sal is greater than or equal to 2000 as shown by the following:

edb=# SELECT empno, ename, job, sal, comm, deptno, authid FROM vpemp;

empno | ename | job | sal | comm | deptno | authid

-------+--------+----------+---------+---------+--------+-------------

7934 | MILLER | CLERK | 1300.00 | | 10 |

7369 | SMITH | CLERK | 800.00 | | 20 | researchmgr

7876 | ADAMS | CLERK | 1100.00 | | 20 | researchmgr

7499 | ALLEN | SALESMAN | 1600.00 | 1200.00 | 30 | salesmgr

7521 | WARD | SALESMAN | 1250.00 | 937.50 | 30 | salesmgr

7654 | MARTIN | SALESMAN | 1250.00 | 937.50 | 30 | salesmgr

7844 | TURNER | SALESMAN | 1500.00 | 1125.00 | 30 | salesmgr

7900 | JAMES | CLERK | 950.00 | 712.50 | 30 | salesmgr

(8 rows)

### DROP\_POLICY

The DROP\_POLICY procedure deletes an existing policy. The policy function and database object associated with the policy are not deleted by the DROP\_POLICY procedure.

You must be a superuser to execute this procedure.

DROP\_POLICY(object\_schema VARCHAR2, object\_name VARCHAR2,

policy\_name VARCHAR2)

Parameters

object\_schema

Name of the schema containing the database object to which the policy applies.

object\_name

Name of the database object to which the policy applies.

policy\_name

Name of the policy to be deleted.

Examples

The following example deletes policy secure\_update on table public.vpemp:

DECLARE

v\_object\_schema VARCHAR2(30) := 'public';

v\_object\_name VARCHAR2(30) := 'vpemp';

v\_policy\_name VARCHAR2(30) := 'secure\_update';

BEGIN

DBMS\_RLS.DROP\_POLICY(

v\_object\_schema,

v\_object\_name,

v\_policy\_name

);

END;

### ENABLE\_POLICY

The ENABLE\_POLICY procedure enables or disables an existing policy on the specified database object.

You must be a superuser to execute this procedure.

ENABLE\_POLICY(object\_schema VARCHAR2, object\_name VARCHAR2,

policy\_name VARCHAR2, enable BOOLEAN)

Parameters

object\_schema

Name of the schema containing the database object to which the policy applies.

object\_name

Name of the database object to which the policy applies.

policy\_name

Name of the policy to be enabled or disabled.

enable

When set to TRUE, the policy is enabled. When set to FALSE, the policy is disabled.

Examples

The following example disables policy secure\_update on table public.vpemp:

DECLARE

v\_object\_schema VARCHAR2(30) := 'public';

v\_object\_name VARCHAR2(30) := 'vpemp';

v\_policy\_name VARCHAR2(30) := 'secure\_update';

v\_enable BOOLEAN := FALSE;

BEGIN

DBMS\_RLS.ENABLE\_POLICY(

v\_object\_schema,

v\_object\_name,

v\_policy\_name,

v\_enable

);

END;

## DBMS\_SCHEDULER

The DBMS\_SCHEDULER package provides a way to create and manage jobs, programs and job schedules.

Table 7.7.2 DBMS\_SCHEDULER Functions and Procedures

| Function/Procedure | Return Type | Description |
| --- | --- | --- |
| CREATE\_JOB(job\_name, job\_type, job\_action, number\_of\_arguments, start\_date, repeat\_interval, end\_date, job\_class, enabled, auto\_drop, comments) | n/a | Use the first form of the CREATE\_JOB procedure to create a job, specifying program and schedule details by means of parameters. |
| CREATE\_JOB(job\_name, program\_name, schedule\_name, job\_class, enabled, auto\_drop, comments) | n/a | Use the second form of CREATE\_JOB to create a job that uses a named program and named schedule. |
| CREATE\_PROGRAM(program\_name, program\_type, program\_action, number\_of\_arguments, enabled, comments) | n/a | Use CREATE\_PROGRAM to create a program. |
| CREATE\_SCHEDULE( schedule\_name, start\_date, repeat\_interval, end\_date, comments) | n/a | Use the CREATE\_SCHEDULE procedure to create a schedule. |
| DEFINE\_PROGRAM\_ARGUMENT( program\_name, argument\_position, argument\_name, argument\_type, default\_value, out\_argument) | n/a | Use the first form of the DEFINE\_PROGRAM\_ARGUMENT procedure to define a program argument that has a default value. |
| DEFINE\_PROGRAM\_ARGUMENT( program\_name, argument\_position, argument\_name, argument\_type, out\_argument) | n/a | Use the first form of the DEFINE\_PROGRAM\_ARGUMENT procedure to define a program argument that does not have a default value. |
| DISABLE(name, force, commit\_semantics) | n/a | Use the DISABLE procedure to disable a job or program. |
| DROP\_JOB(job\_name, force, defer, commit\_semantics) | n/a | Use the DROP\_JOB procedure to drop a job. |
| DROP\_PROGRAM(program\_name, force) | n/a | Use the DROP\_PROGRAM procedure to drop a program. |
| DROP\_PROGRAM\_ARGUMENT( program\_name, argument\_position) | n/a | Use the first form of DROP\_PROGRAM\_ARGUMENT to drop a program argument by specifying the argument position. |
| DROP\_PROGRAM\_ARGUMENT( program\_name, argument\_name) | n/a | Use the second form of DROP\_PROGRAM\_ARGUMENT to drop a program argument by specifying the argument name. |
| DROP\_SCHEDULE(schedule\_name, force) | n/a | Use the DROP SCHEDULE procedure to drop a schedule. |
| ENABLE(name, commit\_semantics) | n/a | Use the ENABLE command to enable a program or job. |
| EVALUATE\_CALENDAR\_STRING( calendar\_string, start\_date, return\_date\_after, next\_run\_date) | n/a | Use EVALUATE\_CALENDAR\_STRING to review the execution date described by a user-defined calendar schedule. |
| RUN\_JOB(job\_name, use\_current\_session, manually) | n/a | Use the RUN\_JOB procedure to execute a job immediately. |
| SET\_JOB\_ARGUMENT\_VALUE( job\_name, argument\_position, argument\_value) | n/a | Use the first form of SET\_JOB\_ARGUMENT value to set the value of a job argument described by the argument's position. |
| SET\_JOB\_ARGUMENT\_VALUE( job\_name, argument\_name, argument\_value) | n/a | Use the second form of SET\_JOB\_ARGUMENT value to set the value of a job argument described by the argument's name. |

The DBMS\_SCHEDULER package is dependent on the pgAgent service; you must have a pgAgent service installed and running on your server before using DBMS\_SCHEDULER.

Before using DBMS\_SCHEDULER, a database superuser must create the catalog tables in which the DBMS\_SCHEDULER programs, schedules and jobs are stored. Use the psql client to connect to the database, and invoke the command:

CREATE EXTENSION dbms\_scheduler;

By default, the dbms\_scheduler extension resides in the contrib/dbms\_scheduler\_ext subdirectory (under the Advanced Server installation).

Note that after creating the DBMS\_SCHEDULER tables, only a superuser will be able to perform a dump or reload of the database.

### Using Calendar Syntax to Specify a Repeating Interval

The CREATE\_JOB and CREATE\_SCHEDULE procedures use a calendar syntax to define the interval with which a job or schedule is repeated. You should provide the scheduling information in the repeat\_interval parameter of each procedure.

repeat\_interval is a value (or series of values) that define the interval between the executions of the scheduled job. Each value is composed of a token, followed by an equal sign, followed by the unit (or units) on which the schedule will execute. Multiple token values must be separated by a semi-colon (;).

For example, the following value:

FREQ=DAILY;BYDAY=MON,TUE,WED,THU,FRI;BYHOUR=17;BYMINUTE=45

Defines a schedule that is executed each weeknight at 5:45.

The token types and syntax described in the table below are supported by Advanced Server:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Token type | Syntax | Valid Values | | |
| FREQ | FREQ=predefined\_interval | Where predefined\_interval is one of the following: YEARLY, MONTHLY, WEEKLY, DAILY, HOURLY, MINUTELY. The SECONDLY keyword is not supported. | | |
| BYMONTH | BYMONTH=month(, month)... | Where month is the three-letter abbreviation of the month name: JAN | FEB | MAR | APR | MAY | JUN | JUL | AUG | SEP | OCT | NOV | DEC | | |
| BYMONTH | BYMONTH=month(, month)... | Where month is the numeric value representing the month: 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | | |
| BYMONTHDAY | BYMONTHDAY=day\_of\_month | Where day\_of\_month is a value from 1 through 31 | | |
| BYDAY | BYDAY=weekday | Where weekday is a three-letter abbreviation or single-digit value representing the day of the week. | | |
| Monday | MON | 1 |
| Tuesday | TUE | 2 |
| Wednesday | WED | 3 |
| Thursday | THU | 4 |
| Friday | FRI | 5 |
| Saturday | SAT | 6 |
| Sunday | SUN | 7 |
| BYDATE | BYDATE=date(, date)... | Where date is YYYYMMDD.  YYYY is a four-digit year representation of the year, MM is a two-digit representation of the month, and DD is a two-digit day representation of the day. | | |
| BYDATE | BYDATE=date(, date)... | Where date is MMDD.  MM is a two-digit representation of the month, and DD is a two-digit day representation of the day | | |
| BYHOUR | BYHOUR=hour | Where hour is a value from 0 through 23. | | |
| BYMINUTE | BYMINUTE=minute | Where minute is a value from 0 through 59. | | |

### CREATE\_JOB

Use the CREATE\_JOB procedure to create a job. The procedure comes in two forms; the first form of the procedure specifies a schedule within the job definition, as well as a job action that will be invoked when the job executes:

CREATE\_JOB(  
job\_name IN VARCHAR2,   
job\_type IN VARCHAR2,   
job\_action IN VARCHAR2,   
 number\_of\_arguments IN PLS\_INTEGER DEFAULT 0,   
 start\_date IN TIMESTAMP WITH TIME ZONE DEFAULT NULL,  
 repeat\_interval IN VARCHAR2 DEFAULT NULL,   
 end\_date IN TIMESTAMP WITH TIME ZONE DEFAULT NULL,  
 job\_class IN VARCHAR2 DEFAULT 'DEFAULT\_JOB\_CLASS',  
 enabled IN BOOLEAN DEFAULT FALSE,  
 auto\_drop IN BOOLEAN DEFAULT TRUE,  
 commentsIN VARCHAR2 DEFAULT NULL)

The second form uses a job schedule to specify the schedule on which the job will execute, and specifies the name of a program that will execute when the job runs:

CREATE\_JOB(  
 job\_name IN VARCHAR2,  
 program\_name IN VARCHAR2,  
 schedule\_name IN VARCHAR2,  
 job\_class IN VARCHAR2 DEFAULT 'DEFAULT\_JOB\_CLASS',  
 enabled IN BOOLEAN DEFAULT FALSE,  
 auto\_dropIN BOOLEAN DEFAULT TRUE,  
 comments IN VARCHAR2 DEFAULT NULL)

Parameters

job\_name

job\_name specifies the optionally schema-qualified name of the job being created.

job\_type

job\_type specifies the type of job. The current implementation of CREATE\_JOB supports a job type of PLSQL\_BLOCK or STORED\_PROCEDURE.

job\_action

If job\_type is PLSQL\_BLOCK, job\_action specifies the content of the PL/SQL block that will be invoked when the job executes. The block must be terminated with a semi-colon (;).

If job\_type is STORED\_PROCEDURE, job\_action specifies the optionally schema-qualified name of the procedure.

number\_of\_arguments

number\_of\_arguments is an INTEGER value that specifies the number of arguments expected by the job. The default is 0.

start\_date

start\_date is a TIMESTAMP WITH TIME ZONE value that specifies the first time that the job is scheduled to execute. The default value is NULL, indicating that the job should be scheduled to execute when the job is enabled.

repeat\_interval

repeat\_interval is a VARCHAR2 value that specifies how often the job will repeat. If a repeat\_interval is not specified, the job will execute only once. The default value is NULL.

For information about defining a repeating schedule for a job, see Section 9.12.1.

end\_date

end\_dateis a TIMESTAMP WITH TIME ZONE value that specifies a time after which the job will no longer execute. If a date is specified, the end\_date must be after start\_date. The default value is NULL.

Please note that if an end\_date is not specified and a repeat\_interval is specified, the job will repeat indefinitely until it is disabled.

program\_name

program\_name is the name of a program that will be executed by the job.

schedule\_name

schedule\_name is the name of the schedule associated with the job.

job\_class

job\_class is accepted for compatibility and ignored.

enabled

enabled is a BOOLEAN value that specifies if the job is enabled when created. By default, a job is created in a disabled state, with enabled set to FALSE. To enable a job, specify a value of TRUE when creating the job, or enable the job with the DBMS\_SCHEDULER.ENABLE procedure.

auto\_drop

The auto\_drop parameter is accepted for compatibility and is ignored. By default, a job's status will be changed to DISABLED after the time specified in end\_date.

comments

Use the comments parameter to specify a comment about the job.

Example

The following example demonstrates a call to the CREATE\_JOB procedure:

EXEC

DBMS\_SCHEDULER.CREATE\_JOB (

job\_name => 'update\_log',

job\_type => 'PLSQL\_BLOCK',

job\_action => 'BEGIN INSERT INTO my\_log VALUES(current\_timestamp);  
 END;',

start\_date => '01-JUN-15 09:00:00.000000',

repeat\_interval => 'FREQ=DAILY;BYDAY=MON,TUE,WED,THU,FRI;BYHOUR=17;',

end\_date => NULL,

enabled => TRUE,

comments => 'This job adds a row to the my\_log table.');

The code fragment creates a job named update\_log that executes each weeknight at 5:00. The job executes a PL/SQL block that inserts the current timestamp into a logfile (my\_log). Since no end\_date is specified, the job will execute until it is disabled by the DBMS\_SCHEDULER.DISABLE procedure.

### CREATE\_PROGRAM

Use the CREATE\_PROGRAM procedure to create a DBMS\_SCHEDULER program. The signature is:

CREATE\_PROGRAM(  
 program\_name IN VARCHAR2,  
 program\_type IN VARCHAR2,  
 program\_action IN VARCHAR2,  
 number\_of\_arguments IN PLS\_INTEGER DEFAULT 0,  
 enabled IN BOOLEAN DEFAULT FALSE,  
 comments IN VARCHAR2 DEFAULT NULL)

Parameters

program\_name

program\_name specifies the name of the program that is being created.

program\_type

program\_type specifies the type of program. The current implementation of CREATE\_PROGRAM supports a program\_type of PLSQL\_BLOCK or PROCEDURE.

program\_action

If program\_type is PLSQL\_BLOCK, program\_action contains the PL/SQL block that will execute when the program is invoked. The PL/SQL block must be terminated with a semi-colon (;).

If program\_type is PROCEDURE, program\_action contains the name of the stored procedure.

number\_of\_arguments

If program\_type is PLSQL\_BLOCK, this argument is ignored.

If program\_type is PROCEDURE, number\_of\_argumentsspecifies the number of arguments required by the procedure. The default value is 0.

enabled

enabled specifies if the program is created enabled or disabled:

* If enabled is TRUE, the program is created enabled.
* If enabled is FALSE, the program is created disabled; use the DBMS\_SCHEDULER.ENABLE program to enable a disabled program.

The default value is FALSE.

comments

Use the comments parameter to specify a comment about the program; by default, this parameter is NULL.

Example

The following call to the CREATE\_PROGRAM procedure creates a program named update\_log:

EXEC

DBMS\_SCHEDULER.CREATE\_PROGRAM (  
 program\_name => 'update\_log',

program\_type => 'PLSQL\_BLOCK',

program\_action => 'BEGIN INSERT INTO my\_log VALUES(current\_timestamp);  
 END;',

enabled => TRUE,

comment => 'This program adds a row to the my\_log table.');

update\_log is a PL/SQL block that adds a row containing the current date and time to the my\_log table. The program will be enabled when the CREATE\_PROGRAM procedure executes.

### CREATE\_SCHEDULE

Use the CREATE\_SCHEDULE procedure to create a job schedule. The signature of the CREATE\_SCHEDULE procedure is:

CREATE\_SCHEDULE(  
 schedule\_name IN VARCHAR2,  
 start\_date IN TIMESTAMP WITH TIME ZONE DEFAULT NULL,  
 repeat\_interval IN VARCHAR2,  
 end\_date IN TIMESTAMP WITH TIME ZONE DEFAULT NULL,  
 comments IN VARCHAR2 DEFAULT NULL)

Parameters

schedule\_name

schedule\_name specifies the name of the schedule.

start\_date

start\_date is a TIMESTAMP WITH TIME ZONE value that specifies the date and time that the schedule is eligible to execute. If a start\_date is not specified, the date that the job is enabled is used as the start\_date. By default, start\_date is NULL.

repeat\_interval

repeat\_interval is a VARCHAR2 value that specifies how often the job will repeat. If a repeat\_interval is not specified, the job will execute only once, on the date specified by start\_date.

For information about defining a repeating schedule for a job, see Section 9.12.1.

Please note: you must provide a value for either start\_date or repeat\_interval; if both start\_date and repeat\_interval are NULL, the server will return an error.

end\_date IN TIMESTAMP WITH TIME ZONE DEFAULT NULL

end\_dateis a TIMESTAMP WITH TIME ZONE value that specifies a time after which the schedule will no longer execute. If a date is specified, the *end\_date* must be after the start\_date. The default value is NULL.

Please note that if a repeat\_interval is specified and an end\_date is not specified, the schedule will repeat indefinitely until it is disabled.

comments IN VARCHAR2 DEFAULT NULL)

Use the comments parameter to specify a comment about the schedule; by default, this parameter is NULL.

Example

The following code fragment calls CREATE\_SCHEDULE to create a schedule named weeknights\_at\_5:

EXEC

DBMS\_SCHEDULER.CREATE\_SCHEDULE (

schedule\_name => 'weeknights\_at\_5',

start\_date => '01-JUN-13 09:00:00.000000'

repeat\_interval => 'FREQ=DAILY;BYDAY=MON,TUE,WED,THU,FRI;BYHOUR=17;',

comments => 'This schedule executes each weeknight at 5:00');

The schedule executes each weeknight, at 5:00 pm, effective after June 1, 2013. Since no end\_date is specified, the schedule will execute indefinitely until it is disabled with DBMS\_SCHEDULER.DISABLE.

### DEFINE\_PROGRAM\_ARGUMENT

Use the DEFINE\_PROGRAM\_ARGUMENT procedure to define a program argument. The DEFINE\_PROGRAM\_ARGUMENT procedure comes in two forms; the first form defines an argument with a default value:

DEFINE\_PROGRAM\_ARGUMENT(  
program\_name IN VARCHAR2,  
 argument\_position IN PLS\_INTEGER,  
 argument\_name IN VARCHAR2 DEFAULT NULL,  
 argument\_type IN VARCHAR2,  
 default\_value IN VARCHAR2,  
 out\_argument IN BOOLEAN DEFAULT FALSE)

The second form defines an argument without a default value:

DEFINE\_PROGRAM\_ARGUMENT(  
 program\_name IN VARCHAR2,  
 argument\_position IN PLS\_INTEGER,  
 argument\_name IN VARCHAR2 DEFAULT NULL,  
 argument\_type IN VARCHAR2,  
 out\_argument IN BOOLEAN DEFAULT FALSE)

Parameters

program\_name

program\_name is the name of the program to which the arguments belong.

argument\_position

argument\_position specifies the position of the argument as it is passed to the program.

argument\_name

argument\_name specifies the optional name of the argument. By default, argument\_name is NULL.

argument\_type IN VARCHAR2

argument\_type specifies the data type of the argument.

default\_value

default\_value specifies the default value assigned to the argument. default\_valuewill be overridden by a value specified by the job when the job executes.

out\_argument IN BOOLEAN DEFAULT FALSE

out\_argument is not currently used; if specified, the value must be FALSE.

Example

The following code fragment uses the DEFINE\_PROGRAM\_ARGUMENT procedure to define the first and second arguments in a program named add\_emp:

EXEC

DBMS\_SCHEDULER.DEFINE\_PROGRAM\_ARGUMENT(

program\_name => 'add\_emp',

argument\_position => 1,

argument\_name => 'dept\_no',

argument\_type => 'INTEGER,

default\_value => '20');

EXEC

DBMS\_SCHEDULER.DEFINE\_PROGRAM\_ARGUMENT(

program\_name => 'add\_emp',

argument\_position => 2,

argument\_name => 'emp\_name',

argument\_type => 'VARCHAR2');

The first argument is an INTEGER value named dept\_no that has a default value of 20. The second argument is a VARCHAR2 value named emp\_name; the second argument does not have a default value.

### DISABLE

Use the DISABLE procedure to disable a program or a job. The signature of the DISABLE procedure is:

DISABLE(  
 name IN VARCHAR2,  
 force IN BOOLEAN DEFAULT FALSE,  
 commit\_semantics IN VARCHAR2 DEFAULT 'STOP\_ON\_FIRST\_ERROR')

Parameters

name

name specifies the name of the program or job that is being disabled.

force

force is accepted for compatibility, and ignored.

commit\_semantics

commit\_semantics instructs the server how to handle an error encountered while disabling a program or job. By default, commit\_semantics is set to STOP\_ON\_FIRST\_ERROR, instructing the server to stop when it encounters an error. Any programs or jobs that were successfully disabled prior to the error will be committed to disk.

The TRANSACTIONAL and ABSORB\_ERRORS keywords are accepted for compatibility, and ignored.

Example

The following call to the DISABLE procedure disables a program named update\_emp:

DBMS\_SCHEDULER.DISABLE('update\_emp');

### DROP\_JOB

Use the DROP\_JOB procedure to DROP a job, DROP any arguments that belong to the job, and eliminate any future job executions. The signature of the procedure is:

DROP\_JOB(  
 job\_name IN VARCHAR2,  
 force IN BOOLEAN DEFAULT FALSE,  
 defer IN BOOLEAN DEFAULT FALSE,  
 commit\_semantics IN VARCHAR2 DEFAULT 'STOP\_ON\_FIRST\_ERROR')

Parameters

job\_name

job\_name specifies the name of the job that is being dropped.

force

force is accepted for compatibility, and ignored.

defer

defer is accepted for compatibility, and ignored.

commit\_semantics

commit\_semantics instructs the server how to handle an error encountered while dropping a program or job. By default, commit\_semantics is set to STOP\_ON\_FIRST\_ERROR, instructing the server to stop when it encounters an error.

The TRANSACTIONAL and ABSORB\_ERRORS keywords are accepted for compatibility, and ignored.

Example

The following call to DROP\_JOB drops a job named update\_log:

DBMS\_SCHEDULER.DROP\_JOB('update\_log');

### DROP\_PROGRAM

The DROP\_PROGRAM procedure

The signature of the DROP\_PROGRAM procedure is:

DROP\_PROGRAM(  
 program\_name IN VARCHAR2,  
 force IN BOOLEAN DEFAULT FALSE)

Parameters

program\_name

program\_name specifies the name of the program that is being dropped.

force

force is a BOOLEAN value that instructs the server how to handle programs with dependent jobs.

Specify FALSE to instruct the server to return an error if the program is referenced by a job.

Specify TRUE to instruct the server to disable any jobs that reference the program before dropping the program.

The default value is FALSE.

Example

The following call to DROP\_PROGRAM drops a job named update\_emp:

DBMS\_SCHEDULER.DROP\_PROGRAM('update\_emp');

### DROP\_PROGRAM\_ARGUMENT

Use the DROP\_PROGRAM\_ARGUMENT procedure to drop a program argument. The DROP\_PROGRAM\_ARGUMENT procedure comes in two forms; the first form uses an argument position to specify which argument to drop:

DROP\_PROGRAM\_ARGUMENT(  
 program\_name IN VARCHAR2,  
 argument\_position IN PLS\_INTEGER)

The second form takes the argument name:

DROP\_PROGRAM\_ARGUMENT(  
 program\_name IN VARCHAR2,  
 argument\_name IN VARCHAR2)

Parameters

program\_name

program\_name specifies the name of the program that is being modified.

argument\_position

argument\_position specifies the position of the argument that is being dropped.

argument\_name

argument\_name specifies the name of the argument that is being dropped.

Examples

The following call to DROP\_PROGRAM\_ARGUMENT drops the first argument in the update\_emp program:

DBMS\_SCHEDULER.DROP\_PROGRAM\_ARGUMENT('update\_emp', 1);

The following call to DROP\_PROGRAM\_ARGUMENT drops an argument named emp\_name:

DBMS\_SCHEDULER.DROP\_PROGRAM\_ARGUMENT(update\_emp', 'emp\_name');

### DROP\_SCHEDULE

Use the DROP\_SCHEDULE procedure to drop a schedule. The signature is:

DROP\_SCHEDULE(  
 schedule\_name IN VARCHAR2,  
 force IN BOOLEAN DEFAULT FALSE)

Parameters

schedule\_name

schedule\_name specifies the name of the schedule that is being dropped.

force

force specifies the behavior of the server if the specified schedule is referenced by any job:

* Specify FALSE to instruct the server to return an error if the specified schedule is referenced by a job. This is the default behavior.
* Specify TRUE to instruct the server to disable to any jobs that use the specified schedule before dropping the schedule. Any running jobs will be allowed to complete before the schedule is dropped.

Example

The following call to DROP\_SCHEDULE drops a schedule named weeknights\_at\_5:

DBMS\_SCHEDULER.DROP\_SCHEDULE('weeknights\_at\_5', TRUE);

The server will disable any jobs that use the schedule before dropping the schedule.

### ENABLE

Use the ENABLE procedure to enable a disabled program or job.

The signature of the ENABLE procedure is:

ENABLE(  
 name IN VARCHAR2,  
 commit\_semantics IN VARCHAR2 DEFAULT 'STOP\_ON\_FIRST\_ERROR')

Parameters

name

name specifies the name of the program or job that is being enabled.

commit\_semantics

commit\_semantics instructs the server how to handle an error encountered while enabling a program or job. By default, commit\_semantics is set to STOP\_ON\_FIRST\_ERROR, instructing the server to stop when it encounters an error.

The TRANSACTIONAL and ABSORB\_ERRORS keywords are accepted for compatibility, and ignored.

Example

The following call to DBMS\_SCHEDULER.ENABLE enables the update\_emp program:

DBMS\_SCHEDULER.ENABLE('update\_emp');

### EVALUATE\_CALENDAR\_STRING

Use the EVALUATE\_CALENDAR\_STRING procedure to evaluate the repeat\_interval value specified when creating a schedule with the CREATE\_SCHEDULE procedure. The EVALUATE\_CALENDAR\_STRING procedure will return the date and time that a specified schedule will execute without actually scheduling the job.

The signature of the EVALUATE\_CALENDAR\_STRING procedure is:

EVALUATE\_CALENDAR\_STRING(  
calendar\_string IN VARCHAR2,  
 start\_date IN TIMESTAMP WITH TIME ZONE,  
 return\_date\_after IN TIMESTAMP WITH TIME ZONE,  
 next\_run\_date OUT TIMESTAMP WITH TIME ZONE)

Parameters

calendar\_string

calendar\_string is the calendar string that describes a repeat\_interval (see Section 9.12.1) that is being evaluated.

start\_date IN TIMESTAMP WITH TIME ZONE

start\_date is the date and time after which the repeat\_interval will become valid.

return\_date\_after

Use the return\_date\_after parameter to specify the date and time that EVALUATE\_CALENDAR\_STRING should use as a starting date when evaluating the repeat\_interval.

For example, if you specify a return\_date\_after value of 01-APR-13 09.00.00.000000, EVALUATE\_CALENDAR\_STRING will return the date and time of the first iteration of the schedule after April 1st, 2013.

next\_run\_date OUT TIMESTAMP WITH TIME ZONE

next\_run\_date is an OUT parameter that will contain the first occurrence of the schedule after the date specified by the return\_date\_after parameter.

Example

The following example evaluates a calendar string and returns the first date and time that the schedule will be executed after June 15, 2013:

DECLARE

result TIMESTAMP;

BEGIN

DBMS\_SCHEDULER.EVALUATE\_CALENDAR\_STRING

(

'FREQ=DAILY;BYDAY=MON,TUE,WED,THU,FRI;BYHOUR=17;',

'15-JUN-2013', NULL, result

);

DBMS\_OUTPUT.PUT\_LINE('next\_run\_date: ' || result);

END;

/

next\_run\_date: 17-JUN-13 05.00.00.000000 PM

June 15, 2013 is a Saturday; the schedule will not execute until Monday, June 17, 2013 at 5:00 pm.

### RUN\_JOB

Use the RUN\_JOB procedure to execute a job immediately. The signature of the RUN\_JOB procedure is:

RUN\_JOB(  
 job\_name IN VARCHAR2,  
 use\_current\_session IN BOOLEAN DEFAULT TRUE

Parameters

job\_name

job\_name specifies the name of the job that will execute.

use\_current\_session

By default, the job will execute in the current session. If specified, use\_current\_session must be set to TRUE ; if use\_current\_session is set to FALSE, Advanced Server will return an error.

Example

The following call to RUN\_JOB executes a job named update\_log:

DBMS\_SCHEDULER.RUN\_JOB('update\_log', TRUE);

Passing a value of TRUE as the second argument instructs the server to invoke the job in the current session.

### SET\_JOB\_ARGUMENT\_VALUE

Use the SET\_JOB\_ARGUMENT\_VALUE procedure to specify a value for an argument. The SET\_JOB\_ARGUMENT\_VALUE procedure comes in two forms; the first form specifies which argument should be modified by position:

SET\_JOB\_ARGUMENT\_VALUE(  
 job\_name IN VARCHAR2,  
 argument\_position IN PLS\_INTEGER,  
 argument\_value IN VARCHAR2)

The second form uses an argument name to specify which argument to modify:

SET\_JOB\_ARGUMENT\_VALUE(  
 job\_name IN VARCHAR2,  
 argument\_name IN VARCHAR2,  
 argument\_value IN VARCHAR2)

Argument values set by the SET\_JOB\_ARGUMENT\_VALUE procedure override any values set by default.

Parameters

job\_name

job\_name specifies the name of the job to which the modified argument belongs.

argument\_position

Use argument\_position to specify the argument position for which the value will be set.

argument\_name

Use argument\_name to specify the argument by name for which the value will be set.

argument\_value

argument\_value specifies the new value of the argument.

Examples

The following example assigns a value of 30 to the first argument in the update\_emp job:

DBMS\_SCHEDULER.SET\_JOB\_ARGUMENT\_VALUE('update\_emp', 1, '30');

The following example sets the emp\_name argument to SMITH:

DBMS\_SCHEDULER.SET\_JOB\_ARGUMENT\_VALUE('update\_emp', 'emp\_name', 'SMITH');

## DBMS\_SESSION

Advanced Server provides support for the DBMS\_SESSION.SET\_ROLE procedure.

Table 7.7.2 DBMS\_SESSION Procedure

| Function/Procedure | Return Type | Description |
| --- | --- | --- |
| SET\_ROLE(role\_cmd) | n/a | Executes a SET ROLE statement followed by the string value specified in role\_cmd. |

Advanced Server's implementation of DBMS\_SESSION is a partial implementation when compared to Oracle's version. Only DBMS\_SESSION.SET\_ROLE is supported.

### SET\_ROLE

The SET\_ROLE procedure sets the current session user to the role specified in *role\_cmd*. After invoking the SET\_ROLE procedure, the current session will use the permissions assigned to the specified role. The signature of the procedure is:

SET\_ROLE(*role*\_*cmd*)

The SET\_ROLE procedure appends the value specified for *role*\_*cmd* to the SET ROLE statement, and then invokes the statement.

**Parameters**

*role\_cmd*

*role\_cmd* specifies a role name in the form of a string value.

**Example**

The following call to the SET\_ROLE procedure invokes the SET ROLE command to set the identity of the current session user to manager:

edb=# exec DBMS\_SESSION.SET\_ROLE('manager');

## DBMS\_SQL

The DBMS\_SQL package provides an application interface to the EnterpriseDB dynamic SQL functionality. With DBMS\_SQL you can construct queries and other commands at run time (rather than when you write the application). EnterpriseDB Advanced Server offers native support for dynamic SQL; DBMS\_SQL provides a way to use dynamic SQL without modifying your application.

DBMS\_SQL assumes the privileges of the current user when executing dynamic SQL statements.

Table ‑ DBMS\_SQL Functions/Procedures

| Function/Procedure | Function or Procedure | Return Type | Description |
| --- | --- | --- | --- |
| BIND\_VARIABLE(c, name, value [, out\_value\_size ]) | Procedure | n/a | Bind a value to a variable. |
| BIND\_VARIABLE\_CHAR(c, name, value [, out\_value\_size ]) | Procedure | n/a | Bind a CHAR value to a variable. |
| BIND\_VARIABLE\_RAW(c, name, value [, out\_value\_size ]) | Procedure | n/a | Bind a RAW value to a variable. |
| CLOSE\_CURSOR(c IN OUT) | Procedure | n/a | Close a cursor. |
| COLUMN\_VALUE(c, position, value OUT [, column\_error OUT [, actual\_length OUT ]]) | Procedure | n/a | Return a column value into a variable. |
| COLUMN\_VALUE\_CHAR(c, position, value OUT [, column\_error OUT [, actual\_length OUT ]]) | Procedure | n/a | Return a CHAR column value into a variable. |
| COLUMN\_VALUE\_RAW(c, position, value OUT [, column\_error OUT [, actual\_length OUT ]]) | Procedure | n/a | Return a RAW column value into a variable. |
| DEFINE\_COLUMN(c, position, column [, column\_size ]) | Procedure | n/a | Define a column in the SELECT list. |
| DEFINE\_COLUMN\_CHAR(c, position, column, column\_size) | Procedure | n/a | Define a CHAR column in the SELECT list. |
| DEFINE\_COLUMN\_RAW(c, position, column, column\_size) | Procedure | n/a | Define a RAW column in the SELECT list. |
| DESCRIBE\_COLUMNS | Procedure | n/a | Defines columns to hold a cursor result set. |
| EXECUTE(c) | Function | INTEGER | Execute a cursor. |
| EXECUTE\_AND\_FETCH(c [, exact ]) | Function | INTEGER | Execute a cursor and fetch a single row. |
| FETCH\_ROWS(c) | Function | INTEGER | Fetch rows from the cursor. |
| IS\_OPEN(c) | Function | BOOLEAN | Check if a cursor is open. |
| LAST\_ROW\_COUNT | Function | INTEGER | Return cumulative number of rows fetched. |
| OPEN\_CURSOR | Function | INTEGER | Open a cursor. |
| PARSE(c, statement, language\_flag) | Procedure | n/a | Parse a statement. |

The following table lists the public variable available in the DBMS\_SQL package.

Table ‑ DBMS\_SQL Public Variables

| Public Variables | Data Type | Value | Description |
| --- | --- | --- | --- |
| native | INTEGER | 1 | See DBMS\_SQL.PARSE for more information. |
| V6 | INTEGER | 2 | See DBMS\_SQL.PARSE for more information. |
| V7 | INTEGER | 3 | See DBMS\_SQL.PARSE for more information |

### BIND\_VARIABLE

The BIND\_VARIABLE procedure provides the capability to associate a value with an IN or IN OUT bind variable in a SQL command.

BIND\_VARIABLE(c INTEGER, name VARCHAR2,

value { BLOB | CLOB | DATE | FLOAT | INTEGER | NUMBER |

TIMESTAMP | VARCHAR2 }

[, out\_value\_size INTEGER ])

Parameters

c

Cursor ID of the cursor for the SQL command with bind variables.

name

Name of the bind variable in the SQL command.

value

Value to be assigned.

out\_value\_size

If name is an IN OUT variable, defines the maximum length of the output value. If not specified, the length of value is assumed.

Examples

The following anonymous block uses bind variables to insert a row into the emp table.

DECLARE

curid INTEGER;

v\_sql VARCHAR2(150) := 'INSERT INTO emp VALUES ' ||

'(:p\_empno, :p\_ename, :p\_job, :p\_mgr, ' ||

':p\_hiredate, :p\_sal, :p\_comm, :p\_deptno)';

v\_empno emp.empno%TYPE;

v\_ename emp.ename%TYPE;

v\_job emp.job%TYPE;

v\_mgr emp.mgr%TYPE;

v\_hiredate emp.hiredate%TYPE;

v\_sal emp.sal%TYPE;

v\_comm emp.comm%TYPE;

v\_deptno emp.deptno%TYPE;

v\_status INTEGER;

BEGIN

curid := DBMS\_SQL.OPEN\_CURSOR;

DBMS\_SQL.PARSE(curid,v\_sql,DBMS\_SQL.native);

v\_empno := 9001;

v\_ename := 'JONES';

v\_job := 'SALESMAN';

v\_mgr := 7369;

v\_hiredate := TO\_DATE('13-DEC-07','DD-MON-YY');

v\_sal := 8500.00;

v\_comm := 1500.00;

v\_deptno := 40;

DBMS\_SQL.BIND\_VARIABLE(curid,':p\_empno',v\_empno);

DBMS\_SQL.BIND\_VARIABLE(curid,':p\_ename',v\_ename);

DBMS\_SQL.BIND\_VARIABLE(curid,':p\_job',v\_job);

DBMS\_SQL.BIND\_VARIABLE(curid,':p\_mgr',v\_mgr);

DBMS\_SQL.BIND\_VARIABLE(curid,':p\_hiredate',v\_hiredate);

DBMS\_SQL.BIND\_VARIABLE(curid,':p\_sal',v\_sal);

DBMS\_SQL.BIND\_VARIABLE(curid,':p\_comm',v\_comm);

DBMS\_SQL.BIND\_VARIABLE(curid,':p\_deptno',v\_deptno);

v\_status := DBMS\_SQL.EXECUTE(curid);

DBMS\_OUTPUT.PUT\_LINE('Number of rows processed: ' || v\_status);

DBMS\_SQL.CLOSE\_CURSOR(curid);

END;

Number of rows processed: 1

### BIND\_VARIABLE\_CHAR

The BIND\_VARIABLE\_CHAR procedure provides the capability to associate a CHAR value with an IN or IN OUT bind variable in a SQL command.

BIND\_VARIABLE\_CHAR(c INTEGER, name VARCHAR2, value CHAR

[, out\_value\_size INTEGER ])

Parameters

c

Cursor ID of the cursor for the SQL command with bind variables.

name

Name of the bind variable in the SQL command.

value

Value of type CHAR to be assigned.

out\_value\_size

If name is an IN OUT variable, defines the maximum length of the output value. If not specified, the length of value is assumed.

### BIND VARIABLE RAW

The BIND\_VARIABLE\_RAW procedure provides the capability to associate a RAW value with an IN or IN OUT bind variable in a SQL command.

BIND\_VARIABLE\_RAW(c INTEGER, name VARCHAR2, value RAW

[, out\_value\_size INTEGER ])

Parameters

c

Cursor ID of the cursor for the SQL command with bind variables.

name

Name of the bind variable in the SQL command.

value

Value of type RAW to be assigned.

out\_value\_size

If name is an IN OUT variable, defines the maximum length of the output value. If not specified, the length of value is assumed.

### CLOSE\_CURSOR

The CLOSE\_CURSOR procedure closes an open cursor. The resources allocated to the cursor are released and it can no longer be used.

CLOSE\_CURSOR(c IN OUT INTEGER)

Parameters

c

Cursor ID of the cursor to be closed.

Examples

The following example closes a previously opened cursor:

DECLARE

curid INTEGER;

BEGIN

curid := DBMS\_SQL.OPEN\_CURSOR;

.

.

.

DBMS\_SQL.CLOSE\_CURSOR(curid);

END;

### COLUMN\_VALUE

The COLUMN\_VALUE procedure defines a variable to receive a value from a cursor.

COLUMN\_VALUE(c INTEGER, position INTEGER, value OUT { BLOB |

CLOB | DATE | FLOAT | INTEGER | NUMBER | TIMESTAMP | VARCHAR2 }

[, column\_error OUT NUMBER [, actual\_length OUT INTEGER ]])

Parameters

c

Cursor id of the cursor returning data to the variable being defined.

position

Position within the cursor of the returned data. The first value in the cursor is position 1.

value

Variable receiving the data returned in the cursor by a prior fetch call.

column\_error

Error number associated with the column, if any.

actual\_length

Actual length of the data prior to any truncation.

Examples

The following example shows the portion of an anonymous block that receives the values from a cursor using the COLUMN\_VALUE procedure.

DECLARE

curid INTEGER;

v\_empno NUMBER(4);

v\_ename VARCHAR2(10);

v\_hiredate DATE;

v\_sal NUMBER(7,2);

v\_comm NUMBER(7,2);

v\_sql VARCHAR2(50) := 'SELECT empno, ename, hiredate, sal, ' ||

'comm FROM emp';

v\_status INTEGER;

BEGIN

.

.

.

LOOP

v\_status := DBMS\_SQL.FETCH\_ROWS(curid);

EXIT WHEN v\_status = 0;

DBMS\_SQL.COLUMN\_VALUE(curid,1,v\_empno);

DBMS\_SQL.COLUMN\_VALUE(curid,2,v\_ename);

DBMS\_SQL.COLUMN\_VALUE(curid,3,v\_hiredate);

DBMS\_SQL.COLUMN\_VALUE(curid,4,v\_sal);

DBMS\_SQL.COLUMN\_VALUE(curid,4,v\_sal);

DBMS\_SQL.COLUMN\_VALUE(curid,5,v\_comm);

DBMS\_OUTPUT.PUT\_LINE(v\_empno || ' ' || RPAD(v\_ename,10) || ' ' ||

TO\_CHAR(v\_hiredate,'yyyy-mm-dd') || ' ' ||

TO\_CHAR(v\_sal,'9,999.99') || ' ' ||

TO\_CHAR(NVL(v\_comm,0),'9,999.99'));

END LOOP;

DBMS\_SQL.CLOSE\_CURSOR(curid);

END;

### COLUMN\_VALUE\_CHAR

The COLUMN\_VALUE\_CHAR procedure defines a variable to receive a CHAR value from a cursor.

COLUMN\_VALUE\_CHAR(c INTEGER, position INTEGER, value OUT CHAR

[, column\_error OUT NUMBER [, actual\_length OUT INTEGER ]])

Parameters

c

Cursor id of the cursor returning data to the variable being defined.

position

Position within the cursor of the returned data. The first value in the cursor is position 1.

value

Variable of data type CHAR receiving the data returned in the cursor by a prior fetch call.

column\_error

Error number associated with the column, if any.

actual\_length

Actual length of the data prior to any truncation.

### COLUMN VALUE RAW

The COLUMN\_VALUE\_RAW procedure defines a variable to receive a RAW value from a cursor.

COLUMN\_VALUE\_RAW(c INTEGER, position INTEGER, value OUT RAW

[, column\_error OUT NUMBER [, actual\_length OUT INTEGER ]])

Parameters

c

Cursor id of the cursor returning data to the variable being defined.

position

Position within the cursor of the returned data. The first value in the cursor is position 1.

value

Variable of data type RAW receiving the data returned in the cursor by a prior fetch call.

column\_error

Error number associated with the column, if any.

actual\_length

Actual length of the data prior to any truncation.

### DEFINE\_COLUMN

The DEFINE\_COLUMN procedure defines a column or expression in the SELECT list that is to be returned and retrieved in a cursor.

DEFINE\_COLUMN(c INTEGER, position INTEGER, column { BLOB |

CLOB | DATE | FLOAT | INTEGER | NUMBER | TIMESTAMP | VARCHAR2 }

[, column\_size INTEGER ])

Parameters

c

Cursor id of the cursor associated with the SELECT command.

position

Position of the column or expression in the SELECT list that is being defined.

column

A variable that is of the same data type as the column or expression in position position of the SELECT list.

column\_size

The maximum length of the returned data. column\_size must be specified only if column is VARCHAR2. Returned data exceeding column\_size is truncated to column\_size characters.

Examples

The following shows how the empno, ename, hiredate, sal, and comm columns of the emp table are defined with the DEFINE\_COLUMN procedure.

DECLARE

curid INTEGER;

v\_empno NUMBER(4);

v\_ename VARCHAR2(10);

v\_hiredate DATE;

v\_sal NUMBER(7,2);

v\_comm NUMBER(7,2);

v\_sql VARCHAR2(50) := 'SELECT empno, ename, hiredate, sal, ' ||

'comm FROM emp';

v\_status INTEGER;

BEGIN

curid := DBMS\_SQL.OPEN\_CURSOR;

DBMS\_SQL.PARSE(curid,v\_sql,DBMS\_SQL.native);

DBMS\_SQL.DEFINE\_COLUMN(curid,1,v\_empno);

DBMS\_SQL.DEFINE\_COLUMN(curid,2,v\_ename,10);

DBMS\_SQL.DEFINE\_COLUMN(curid,3,v\_hiredate);

DBMS\_SQL.DEFINE\_COLUMN(curid,4,v\_sal);

DBMS\_SQL.DEFINE\_COLUMN(curid,5,v\_comm);

.

.

.

END;

The following shows an alternative to the prior example that produces the exact same results. Note that the lengths of the data types are irrelevant – the empno, sal, and comm columns will still return data equivalent to NUMBER(4) and NUMBER(7,2), respectively, even though v\_num is defined as NUMBER(1) (assuming the declarations in the COLUMN\_VALUE procedure are of the appropriate maximum sizes). The ename column will return data up to ten characters in length as defined by the length parameter in the DEFINE\_COLUMN call, not by the data type declaration, VARCHAR2(1) declared for v\_varchar. The actual size of the returned data is dictated by the COLUMN\_VALUE procedure.

DECLARE

curid INTEGER;

v\_num NUMBER(1);

v\_varchar VARCHAR2(1);

v\_date DATE;

v\_sql VARCHAR2(50) := 'SELECT empno, ename, hiredate, sal, ' ||

'comm FROM emp';

v\_status INTEGER;

BEGIN

curid := DBMS\_SQL.OPEN\_CURSOR;

DBMS\_SQL.PARSE(curid,v\_sql,DBMS\_SQL.native);

DBMS\_SQL.DEFINE\_COLUMN(curid,1,v\_num);

DBMS\_SQL.DEFINE\_COLUMN(curid,2,v\_varchar,10);

DBMS\_SQL.DEFINE\_COLUMN(curid,3,v\_date);

DBMS\_SQL.DEFINE\_COLUMN(curid,4,v\_num);

DBMS\_SQL.DEFINE\_COLUMN(curid,5,v\_num);

.

.

.

END;

### DEFINE\_COLUMN\_CHAR

The DEFINE\_COLUMN\_CHAR procedure defines a CHAR column or expression in the SELECT list that is to be returned and retrieved in a cursor.

DEFINE\_COLUMN\_CHAR(c INTEGER, position INTEGER, column CHAR, column\_size INTEGER)

Parameters

c

Cursor id of the cursor associated with the SELECT command.

position

Position of the column or expression in the SELECT list that is being defined.

column

A CHAR variable.

column\_size

The maximum length of the returned data. Returned data exceeding column\_size is truncated to column\_size characters.

### DEFINE COLUMN RAW

The DEFINE\_COLUMN\_RAW procedure defines a RAW column or expression in the SELECT list that is to be returned and retrieved in a cursor.

DEFINE\_COLUMN\_RAW(c INTEGER, position INTEGER, column RAW,

column\_size INTEGER)

Parameters

c

Cursor id of the cursor associated with the SELECT command.

position

Position of the column or expression in the SELECT list that is being defined.

column

A RAW variable.

column\_size

The maximum length of the returned data. Returned data exceeding column\_size is truncated to column\_size characters.

### DESCRIBE COLUMNS

The DESCRIBE\_COLUMNS procedure describes the columns returned by a cursor.

DESCRIBE\_COLUMNS(c INTEGER, col\_cnt OUT INTEGER, desc\_t OUT  
 DESC\_TAB);

Parameters

c

The cursor ID of the cursor.

col\_cnt

The number of columns in cursor result set.

desc\_tab

The table that contains a description of each column returned by the cursor. The descriptions are of type DESC\_REC, and contain the following values:

|  |  |
| --- | --- |
| Column Name | Type |
| col\_type | INTEGER |
| col\_max\_len | INTEGER |
| col\_name | VARCHAR2(128) |
| col\_name\_len | INTEGER |
| col\_schema\_name | VARCHAR2(128) |
| col\_schema\_name\_len | INTEGER |
| col\_precision | INTEGER |
| col\_scale | INTEGER |
| col\_charsetid | INTEGER |
| col\_charsetform | INTEGER |
| col\_null\_ok | BOOLEAN |

### EXECUTE

The EXECUTE function executes a parsed SQL command or SPL block.

status INTEGER EXECUTE(c INTEGER)

Parameters

c

Cursor ID of the parsed SQL command or SPL block to be executed.

status

Number of rows processed if the SQL command was DELETE, INSERT, or UPDATE. status is meaningless for all other commands.

Examples

The following anonymous block inserts a row into the dept table.

DECLARE

curid INTEGER;

v\_sql VARCHAR2(50);

v\_status INTEGER;

BEGIN

curid := DBMS\_SQL.OPEN\_CURSOR;

v\_sql := 'INSERT INTO dept VALUES (50, ''HR'', ''LOS ANGELES'')';

DBMS\_SQL.PARSE(curid, v\_sql, DBMS\_SQL.native);

v\_status := DBMS\_SQL.EXECUTE(curid);

DBMS\_OUTPUT.PUT\_LINE('Number of rows processed: ' || v\_status);

DBMS\_SQL.CLOSE\_CURSOR(curid);

END;

### EXECUTE\_AND\_FETCH

Function EXECUTE\_AND\_FETCH executes a parsed SELECT command and fetches one row.

status INTEGER EXECUTE\_AND\_FETCH(c INTEGER

[, exact BOOLEAN ])

Parameters

c

Cursor id of the cursor for the SELECT command to be executed.

exact

If set to TRUE, an exception is thrown if the number of rows in the result set is not exactly equal to 1. If set to FALSE, no exception is thrown. The default is FALSE. A NO\_DATA\_FOUND exception is thrown if exact is TRUE and there are no rows in the result set. A TOO\_MANY\_ROWS exception is thrown if exact is TRUE and there is more than one row in the result set.

status

Returns 1 if a row was successfully fetched, 0 if no rows to fetch. If an exception is thrown, no value is returned.

Examples

The following stored procedure uses the EXECUTE\_AND\_FETCH function to retrieve one employee using the employee’s name. An exception will be thrown if the employee is not found, or there is more than one employee with the same name.

CREATE OR REPLACE PROCEDURE select\_by\_name(

p\_ename emp.ename%TYPE

)

IS

curid INTEGER;

v\_empno emp.empno%TYPE;

v\_hiredate emp.hiredate%TYPE;

v\_sal emp.sal%TYPE;

v\_comm emp.comm%TYPE;

v\_dname dept.dname%TYPE;

v\_disp\_date VARCHAR2(10);

v\_sql VARCHAR2(120) := 'SELECT empno, hiredate, sal, ' ||

'NVL(comm, 0), dname ' ||

'FROM emp e, dept d ' ||

'WHERE ename = :p\_ename ' ||

'AND e.deptno = d.deptno';

v\_status INTEGER;

BEGIN

curid := DBMS\_SQL.OPEN\_CURSOR;

DBMS\_SQL.PARSE(curid,v\_sql,DBMS\_SQL.native);

DBMS\_SQL.BIND\_VARIABLE(curid,':p\_ename',UPPER(p\_ename));

DBMS\_SQL.DEFINE\_COLUMN(curid,1,v\_empno);

DBMS\_SQL.DEFINE\_COLUMN(curid,2,v\_hiredate);

DBMS\_SQL.DEFINE\_COLUMN(curid,3,v\_sal);

DBMS\_SQL.DEFINE\_COLUMN(curid,4,v\_comm);

DBMS\_SQL.DEFINE\_COLUMN(curid,5,v\_dname,14);

v\_status := DBMS\_SQL.EXECUTE\_AND\_FETCH(curid,TRUE);

DBMS\_SQL.COLUMN\_VALUE(curid,1,v\_empno);

DBMS\_SQL.COLUMN\_VALUE(curid,2,v\_hiredate);

DBMS\_SQL.COLUMN\_VALUE(curid,3,v\_sal);

DBMS\_SQL.COLUMN\_VALUE(curid,4,v\_comm);

DBMS\_SQL.COLUMN\_VALUE(curid,5,v\_dname);

v\_disp\_date := TO\_CHAR(v\_hiredate, 'MM/DD/YYYY');

DBMS\_OUTPUT.PUT\_LINE('Number : ' || v\_empno);

DBMS\_OUTPUT.PUT\_LINE('Name : ' || UPPER(p\_ename));

DBMS\_OUTPUT.PUT\_LINE('Hire Date : ' || v\_disp\_date);

DBMS\_OUTPUT.PUT\_LINE('Salary : ' || v\_sal);

DBMS\_OUTPUT.PUT\_LINE('Commission: ' || v\_comm);

DBMS\_OUTPUT.PUT\_LINE('Department: ' || v\_dname);

DBMS\_SQL.CLOSE\_CURSOR(curid);

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

DBMS\_OUTPUT.PUT\_LINE('Employee ' || p\_ename || ' not found');

DBMS\_SQL.CLOSE\_CURSOR(curid);

WHEN TOO\_MANY\_ROWS THEN

DBMS\_OUTPUT.PUT\_LINE('Too many employees named, ' ||

p\_ename || ', found');

DBMS\_SQL.CLOSE\_CURSOR(curid);

WHEN OTHERS THEN

DBMS\_OUTPUT.PUT\_LINE('The following is SQLERRM:');

DBMS\_OUTPUT.PUT\_LINE(SQLERRM);

DBMS\_OUTPUT.PUT\_LINE('The following is SQLCODE:');

DBMS\_OUTPUT.PUT\_LINE(SQLCODE);

DBMS\_SQL.CLOSE\_CURSOR(curid);

END;

EXEC select\_by\_name('MARTIN')

Number : 7654

Name : MARTIN

Hire Date : 09/28/1981

Salary : 1250

Commission: 1400

Department: SALES

### FETCH\_ROWS

The FETCH\_ROWS function retrieves a row from a cursor.

status INTEGER FETCH\_ROWS(c INTEGER)

Parameters

c

Cursor ID of the cursor from which to fetch a row.

status

Returns 1 if a row was successfully fetched, 0 if no more rows to fetch.

Examples

The following example fetches the rows from the emp table and displays the results.

DECLARE

curid INTEGER;

v\_empno NUMBER(4);

v\_ename VARCHAR2(10);

v\_hiredate DATE;

v\_sal NUMBER(7,2);

v\_comm NUMBER(7,2);

v\_sql VARCHAR2(50) := 'SELECT empno, ename, hiredate, sal, ' ||

'comm FROM emp';

v\_status INTEGER;

BEGIN

curid := DBMS\_SQL.OPEN\_CURSOR;

DBMS\_SQL.PARSE(curid,v\_sql,DBMS\_SQL.native);

DBMS\_SQL.DEFINE\_COLUMN(curid,1,v\_empno);

DBMS\_SQL.DEFINE\_COLUMN(curid,2,v\_ename,10);

DBMS\_SQL.DEFINE\_COLUMN(curid,3,v\_hiredate);

DBMS\_SQL.DEFINE\_COLUMN(curid,4,v\_sal);

DBMS\_SQL.DEFINE\_COLUMN(curid,5,v\_comm);

v\_status := DBMS\_SQL.EXECUTE(curid);

DBMS\_OUTPUT.PUT\_LINE('EMPNO ENAME HIREDATE SAL COMM');

DBMS\_OUTPUT.PUT\_LINE('----- ---------- ---------- -------- ' ||

'--------');

LOOP

v\_status := DBMS\_SQL.FETCH\_ROWS(curid);

EXIT WHEN v\_status = 0;

DBMS\_SQL.COLUMN\_VALUE(curid,1,v\_empno);

DBMS\_SQL.COLUMN\_VALUE(curid,2,v\_ename);

DBMS\_SQL.COLUMN\_VALUE(curid,3,v\_hiredate);

DBMS\_SQL.COLUMN\_VALUE(curid,4,v\_sal);

DBMS\_SQL.COLUMN\_VALUE(curid,4,v\_sal);

DBMS\_SQL.COLUMN\_VALUE(curid,5,v\_comm);

DBMS\_OUTPUT.PUT\_LINE(v\_empno || ' ' || RPAD(v\_ename,10) || ' ' ||

TO\_CHAR(v\_hiredate,'yyyy-mm-dd') || ' ' ||

TO\_CHAR(v\_sal,'9,999.99') || ' ' ||

TO\_CHAR(NVL(v\_comm,0),'9,999.99'));

END LOOP;

DBMS\_SQL.CLOSE\_CURSOR(curid);

END;

EMPNO ENAME HIREDATE SAL COMM

----- ---------- ---------- -------- --------

7369 SMITH 1980-12-17 800.00 .00

7499 ALLEN 1981-02-20 1,600.00 300.00

7521 WARD 1981-02-22 1,250.00 500.00

7566 JONES 1981-04-02 2,975.00 .00

7654 MARTIN 1981-09-28 1,250.00 1,400.00

7698 BLAKE 1981-05-01 2,850.00 .00

7782 CLARK 1981-06-09 2,450.00 .00

7788 SCOTT 1987-04-19 3,000.00 .00

7839 KING 1981-11-17 5,000.00 .00

7844 TURNER 1981-09-08 1,500.00 .00

7876 ADAMS 1987-05-23 1,100.00 .00

7900 JAMES 1981-12-03 950.00 .00

7902 FORD 1981-12-03 3,000.00 .00

7934 MILLER 1982-01-23 1,300.00 .00

### IS\_OPEN

The IS\_OPEN function provides the capability to test if the given cursor is open.

status BOOLEAN IS\_OPEN(c INTEGER)

Parameters

c

Cursor ID of the cursor to be tested.

status

Set to TRUE if the cursor is open, set to FALSE if the cursor is not open.

### LAST\_ROW\_COUNT

The LAST\_ROW\_COUNT function returns the number of rows that have been currently fetched.

rowcnt INTEGER LAST\_ROW\_COUNT

Parameters

rowcnt

Number of row fetched thus far.

Examples

The following example uses the LAST\_ROW\_COUNT function to display the total number of rows fetched in the query.

DECLARE

curid INTEGER;

v\_empno NUMBER(4);

v\_ename VARCHAR2(10);

v\_hiredate DATE;

v\_sal NUMBER(7,2);

v\_comm NUMBER(7,2);

v\_sql VARCHAR2(50) := 'SELECT empno, ename, hiredate, sal, ' ||

'comm FROM emp';

v\_status INTEGER;

BEGIN

curid := DBMS\_SQL.OPEN\_CURSOR;

DBMS\_SQL.PARSE(curid,v\_sql,DBMS\_SQL.native);

DBMS\_SQL.DEFINE\_COLUMN(curid,1,v\_empno);

DBMS\_SQL.DEFINE\_COLUMN(curid,2,v\_ename,10);

DBMS\_SQL.DEFINE\_COLUMN(curid,3,v\_hiredate);

DBMS\_SQL.DEFINE\_COLUMN(curid,4,v\_sal);

DBMS\_SQL.DEFINE\_COLUMN(curid,5,v\_comm);

v\_status := DBMS\_SQL.EXECUTE(curid);

DBMS\_OUTPUT.PUT\_LINE('EMPNO ENAME HIREDATE SAL COMM');

DBMS\_OUTPUT.PUT\_LINE('----- ---------- ---------- -------- ' ||

'--------');

LOOP

v\_status := DBMS\_SQL.FETCH\_ROWS(curid);

EXIT WHEN v\_status = 0;

DBMS\_SQL.COLUMN\_VALUE(curid,1,v\_empno);

DBMS\_SQL.COLUMN\_VALUE(curid,2,v\_ename);

DBMS\_SQL.COLUMN\_VALUE(curid,3,v\_hiredate);

DBMS\_SQL.COLUMN\_VALUE(curid,4,v\_sal);

DBMS\_SQL.COLUMN\_VALUE(curid,4,v\_sal);

DBMS\_SQL.COLUMN\_VALUE(curid,5,v\_comm);

DBMS\_OUTPUT.PUT\_LINE(v\_empno || ' ' || RPAD(v\_ename,10) || ' ' ||

TO\_CHAR(v\_hiredate,'yyyy-mm-dd') || ' ' ||

TO\_CHAR(v\_sal,'9,999.99') || ' ' ||

TO\_CHAR(NVL(v\_comm,0),'9,999.99'));

END LOOP;

DBMS\_OUTPUT.PUT\_LINE('Number of rows: ' || DBMS\_SQL.LAST\_ROW\_COUNT);

DBMS\_SQL.CLOSE\_CURSOR(curid);

END;

EMPNO ENAME HIREDATE SAL COMM

----- ---------- ---------- -------- --------

7369 SMITH 1980-12-17 800.00 .00

7499 ALLEN 1981-02-20 1,600.00 300.00

7521 WARD 1981-02-22 1,250.00 500.00

7566 JONES 1981-04-02 2,975.00 .00

7654 MARTIN 1981-09-28 1,250.00 1,400.00

7698 BLAKE 1981-05-01 2,850.00 .00

7782 CLARK 1981-06-09 2,450.00 .00

7788 SCOTT 1987-04-19 3,000.00 .00

7839 KING 1981-11-17 5,000.00 .00

7844 TURNER 1981-09-08 1,500.00 .00

7876 ADAMS 1987-05-23 1,100.00 .00

7900 JAMES 1981-12-03 950.00 .00

7902 FORD 1981-12-03 3,000.00 .00

7934 MILLER 1982-01-23 1,300.00 .00

Number of rows: 14

### OPEN\_CURSOR

The OPEN\_CURSOR function creates a new cursor. A cursor must be used to parse and execute any dynamic SQL statement. Once a cursor has been opened, it can be re-used with the same or different SQL statements. The cursor does not have to be closed and re-opened in order to be re-used.

c INTEGER OPEN\_CURSOR

Parameters

c

Cursor ID number associated with the newly created cursor.

Examples

The following example creates a new cursor:

DECLARE

curid INTEGER;

BEGIN

curid := DBMS\_SQL.OPEN\_CURSOR;

.

.

.

END;

### PARSE

The PARSE procedure parses a SQL command or SPL block. If the SQL command is a DDL command, it is immediately executed and does not require running the EXECUTE function.

PARSE(c INTEGER, statement VARCHAR2, language\_flag INTEGER)

Parameters

c

Cursor ID of an open cursor.

statement

SQL command or SPL block to be parsed. A SQL command must not end with the semi-colon terminator, however an SPL block does require the semi-colon terminator.

language\_flag

Use DBMS\_SQL.V6, DBMS\_SQL.V7 or DBMS\_SQL.native. This flag is ignored, and all syntax is assumed to be in Advanced Server form.

Examples

The following anonymous block creates a table named, job. Note that DDL statements are executed immediately by the PARSE procedure and do not require a separate EXECUTE step.

DECLARE

curid INTEGER;

BEGIN

curid := DBMS\_SQL.OPEN\_CURSOR;

DBMS\_SQL.PARSE(curid, 'CREATE TABLE job (jobno NUMBER(3), ' ||

'jname VARCHAR2(9))',DBMS\_SQL.native);

DBMS\_SQL.CLOSE\_CURSOR(curid);

END;

The following inserts two rows into the job table.

DECLARE

curid INTEGER;

v\_sql VARCHAR2(50);

v\_status INTEGER;

BEGIN

curid := DBMS\_SQL.OPEN\_CURSOR;

v\_sql := 'INSERT INTO job VALUES (100, ''ANALYST'')';

DBMS\_SQL.PARSE(curid, v\_sql, DBMS\_SQL.native);

v\_status := DBMS\_SQL.EXECUTE(curid);

DBMS\_OUTPUT.PUT\_LINE('Number of rows processed: ' || v\_status);

v\_sql := 'INSERT INTO job VALUES (200, ''CLERK'')';

DBMS\_SQL.PARSE(curid, v\_sql, DBMS\_SQL.native);

v\_status := DBMS\_SQL.EXECUTE(curid);

DBMS\_OUTPUT.PUT\_LINE('Number of rows processed: ' || v\_status);

DBMS\_SQL.CLOSE\_CURSOR(curid);

END;

Number of rows processed: 1

Number of rows processed: 1

The following anonymous block uses the DBMS\_SQL package to execute a block containing two INSERT statements. Note that the end of the block contains a terminating semi-colon, while in the prior example, each individual INSERT statement does not have a terminating semi-colon.

DECLARE

curid INTEGER;

v\_sql VARCHAR2(100);

v\_status INTEGER;

BEGIN

curid := DBMS\_SQL.OPEN\_CURSOR;

v\_sql := 'BEGIN ' ||

'INSERT INTO job VALUES (300, ''MANAGER''); ' ||

'INSERT INTO job VALUES (400, ''SALESMAN''); ' ||

'END;';

DBMS\_SQL.PARSE(curid, v\_sql, DBMS\_SQL.native);

v\_status := DBMS\_SQL.EXECUTE(curid);

DBMS\_SQL.CLOSE\_CURSOR(curid);

END;

## DBMS\_UTILITY

The DBMS\_UTILITY package provides various utility programs.

Table ‑ DBMS\_UTILITY Functions/Procedures

| Function/Procedure | Function or Procedure | Return Type | Description |
| --- | --- | --- | --- |
| ANALYZE\_DATABASE(method [, estimate\_rows [, estimate\_percent [, method\_opt ]]]) | Procedure | n/a | Analyze database tables. |
| ANALYZE\_PART\_OBJECT(schema, object\_name [, object\_type [, command\_type [, command\_opt [, sample\_clause ]]]]) | Procedure | n/a | Analyze a partitioned table. |
| ANALYZE\_SCHEMA(schema, method [, estimate\_rows [, estimate\_percent [, method\_opt ]]]) | Procedure | n/a | Analyze schema tables. |
| CANONICALIZE(name, canon\_name OUT, canon\_len) | Procedure | n/a | Canonicalizes a string – e.g., strips off white space. |
| COMMA\_TO\_TABLE(list, tablen OUT, tab OUT) | Procedure | n/a | Convert a comma-delimited list of names to a table of names. |
| DB\_VERSION(version OUT, compatibility OUT) | Procedure | n/a | Get the database version. |
| EXEC\_DDL\_STATEMENT(parse\_string) | Procedure | n/a | Execute a DDL statement. |
| FORMAT\_CALL\_STACK | Function | TEXT | Formats the current call stack. |
| GET\_CPU\_TIME | Function | NUMBER | Get the current CPU time. |
| GET\_DEPENDENCY(type, schema, name) | Procedure | n/a | Get objects that are dependent upon the given object.. |
| GET\_HASH\_VALUE(name, base, hash\_size) | Function | NUMBER | Compute a hash value. |
| GET\_PARAMETER\_VALUE(parnam, intval OUT, strval OUT) | Procedure | BINARY\_INTEGER | Get database initialization parameter settings. |
| GET\_TIME | Function | NUMBER | Get the current time. |
| NAME\_TOKENIZE(name, a OUT, b OUT, c OUT, dblink OUT, nextpos OUT) | Procedure | n/a | Parse the given name into its component parts. |
| TABLE\_TO\_COMMA(tab, tablen OUT, list OUT) | Procedure | n/a | Convert a table of names to a comma-delimited list. |

The following table lists the public variables available in the DBMS\_UTILITY package.

Table ‑ DBMS\_UTILITY Public Variables

| Public Variables | Data Type | Value | Description |
| --- | --- | --- | --- |
| inv\_error\_on\_restrictions | PLS\_INTEGER | 1 | Used by the INVALIDATE procedure. |
| lname\_array | TABLE |  | For lists of long names. |
| uncl\_array | TABLE |  | For lists of users and names. |

### LNAME\_ARRAY

The LNAME\_ARRAY is for storing lists of long names including fully-qualified names.

TYPE lname\_array IS TABLE OF VARCHAR2(4000) INDEX BY BINARY\_INTEGER;

### UNCL\_ARRAY

The UNCL\_ARRAY is for storing lists of users and names.

TYPE uncl\_array IS TABLE OF VARCHAR2(227) INDEX BY BINARY\_INTEGER;

### ANALYZE\_DATABASE, ANALYZE SCHEMA and ANALYZE PART\_OBJECT

The ANALYZE\_DATABASE(), ANALYZE\_SCHEMA() and ANALYZE\_PART\_OBJECT() procedures provide the capability to gather statistics on tables in the database. When you execute the ANALYZE statement, Postgres samples the data in a table and records distribution statistics in the pg\_statistics system table.

ANALYZE\_DATABASE, ANALYZE\_SCHEMA, and ANALYZE\_PART\_OBJECT differ primarily in the number of tables that are processed:

* ANALYZE\_DATABASE analyzes all tables in all schemas within the current database.
* ANALYZE\_SCHEMA analyzes all tables in a given schema (within the current database).
* ANALYZE\_PART\_OBJECT analyzes a single table.

The syntax for the ANALYZE commands are:

ANALYZE\_DATABASE(method VARCHAR2 [, estimate\_rows NUMBER

[, estimate\_percent NUMBER [, method\_opt VARCHAR2 ]]])

ANALYZE\_SCHEMA(schema VARCHAR2, method VARCHAR2

[, estimate\_rows NUMBER [, estimate\_percent NUMBER

[, method\_opt VARCHAR2 ]]])

ANALYZE\_PART\_OBJECT(schema VARCHAR2, object\_name VARCHAR2

[, object\_type CHAR [, command\_type CHAR

[, command\_opt VARCHAR2 [, sample\_clause ]]]])

Parameters – ANALYZE\_DATABASE and ANALYZE\_SCHEMA

method

method determines whether the ANALYZE procedure populates the pg\_statistics table or removes entries from the pg\_statistics table. If you specify a method of DELETE, the ANALYZE procedure removes the relevant rows from pg\_statistics. If you specify a method of COMPUTE or ESTIMATE, the ANALYZE procedure analyzes a table (or multiple tables) and records the distribution information in pg\_statistics. There is no difference between COMPUTE and ESTIMATE; both methods execute the Postgres ANALYZE statement. All other parameters are validated and then ignored.

estimate\_rows

Number of rows upon which to base estimated statistics. One of estimate\_rows or estimate\_percent must be specified if method is ESTIMATE.

This argument is ignored, but is included for compatibility.

estimate\_percent

Percentage of rows upon which to base estimated statistics. One of estimate\_rows or estimate\_percent must be specified if method is ESTIMATE.

This argument is ignored, but is included for compatibility.

method\_opt

Object types to be analyzed. Any combination of the following:

[ FOR TABLE ]

[ FOR ALL [ INDEXED ] COLUMNS ] [ SIZE n ]

[ FOR ALL INDEXES ]

This argument is ignored, but is included for compatibility.

Parameters – ANALYZE\_PART\_OBJECT

schema

Name of the schema whose objects are to be analyzed.

object\_name

Name of the partitioned object to be analyzed.

object\_type

Type of object to be analyzed. Valid values are: T – table, I – index.

This argument is ignored, but is included for compatibility.

command\_type

Type of analyze functionality to perform. Valid values are: E - gather estimated statistics based upon on a specified number of rows or a percentage of rows in the sample\_clause clause; C - compute exact statistics; or V – validate the structure and integrity of the partitions.

This argument is ignored, but is included for compatibility.

command\_opt

For command\_type C or E, can be any combination of:

[ FOR TABLE ]

[ FOR ALL COLUMNS ]

[ FOR ALL LOCAL INDEXES ]

For command\_type V, can be CASCADE if object\_type is T.

This argument is ignored, but is included for compatibility.

sample\_clause

If command\_type is E, contains the following clause to specify the number of rows or percentage or rows on which to base the estimate.

SAMPLE n { ROWS | PERCENT }

This argument is ignored, but is included for compatibility.

### CANONICALIZE

The CANONICALIZE procedure performs the following operations on an input string:

* If the string is not double-quoted, verifies that it uses the characters of a legal identifier. If not, an exception is thrown. If the string is double-quoted, all characters are allowed.
* If the string is not double-quoted and does not contain periods, uppercases all alphabetic characters and eliminates leading and trailing spaces.
* If the string is double-quoted and does not contain periods, strips off the double quotes.
* If the string contains periods and no portion of the string is double-quoted, uppercases each portion of the string and encloses each portion in double quotes.
* If the string contains periods and portions of the string are double-quoted, returns the double-quoted portions unchanged including the double quotes and returns the non-double-quoted portions uppercased and enclosed in double quotes.

CANONICALIZE(name VARCHAR2, canon\_name OUT VARCHAR2,

canon\_len BINARY\_INTEGER)

Parameters

name

String to be canonicalized.

canon\_name

The canonicalized string.

canon\_len

Number of bytes in name to canonicalize starting from the first character.

Examples

The following procedure applies the CANONICALIZE procedure on its input parameter and displays the results.

CREATE OR REPLACE PROCEDURE canonicalize (

p\_name VARCHAR2,

p\_length BINARY\_INTEGER DEFAULT 30

)

IS

v\_canon VARCHAR2(100);

BEGIN

DBMS\_UTILITY.CANONICALIZE(p\_name,v\_canon,p\_length);

DBMS\_OUTPUT.PUT\_LINE('Canonicalized name ==>' || v\_canon || '<==');

DBMS\_OUTPUT.PUT\_LINE('Length: ' || LENGTH(v\_canon));

EXCEPTION

WHEN OTHERS THEN

DBMS\_OUTPUT.PUT\_LINE('SQLERRM: ' || SQLERRM);

DBMS\_OUTPUT.PUT\_LINE('SQLCODE: ' || SQLCODE);

END;

EXEC canonicalize('Identifier')

Canonicalized name ==>IDENTIFIER<==

Length: 10

EXEC canonicalize('"Identifier"')

Canonicalized name ==>Identifier<==

Length: 10

EXEC canonicalize('"\_+142%"')

Canonicalized name ==>\_+142%<==

Length: 6

EXEC canonicalize('abc.def.ghi')

Canonicalized name ==>"ABC"."DEF"."GHI"<==

Length: 17

EXEC canonicalize('"abc.def.ghi"')

Canonicalized name ==>abc.def.ghi<==

Length: 11

EXEC canonicalize('"abc".def."ghi"')

Canonicalized name ==>"abc"."DEF"."ghi"<==

Length: 17

EXEC canonicalize('"abc.def".ghi')

Canonicalized name ==>"abc.def"."GHI"<==

Length: 15

### COMMA\_TO\_TABLE

The COMMA\_TO\_TABLE procedure converts a comma-delimited list of names into a table of names. Each entry in the list becomes a table entry. The names must be formatted as valid identifiers.

COMMA\_TO\_TABLE(list VARCHAR2, tablen OUT BINARY\_INTEGER,

tab OUT { LNAME\_ARRAY | UNCL\_ARRAY })

Parameters

list

Comma-delimited list of names.

tablen

Number of entries in tab.

tab

Table containing the individual names in list.

LNAME\_ARRAY

A DBMS\_UTILITY LNAME\_ARRAY (as described in Section 9.15.1).

UNCL\_ARRAY

A DBMS\_UTILITY UNCL\_ARRAY (as described in Section 9.15.2).

Examples

The following procedure uses the COMMA\_TO\_TABLE procedure to convert a list of names to a table. The table entries are then displayed.

CREATE OR REPLACE PROCEDURE comma\_to\_table (

p\_list VARCHAR2

)

IS

r\_lname DBMS\_UTILITY.LNAME\_ARRAY;

v\_length BINARY\_INTEGER;

BEGIN

DBMS\_UTILITY.COMMA\_TO\_TABLE(p\_list,v\_length,r\_lname);

FOR i IN 1..v\_length LOOP

DBMS\_OUTPUT.PUT\_LINE(r\_lname(i));

END LOOP;

END;

EXEC comma\_to\_table('edb.dept, edb.emp, edb.jobhist')

edb.dept

edb.emp

edb.jobhist

### DB\_VERSION

The DB\_VERSION procedure returns the version number of the database.

DB\_VERSION(version OUT VARCHAR2, compatibility OUT VARCHAR2)

Parameters

version

Database version number.

compatibility

Compatibility setting of the database. (To be implementation-defined as to its meaning.)

Examples

The following anonymous block displays the database version information.

DECLARE

v\_version VARCHAR2(150);

v\_compat VARCHAR2(150);

BEGIN

DBMS\_UTILITY.DB\_VERSION(v\_version,v\_compat);

DBMS\_OUTPUT.PUT\_LINE('Version: ' || v\_version);

DBMS\_OUTPUT.PUT\_LINE('Compatibility: ' || v\_compat);

END;

Version: EnterpriseDB 9.5.0.0 on i686-pc-linux-gnu, compiled by GCC gcc (GCC) 4.1.2 20080704 (Red Hat 4.1.2-48), 32-bit

Compatibility: EnterpriseDB 9.5.0.0 on i686-pc-linux-gnu, compiled by GCC gcc (GCC) 4.1.220080704 (Red Hat 4.1.2-48), 32-bit

### EXEC\_DDL\_STATEMENT

The EXEC\_DDL\_STATEMENT provides the capability to execute a DDL command.

EXEC\_DDL\_STATEMENT(parse\_string VARCHAR2)

Parameters

parse\_string

The DDL command to be executed.

Examples

The following anonymous block creates the job table.

BEGIN

DBMS\_UTILITY.EXEC\_DDL\_STATEMENT(

'CREATE TABLE job (' ||

'jobno NUMBER(3),' ||

'jname VARCHAR2(9))'

);

END;

If the parse\_string does not include a valid DDL statement, Advanced Server returns the following error:

edb=#  exec dbms\_utility.exec\_ddl\_statement('select rownum from dual');

ERROR:  EDB-20001: 'parse\_string' must be a valid DDL statement

### FORMAT\_CALL\_STACK

The FORMAT\_CALL\_STACK function returns the formatted contents of the current call stack.

DBMS\_UTILITY.FORMAT\_CALL\_STACK  
return VARCHAR2

This function can be used in a stored procedure, function or package to return the current call stack in a readable format. This function is useful for debugging purposes.

### GET\_CPU\_TIME

The GET\_CPU\_TIME function returns the CPU time in hundredths of a second from some arbitrary point in time.

cputime NUMBER GET\_CPU\_TIME

Parameters

cputime

Number of hundredths of a second of CPU time.

Examples

The following SELECT command retrieves the current CPU time, which is 603 hundredths of a second or .0603 seconds.

SELECT DBMS\_UTILITY.GET\_CPU\_TIME FROM DUAL;

get\_cpu\_time

--------------

603

### GET\_DEPENDENCY

The GET\_DEPENDENCY procedure provides the capability to list the objects that are dependent upon the specified object. GET\_DEPENDENCY does not show dependencies for functions or procedures.

GET\_DEPENDENCY(type VARCHAR2, schema VARCHAR2,

name VARCHAR2)

Parameters

type

The object type of name. Valid values are INDEX, PACKAGE, PACKAGE BODY, SEQUENCE, TABLE, TRIGGER, TYPE and VIEW.

schema

Name of the schema in which name exists.

name

Name of the object for which dependencies are to be obtained.

Examples

The following anonymous block finds dependencies on the EMP table.

BEGIN

DBMS\_UTILITY.GET\_DEPENDENCY('TABLE','public','EMP');

END;

DEPENDENCIES ON public.EMP

------------------------------------------------------------------

\*TABLE public.EMP()

\* CONSTRAINT c public.emp()

\* CONSTRAINT f public.emp()

\* CONSTRAINT p public.emp()

\* TYPE public.emp()

\* CONSTRAINT c public.emp()

\* CONSTRAINT f public.jobhist()

\* VIEW .empname\_view()

### GET\_HASH\_VALUE

The GET\_HASH\_VALUE function provides the capability to compute a hash value for a given string.

hash NUMBER GET\_HASH\_VALUE(name VARCHAR2, base NUMBER,

hash\_size NUMBER)

Parameters

name

The string for which a hash value is to be computed.

base

Starting value at which hash values are to be generated.

hash\_size

The number of hash values for the desired hash table.

hash

The generated hash value.

Examples

The following anonymous block creates a table of hash values using the ename column of the emp table and then displays the key along with the hash value. The hash values start at 100 with a maximum of 1024 distinct values.

DECLARE

v\_hash NUMBER;

TYPE hash\_tab IS TABLE OF NUMBER INDEX BY VARCHAR2(10);

r\_hash HASH\_TAB;

CURSOR emp\_cur IS SELECT ename FROM emp;

BEGIN

FOR r\_emp IN emp\_cur LOOP

r\_hash(r\_emp.ename) :=

DBMS\_UTILITY.GET\_HASH\_VALUE(r\_emp.ename,100,1024);

END LOOP;

FOR r\_emp IN emp\_cur LOOP

DBMS\_OUTPUT.PUT\_LINE(RPAD(r\_emp.ename,10) || ' ' ||

r\_hash(r\_emp.ename));

END LOOP;

END;

SMITH 377

ALLEN 740

WARD 718

JONES 131

MARTIN 176

BLAKE 568

CLARK 621

SCOTT 1097

KING 235

TURNER 850

ADAMS 156

JAMES 942

FORD 775

MILLER 148

### GET\_PARAMETER\_VALUE

The GET\_PARAMETER\_VALUE procedure provides the capability to retrieve database initialization parameter settings.

status BINARY\_INTEGER GET\_PARAMETER\_VALUE(parnam VARCHAR2,

intval OUT INTEGER, strval OUT VARCHAR2)

Parameters

parnam

Name of the parameter whose value is to be returned. The parameters are listed in the pg\_settings system view.

intval

Value of an integer parameter or the length of strval.

strval

Value of a string parameter.

status

Returns 0 if the parameter value is INTEGER or BOOLEAN. Returns 1 if the parameter value is a string.

Examples

The following anonymous block shows the values of two initialization parameters.

DECLARE

v\_intval INTEGER;

v\_strval VARCHAR2(80);

BEGIN

DBMS\_UTILITY.GET\_PARAMETER\_VALUE('max\_fsm\_pages', v\_intval, v\_strval);

DBMS\_OUTPUT.PUT\_LINE('max\_fsm\_pages' || ': ' || v\_intval);

DBMS\_UTILITY.GET\_PARAMETER\_VALUE('client\_encoding', v\_intval, v\_strval);

DBMS\_OUTPUT.PUT\_LINE('client\_encoding' || ': ' || v\_strval);

END;

max\_fsm\_pages: 72625

client\_encoding: SQL\_ASCII

### GET\_TIME

The GET\_TIME function provides the capability to return the current time in hundredths of a second.

time NUMBER GET\_TIME

Parameters

time

Number of hundredths of a second from the time in which the program is started.

Examples

The following example shows calls to the GET\_TIME function.

SELECT DBMS\_UTILITY.GET\_TIME FROM DUAL;

get\_time

----------

1555860

SELECT DBMS\_UTILITY.GET\_TIME FROM DUAL;

get\_time

----------

1556037

### NAME\_TOKENIZE

The NAME\_TOKENIZE procedure parses a name into its component parts. Names without double quotes are uppercased. The double quotes are stripped from names with double quotes.

NAME\_TOKENIZE(name VARCHAR2, a OUT VARCHAR2, b OUT VARCHAR2,

c OUT VARCHAR2, dblink OUT VARCHAR2,

nextpos OUT BINARY\_INTEGER)

Parameters

name

String containing a name in the following format:

a[.b[.c]][@dblink ]

a

Returns the leftmost component.

b

Returns the second component, if any.

c

Returns the third component, if any.

dblink

Returns the database link name.

nextpos

Position of the last character parsed in name.

Examples

The following stored procedure is used to display the returned parameter values of the NAME\_TOKENIZE procedure for various names.

CREATE OR REPLACE PROCEDURE name\_tokenize (

p\_name VARCHAR2

)

IS

v\_a VARCHAR2(30);

v\_b VARCHAR2(30);

v\_c VARCHAR2(30);

v\_dblink VARCHAR2(30);

v\_nextpos BINARY\_INTEGER;

BEGIN

DBMS\_UTILITY.NAME\_TOKENIZE(p\_name,v\_a,v\_b,v\_c,v\_dblink,v\_nextpos);

DBMS\_OUTPUT.PUT\_LINE('name : ' || p\_name);

DBMS\_OUTPUT.PUT\_LINE('a : ' || v\_a);

DBMS\_OUTPUT.PUT\_LINE('b : ' || v\_b);

DBMS\_OUTPUT.PUT\_LINE('c : ' || v\_c);

DBMS\_OUTPUT.PUT\_LINE('dblink : ' || v\_dblink);

DBMS\_OUTPUT.PUT\_LINE('nextpos: ' || v\_nextpos);

END;

Tokenize the name, emp:

BEGIN

name\_tokenize('emp');

END;

name : emp

a : EMP

b :

c :

dblink :

nextpos: 3

Tokenize the name, edb.list\_emp:

BEGIN

name\_tokenize('edb.list\_emp');

END;

name : edb.list\_emp

a : EDB

b : LIST\_EMP

c :

dblink :

nextpos: 12

Tokenize the name, "edb"."Emp\_Admin".update\_emp\_sal:

BEGIN

name\_tokenize('"edb"."Emp\_Admin".update\_emp\_sal');

END;

name : "edb"."Emp\_Admin".update\_emp\_sal

a : edb

b : Emp\_Admin

c : UPDATE\_EMP\_SAL

dblink :

nextpos: 32

Tokenize the name edb.emp@edb\_dblink:

BEGIN

name\_tokenize('edb.emp@edb\_dblink');

END;

name : edb.emp@edb\_dblink

a : EDB

b : EMP

c :

dblink : EDB\_DBLINK

nextpos: 18

### TABLE\_TO\_COMMA

The TABLE\_TO\_COMMA procedure converts table of names into a comma-delimited list of names. Each table entry becomes a list entry. The names must be formatted as valid identifiers.

TABLE\_TO\_COMMA(tab { LNAME\_ARRAY | UNCL\_ARRAY },

tablen OUT BINARY\_INTEGER, list OUT VARCHAR2)

Parameters

tab

Table containing names.

LNAME\_ARRAY

A DBMS\_UTILITY LNAME\_ARRAY (as described in Section 9.15.1).

UNCL\_ARRAY

A DBMS\_UTILITY UNCL\_ARRAY (as described in Section 9.15.2).

tablen

Number of entries in list.

list

Comma-delimited list of names from tab.

Examples

The following example first uses the COMMA\_TO\_TABLE procedure to convert a comma-delimited list to a table. The TABLE\_TO\_COMMA procedure then converts the table back to a comma-delimited list that is displayed.

CREATE OR REPLACE PROCEDURE table\_to\_comma (

p\_list VARCHAR2

)

IS

r\_lname DBMS\_UTILITY.LNAME\_ARRAY;

v\_length BINARY\_INTEGER;

v\_listlen BINARY\_INTEGER;

v\_list VARCHAR2(80);

BEGIN

DBMS\_UTILITY.COMMA\_TO\_TABLE(p\_list,v\_length,r\_lname);

DBMS\_OUTPUT.PUT\_LINE('Table Entries');

DBMS\_OUTPUT.PUT\_LINE('-------------');

FOR i IN 1..v\_length LOOP

DBMS\_OUTPUT.PUT\_LINE(r\_lname(i));

END LOOP;

DBMS\_OUTPUT.PUT\_LINE('-------------');

DBMS\_UTILITY.TABLE\_TO\_COMMA(r\_lname,v\_listlen,v\_list);

DBMS\_OUTPUT.PUT\_LINE('Comma-Delimited List: ' || v\_list);

END;

EXEC table\_to\_comma('edb.dept, edb.emp, edb.jobhist')

Table Entries

-------------

edb.dept

edb.emp

edb.jobhist

-------------

Comma-Delimited List: edb.dept, edb.emp, edb.jobhist

## UTL\_ENCODE

The UTL\_ENCODE package provides a way to encode and decode data.

Table 7.7.2 UTL\_ENCODE Functions and Procedures

| Function/Procedure | Return Type | Description |
| --- | --- | --- |
| BASE64\_DECODE(r) | RAW | Use the BASE64\_DECODE function to translate a Base64 encoded string to the original RAW value. |
| BASE64\_ENCODE(r) | RAW | Use the BASE64\_ENCODE function to translate a RAW string to an encoded Base64 value. |
| BASE64\_ENCODE(loid) | TEXT | Use the BASE64\_ENCODE function to translate a TEXT string to an encoded Base64 value. |
| MIMEHEADER\_DECODE(buf) | VARCHAR2 | Use the MIMEHEADER\_DECODE function to translate an encoded MIMEHEADER formatted string to its original value. |
| MIMEHEADER\_ENCODE(buf, encode\_charset, encoding) | VARCHAR2 | Use the MIMEHEADER\_ENCODE function to convert and encode a string in MIMEHEADER format. |
| QUOTED\_PRINTABLE\_DECODE(r) | RAW | Use the QUOTED\_PRINTABLE\_DECODE function to translate an encoded string to a RAW value. |
| QUOTED\_PRINTABLE\_ENCODE(r) | RAW | Use the QUOTED\_PRINTABLE\_ENCODE function to translate an input string to a quoted-printable formatted RAW value. |
| TEXT\_DECODE(buf, encode\_charset, encoding) | VARCHAR2 | Use the TEXT\_DECODE function to decode a string encoded by TEXT\_ENCODE. |
| TEXT\_ENCODE(buf, encode\_charset, encoding) | VARCHAR2 | Use the TEXT\_ENCODE function to translate a string to a user-specified character set, and then encode the string. |
| UUDECODE(r) | RAW | Use the UUDECODE function to translate a uuencode encoded string to a RAW value. |
| UUENCODE(r, type, filename, permission) | RAW | Use the UUENCODE function to translate a RAW string to an encoded uuencode value. |

### BASE64\_DECODE

Use the BASE64\_DECODE function to translate a Base64 encoded string to the original value originally encoded by BASE64\_ENCODE. The signature is:

BASE64\_DECODE(r IN RAW)

This function returns a RAW value.

Parameters

r

ris the string that contains the Base64 encoded data that will be translated to RAW form.

Examples

Note: Before executing the following example, invoke the command:

SET bytea\_output = escape;

This command instructs the server to escape any non-printable characters, and to display BYTEA or RAW values onscreen in readable form. For more information, please refer to the Postgres Core Documentation available at:

<http://www.postgresql.org/docs/9.5/static/datatype-binary.html>

The following example first encodes (using BASE64\_ENCODE), and then decodes (using BASE64\_DECODE) a string that contains the text abc:

edb=# SELECT UTL\_ENCODE.BASE64\_ENCODE(CAST ('abc' AS RAW));

base64\_encode

---------------

YWJj

(1 row)

edb=# SELECT UTL\_ENCODE.BASE64\_DECODE(CAST ('YWJj' AS RAW));

base64\_decode

---------------

abc

(1 row)

### BASE64\_ENCODE

Use the BASE64\_ENCODE function to translate and encode a string in Base64 format (as described in RFC 4648). This function can be useful when composing MIME email that you intend to send using the UTL\_SMTP package. The BASE64\_ENCODE function has two signatures:

BASE64\_ENCODE(r IN RAW)

and

BASE64\_ENCODE(loid IN OID)

This function returns a RAW value or an OID.

Parameters

r

rspecifies the RAW string that will be translated to Base64.

loid

loid specifies the object ID of a large object that will be translated to Base64.

Examples

Note: Before executing the following example, invoke the command:

SET bytea\_output = escape;

This command instructs the server to escape any non-printable characters, and to display BYTEA or RAW values onscreen in readable form. For more information, please refer to the Postgres Core Documentation available at:

<http://www.postgresql.org/docs/9.5/static/datatype-binary.html>

The following example first encodes (using BASE64\_ENCODE), and then decodes (using BASE64\_DECODE) a string that contains the text abc:

edb=# SELECT UTL\_ENCODE.BASE64\_ENCODE(CAST ('abc' AS RAW));

base64\_encode

---------------

YWJj

(1 row)

edb=# SELECT UTL\_ENCODE.BASE64\_DECODE(CAST ('YWJj' AS RAW));

base64\_decode

---------------

abc

(1 row)

### MIMEHEADER\_DECODE

Use the MIMEHEADER\_DECODE function to decode values that are encoded by the MIMEHEADER\_ENCODE function. The signature is:

MIMEHEADER\_DECODE(buf IN VARCHAR2)

This function returns a VARCHAR2 value.

Parameters

buf

bufcontains the value (encoded by MIMEHEADER\_ENCODE) that will be decoded.

Examples

The following examples use the MIMEHEADER\_ENCODE and MIMEHEADER\_DECODE functions to first encode, and then decode a string:

edb=# SELECT UTL\_ENCODE.MIMEHEADER\_ENCODE('What is the date?') FROM DUAL;

mimeheader\_encode

------------------------------

=?UTF8?Q?What is the date??=

(1 row)

edb=# SELECT UTL\_ENCODE.MIMEHEADER\_DECODE('=?UTF8?Q?What is the date??=') FROM DUAL;

mimeheader\_decode

-------------------

What is the date?

(1 row)

### MIMEHEADER\_ENCODE

Use the MIMEHEADER\_ENCODE function to convert a string into mime header format, and then encode the string. The signature is:

MIMEHEADER\_ENCODE(buf IN VARCHAR2, encode\_charset IN VARCHAR2 DEFAULT NULL, encoding IN INTEGER DEFAULT NULL)

This function returns a VARCHAR2 value.

Parameters

buf

bufcontains the string that will be formatted and encoded. The string is a VARCHAR2 value.

encode\_charset

encode\_charsetspecifies the character set to which the string will be converted before being formatted and encoded. The default value is NULL.

encoding

encodingspecifies the encoding type used when encoding the string. You can specify:

* Q to enable quoted-printable encoding. If you do not specify a value, MIMEHEADER\_ENCODE will use quoted-printable encoding.
* B to enable base-64 encoding.

Examples

The following examples use the MIMEHEADER\_ENCODE and MIMEHEADER\_DECODE functions to first encode, and then decode a string:

edb=# SELECT UTL\_ENCODE.MIMEHEADER\_ENCODE('What is the date?') FROM DUAL;

mimeheader\_encode

------------------------------

=?UTF8?Q?What is the date??=

(1 row)

edb=# SELECT UTL\_ENCODE.MIMEHEADER\_DECODE('=?UTF8?Q?What is the date??=') FROM DUAL;

mimeheader\_decode

-------------------

What is the date?

(1 row)

### QUOTED\_PRINTABLE\_DECODE

Use the QUOTED\_PRINTABLE\_DECODE function to translate an encoded quoted-printable string into a decoded RAW string.

The signature is:

QUOTED\_PRINTABLE\_DECODE(r IN RAW)

This function returns a RAW value.

Parameters

r

rcontains the encoded string that will be decoded. The string is a RAW value, encoded by QUOTED\_PRINTABLE\_ENCODE.

Examples

Note: Before executing the following example, invoke the command:

SET bytea\_output = escape;

This command instructs the server to escape any non-printable characters, and to display BYTEA or RAW values onscreen in readable form. For more information, please refer to the Postgres Core Documentation available at:

<http://www.postgresql.org/docs/9.5/static/datatype-binary.html>

The following example first encodes and then decodes a string:

edb=# SELECT UTL\_ENCODE.QUOTED\_PRINTABLE\_ENCODE('E=mc2') FROM DUAL; quoted\_printable\_encode

-------------------------

E=3Dmc2

(1 row)

edb=# SELECT UTL\_ENCODE.QUOTED\_PRINTABLE\_DECODE('E=3Dmc2') FROM DUAL;

quoted\_printable\_decode

-------------------------

E=mc2

(1 row)

### QUOTED\_PRINTABLE\_ENCODE

Use the QUOTED\_PRINTABLE\_ENCODE function to translate and encode a string in quoted-printable format. The signature is:

QUOTED\_PRINTABLE\_ENCODE(r IN RAW)

This function returns a RAW value.

Parameters

r

rcontains the string (a RAW value) that will be encoded in a quoted-printable format.

Examples

Note: Before executing the following example, invoke the command:

SET bytea\_output = escape;

This command instructs the server to escape any non-printable characters, and to display BYTEA or RAW values onscreen in readable form. For more information, please refer to the Postgres Core Documentation available at:

<http://www.postgresql.org/docs/9.5/static/datatype-binary.html>

The following example first encodes and then decodes a string:

edb=# SELECT UTL\_ENCODE.QUOTED\_PRINTABLE\_ENCODE('E=mc2') FROM DUAL; quoted\_printable\_encode

-------------------------

E=3Dmc2

(1 row)

edb=# SELECT UTL\_ENCODE.QUOTED\_PRINTABLE\_DECODE('E=3Dmc2') FROM DUAL;

quoted\_printable\_decode

-------------------------

E=mc2

(1 row)

### TEXT\_DECODE

Use the TEXT\_DECODE function to translate and decode an encoded string to the VARCHAR2 value that was originally encoded by the TEXT\_ENCODE function. The signature is:

TEXT\_DECODE(buf IN VARCHAR2, encode\_charset IN VARCHAR2 DEFAULT NULL, encoding IN PLS\_INTEGER DEFAULT NULL)

This function returns a VARCHAR2 value.

Parameters

buf

bufcontains the encoded string that will be translated to the original value encoded by TEXT\_ENCODE.

encode\_charset

encode\_charset specifies the character set to which the string will be translated before encoding. The default value is NULL.

encoding

encoding specifies the encoding type used by TEXT\_DECODE. Specify:

* UTL\_ENCODE.BASE64 to specify base-64 encoding.
* UTL\_ENCODE.QUOTED\_PRINTABLE to specify quoted printable encoding. This is the default.

Examples

The following example uses the TEXT\_ENCODE and TEXT\_DECODE functions to first encode, and then decode a string:

edb=# SELECT UTL\_ENCODE.TEXT\_ENCODE('What is the date?', 'BIG5', UTL\_ENCODE.BASE64) FROM DUAL;

text\_encode

--------------------------

V2hhdCBpcyB0aGUgZGF0ZT8=

(1 row)

edb=# SELECT UTL\_ENCODE.TEXT\_DECODE('V2hhdCBpcyB0aGUgZGF0ZT8=', 'BIG5', UTL\_ENCODE.BASE64) FROM DUAL;

text\_decode

-------------------

What is the date?

(1 row)

### TEXT\_ENCODE

Use the TEXT\_ENCODE function to translate a string to a user-specified character set, and then encode the string. The signature is:

TEXT\_DECODE(buf IN VARCHAR2, encode\_charset IN VARCHAR2 DEFAULT NULL, encoding IN PLS\_INTEGER DEFAULT NULL)

This function returns a VARCHAR2 value.

Parameters

buf

bufcontains the encoded string that will be translated to the specified character set and encoded by TEXT\_ENCODE.

encode\_charset

encode\_charset specifies the character set to which the value will be translated before encoding. The default value is NULL.

encoding

encoding specifies the encoding type used by TEXT\_ENCODE. Specify:

* UTL\_ENCODE.BASE64 to specify base-64 encoding.
* UTL\_ENCODE.QUOTED\_PRINTABLE to specify quoted printable encoding. This is the default.

Examples

The following example uses the TEXT\_ENCODE and TEXT\_DECODE functions to first encode, and then decode a string:

edb=# SELECT UTL\_ENCODE.TEXT\_ENCODE('What is the date?', 'BIG5', UTL\_ENCODE.BASE64) FROM DUAL;

text\_encode

--------------------------

V2hhdCBpcyB0aGUgZGF0ZT8=

(1 row)

edb=# SELECT UTL\_ENCODE.TEXT\_DECODE('V2hhdCBpcyB0aGUgZGF0ZT8=', 'BIG5', UTL\_ENCODE.BASE64) FROM DUAL;

text\_decode

-------------------

What is the date?

(1 row)

### UUDECODE

Use the UUDECODE function to translate and decode a uuencode encoded string to the RAW value that was originally encoded by the UUENCODE function. The signature is:

UUDECODE(r IN RAW)

This function returns a RAW value.

Note: If you are using the Advanced Server UUDECODE function to decode uuencoded data that was created by the Oracle implementation of the UTL\_ENCODE.UUENCODE function, then you must first set the Advanced Server configuration parameter utl\_encode.uudecode\_redwood to TRUE before invoking the Advanced Server UUDECODE function on the Oracle-created data. (For example, this situation may occur if you migrated Oracle tables containing uuencoded data to an Advanced Server database.)

The uuencoded data created by the Oracle version of the UUENCODE function results in a format that differs from the uuencoded data created by the Advanced Server UUENCODE function. As a result, attempting to use the Advanced Server UUDECODE function on the Oracle uuencoded data results in an error unless the configuration parameter utl\_encode.uudecode\_redwood is set to TRUE.

However, if you are using the Advanced Server UUDECODE function on uuencoded data created by the Advanced Server UUENCODE function, then utl\_encode.uudecode\_redwood must be set to FALSE, which is the default setting.

Parameters

r

rcontains the uuencoded string that will be translated to RAW.

Examples

Note: Before executing the following example, invoke the command:

SET bytea\_output = escape;

This command instructs the server to escape any non-printable characters, and to display BYTEA or RAW values onscreen in readable form. For more information, please refer to the Postgres Core Documentation available at:

<http://www.postgresql.org/docs/9.5/static/datatype-binary.html>

The following example uses UUENCODE and UUDECODE to first encode and then decode a string:

edb=# SET bytea\_output = escape;

SET

edb=# SELECT UTL\_ENCODE.UUENCODE('What is the date?') FROM DUAL;

uuencode

--------------------------------------------------------------------

begin 0 uuencode.txt\01215VAA="!I<R!T:&4@9&%T93\\`\012`\012end\012

(1 row)

edb=# SELECT UTL\_ENCODE.UUDECODE

edb-# ('begin 0 uuencode.txt\01215VAA="!I<R!T:&4@9&%T93\\`\012`\012end\012')

edb-# FROM DUAL;

uudecode

-------------------

What is the date?

(1 row)

### UUENCODE

Use the UUENCODE function to translate RAW data into a uuencode formatted encoded string. The signature is:

UUENCODE(r IN RAW, type IN INTEGER DEFAULT 1, filename IN VARCHAR2 DEFAULT NULL, permission IN VARCHAR2 DEFAULT NULL)

This function returns a RAW value.

Parameters

r

rcontains the RAW string that will be translated to uuencode format.

type

type is an INTEGER value or constant that specifies the type of uuencoded string that will be returned; the default value is 1. The possible values are:

|  |  |
| --- | --- |
| Value | Constant |
| 1 | complete |
| 2 | header\_piece |
| 3 | middle\_piece |
| 4 | end\_piece |

filename

filename is a VARCHAR2 value that specifies the file name that you want to embed in the encoded form; if you do not specify a file name, UUENCODE will include a filename of uuencode.txt in the encoded form.

permission

permission is a VARCHAR2 that specifies the permission mode; the default value is NULL.

Examples

Note: Before executing the following example, invoke the command:

SET bytea\_output = escape;

This command instructs the server to escape any non-printable characters, and to display BYTEA or RAW values onscreen in readable form.

For more information, please refer to the Postgres Core Documentation available at:

<http://www.postgresql.org/docs/9.5/static/datatype-binary.html>

The following example uses UUENCODE and UUDECODE to first encode and then decode a string:

edb=# SET bytea\_output = escape;

SET

edb=# SELECT UTL\_ENCODE.UUENCODE('What is the date?') FROM DUAL;

uuencode

--------------------------------------------------------------------

begin 0 uuencode.txt\01215VAA="!I<R!T:&4@9&%T93\\`\012`\012end\012

(1 row)

edb=# SELECT UTL\_ENCODE.UUDECODE

edb-# ('begin 0 uuencode.txt\01215VAA="!I<R!T:&4@9&%T93\\`\012`\012end\012')

edb-# FROM DUAL;

uudecode

-------------------

What is the date?

(1 row)

## UTL\_FILE

The UTL\_FILE package provides the capability to read from, and write to files on the operating system’s file system. Non-superusers must be granted EXECUTE privilege on the UTL\_FILE package by a superuser before using any of the functions or procedures in the package. For example the following command grants the privilege to user mary:

GRANT EXECUTE ON PACKAGE SYS.UTL\_FILE TO mary;

Also, the operating system username, enterprisedb, must have the appropriate read and/or write permissions on the directories and files to be accessed using the UTL\_FILE functions and procedures. If the required file permissions are not in place, an exception is thrown in the UTL\_FILE function or procedure.

A handle to the file to be written to, or read from is used to reference the file. The file handle is defined by a public variable in the UTL\_FILE package named, UTL\_FILE.FILE\_TYPE. A variable of type FILE\_TYPE must be declared to receive the file handle returned by calling the FOPEN function. The file handle is then used for all subsequent operations on the file.

References to directories on the file system are done using the directory name or alias that is assigned to the directory using the CREATE DIRECTORY command. The procedures and functions available in the UTL\_FILE package are listed in the following table.

Table 7‑9‑18 UTL\_FILE Functions/Procedures

| Function/Procedure | Return Type | Description |
| --- | --- | --- |
| FCLOSE(file IN OUT) | n/a | Closes the specified file identified by file. |
| FCLOSE\_ALL | n/a | Closes all open files. |
| FCOPY(location, filename, dest\_dir, dest\_file [, start\_line [, end\_line ] ]) | n/a | Copies filename in the directory identified by location to file, dest\_file, in directory, dest\_dir, starting from line, start\_line, to line, end\_line. |
| FFLUSH(file) | n/a | Forces data in the buffer to be written to disk in the file identified by file. |
| FOPEN(location, filename, open\_mode [, max\_linesize ]) | FILE\_TYPE | Opens file, filename, in the directory identified by location. |
| FREMOVE(location, filename) | n/a | Removes the specified file from the file system. |
| FRENAME(location, filename, dest\_dir, dest\_file [, overwrite ]) | n/a | Renames the specified file. |
| GET\_LINE(file, buffer OUT) | n/a | Reads a line of text into variable, buffer, from the file identified by file. |
| IS\_OPEN(file) | BOOLEAN | Determines whether or not the given file is open. |
| NEW\_LINE(file [, lines ]) | n/a | Writes an end-of-line character sequence into the file. |
| PUT(file, buffer) | n/a | Writes buffer to the given file. PUT does not write an end-of-line character sequence. |
| PUT\_LINE(file, buffer) | n/a | Writes buffer to the given file. An end-of-line character sequence is added by the PUT\_LINE procedure. |
| PUTF(file, format [, arg1 ] [, ...]) | n/a | Writes a formatted string to the given file. Up to five substitution parameters, arg1,...arg5 may be specified for replacement in format. |

UTL\_FILE Exception Codes

The UTL\_FILE package reports the following exception codes:

|  |  |
| --- | --- |
| Exception Code | Condition name |
| -29283 | invalid\_operation |
| -29285 | write\_error |
| -29284 | read\_error |
| -29282 | invalid\_filehandle |
| -29287 | invalid\_maxlinesize |
| -29281 | invalid\_mode |
| -29280 | invalid\_path |

### Setting File Permissions with utl\_file.umask

When a UTL\_FILE function or procedure creates a file, there are default file permissions as shown by the following.

-rw------- 1 enterprisedb enterprisedb 21 Jul 24 16:08 utlfile

Note that all permissions are denied on users belonging to the enterprisedb group as well as all other users. Only the enterprisedb user has read and write permissions on the created file.

If you wish to have a different set of file permissions on files created by the UTL\_FILE functions and procedures, you can accomplish this by setting the utl\_file.umask configuration parameter.

The utl\_file.umask parameter sets the file mode creation mask or simply, the mask, in a manner similar to the Linux umask command. This is for usage only within the Advanced Server UTL\_FILE package.

Note: The utl\_file.umask parameter is not supported on Windows systems.

The value specified for utl\_file.umask is a 3 or 4-character octal string that would be valid for the Linux umask command. The setting determines the permissions on files created by the UTL\_FILE functions and procedures. (Refer to any information source regarding Linux or Unix systems for information on file permissions and the usage of the umask command.)

The following is an example of setting the file permissions with utl\_file.umask.

First, set up the directory in the file system to be used by the UTL\_FILE package. Be sure the operating system account, enterprisedb or postgres, whichever is applicable, can read and write in the directory.

mkdir /tmp/utldir

chmod 777 /tmp/utldir

The CREATE DIRECTORY command is issued in psql to create the directory database object using the file system directory created in the preceding step.

CREATE DIRECTORY utldir AS '/tmp/utldir';

Set the utl\_file.umask configuration parameter. The following setting allows the file owner any permission. Group users and other users are permitted any permission except for the execute permission.

SET utl\_file.umask TO '0011';

In the same session during which the utl\_file.umask parameter is set to the desired value, run the UTL\_FILE functions and procedures.

DECLARE

v\_utlfile UTL\_FILE.FILE\_TYPE;

v\_directory VARCHAR2(50) := 'utldir';

v\_filename VARCHAR2(20) := 'utlfile';

BEGIN

v\_utlfile := UTL\_FILE.FOPEN(v\_directory, v\_filename, 'w');

UTL\_FILE.PUT\_LINE(v\_utlfile, 'Simple one-line file');

DBMS\_OUTPUT.PUT\_LINE('Created file: ' || v\_filename);

UTL\_FILE.FCLOSE(v\_utlfile);

END;

The permission settings on the resulting file show that group users and other users have read and write permissions on the file as well as the file owner.

$ pwd

/tmp/utldir

$ ls -l

total 4

-rw-rw-rw- 1 enterprisedb enterprisedb 21 Jul 24 16:04 utlfile

This parameter can also be set on a per role basis with the ALTER ROLE command, on a per database basis with the ALTER DATABASE command, or for the entire database server instance by setting it in the postgresql.conf file.

### FCLOSE

The FCLOSE procedure closes an open file.

FCLOSE(file IN OUT FILE\_TYPE)

Parameters

file

Variable of type FILE\_TYPE containing a file handle of the file to be closed.

### FCLOSE\_ALL

The FLCLOSE\_ALL procedures closes all open files. The procedure executes successfully even if there are no open files to close.

FCLOSE\_ALL

### FCOPY

The FCOPY procedure copies text from one file to another.

FCOPY(location VARCHAR2, filename VARCHAR2,

dest\_dir VARCHAR2, dest\_file VARCHAR2

[, start\_line PLS\_INTEGER [, end\_line PLS\_INTEGER ] ])

Parameters

location

Directory name, as stored in pg\_catalog.edb\_dir.dirname, of the directory containing the file to be copied.

filename

Name of the source file to be copied.

dest\_dir

Directory name, as stored in pg\_catalog.edb\_dir.dirname, of the directory to which the file is to be copied.

dest\_file

Name of the destination file.

start\_line

Line number in the source file from which copying will begin. The default is 1.

end\_line

Line number of the last line in the source file to be copied. If omitted or null, copying will go to the last line of the file.

Examples

The following makes a copy of a file, C:\TEMP\EMPDIR\empfile.csv, containing a comma-delimited list of employees from the emp table. The copy, empcopy.csv, is then listed.

CREATE DIRECTORY empdir AS 'C:/TEMP/EMPDIR';

DECLARE

v\_empfile UTL\_FILE.FILE\_TYPE;

v\_src\_dir VARCHAR2(50) := 'empdir';

v\_src\_file VARCHAR2(20) := 'empfile.csv';

v\_dest\_dir VARCHAR2(50) := 'empdir';

v\_dest\_file VARCHAR2(20) := 'empcopy.csv';

v\_emprec VARCHAR2(120);

v\_count INTEGER := 0;

BEGIN

UTL\_FILE.FCOPY(v\_src\_dir,v\_src\_file,v\_dest\_dir,v\_dest\_file);

v\_empfile := UTL\_FILE.FOPEN(v\_dest\_dir,v\_dest\_file,'r');

DBMS\_OUTPUT.PUT\_LINE('The following is the destination file, ''' ||

v\_dest\_file || '''');

LOOP

UTL\_FILE.GET\_LINE(v\_empfile,v\_emprec);

DBMS\_OUTPUT.PUT\_LINE(v\_emprec);

v\_count := v\_count + 1;

END LOOP;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

UTL\_FILE.FCLOSE(v\_empfile);

DBMS\_OUTPUT.PUT\_LINE(v\_count || ' records retrieved');

WHEN OTHERS THEN

DBMS\_OUTPUT.PUT\_LINE('SQLERRM: ' || SQLERRM);

DBMS\_OUTPUT.PUT\_LINE('SQLCODE: ' || SQLCODE);

END;

The following is the destination file, 'empcopy.csv'

7369,SMITH,CLERK,7902,17-DEC-80,800,,20

7499,ALLEN,SALESMAN,7698,20-FEB-81,1600,300,30

7521,WARD,SALESMAN,7698,22-FEB-81,1250,500,30

7566,JONES,MANAGER,7839,02-APR-81,2975,,20

7654,MARTIN,SALESMAN,7698,28-SEP-81,1250,1400,30

7698,BLAKE,MANAGER,7839,01-MAY-81,2850,,30

7782,CLARK,MANAGER,7839,09-JUN-81,2450,,10

7788,SCOTT,ANALYST,7566,19-APR-87,3000,,20

7839,KING,PRESIDENT,,17-NOV-81,5000,,10

7844,TURNER,SALESMAN,7698,08-SEP-81,1500,0,30

7876,ADAMS,CLERK,7788,23-MAY-87,1100,,20

7900,JAMES,CLERK,7698,03-DEC-81,950,,30

7902,FORD,ANALYST,7566,03-DEC-81,3000,,20

7934,MILLER,CLERK,7782,23-JAN-82,1300,,10

14 records retrieved

### FFLUSH

The FFLUSH procedure flushes unwritten data from the write buffer to the file.

FFLUSH(file FILE\_TYPE)

Parameters

file

Variable of type FILE\_TYPE containing a file handle.

Examples

Each line is flushed after the NEW\_LINE procedure is called.

DECLARE

v\_empfile UTL\_FILE.FILE\_TYPE;

v\_directory VARCHAR2(50) := 'empdir';

v\_filename VARCHAR2(20) := 'empfile.csv';

CURSOR emp\_cur IS SELECT \* FROM emp ORDER BY empno;

BEGIN

v\_empfile := UTL\_FILE.FOPEN(v\_directory,v\_filename,'w');

FOR i IN emp\_cur LOOP

UTL\_FILE.PUT(v\_empfile,i.empno);

UTL\_FILE.PUT(v\_empfile,',');

UTL\_FILE.PUT(v\_empfile,i.ename);

UTL\_FILE.PUT(v\_empfile,',');

UTL\_FILE.PUT(v\_empfile,i.job);

UTL\_FILE.PUT(v\_empfile,',');

UTL\_FILE.PUT(v\_empfile,i.mgr);

UTL\_FILE.PUT(v\_empfile,',');

UTL\_FILE.PUT(v\_empfile,i.hiredate);

UTL\_FILE.PUT(v\_empfile,',');

UTL\_FILE.PUT(v\_empfile,i.sal);

UTL\_FILE.PUT(v\_empfile,',');

UTL\_FILE.PUT(v\_empfile,i.comm);

UTL\_FILE.PUT(v\_empfile,',');

UTL\_FILE.PUT(v\_empfile,i.deptno);

UTL\_FILE.NEW\_LINE(v\_empfile);

UTL\_FILE.FFLUSH(v\_empfile);

END LOOP;

DBMS\_OUTPUT.PUT\_LINE('Created file: ' || v\_filename);

UTL\_FILE.FCLOSE(v\_empfile);

END;

### FOPEN

The FOPEN function opens a file for I/O.

filetype FILE\_TYPE FOPEN(location VARCHAR2, filename VARCHAR2,

open\_mode VARCHAR2 [, max\_linesize BINARY\_INTEGER ])

Parameters

location

Directory name, as stored in pg\_catalog.edb\_dir.dirname, of the directory containing the file to be opened.

filename

Name of the file to be opened.

open\_mode

Mode in which the file will be opened. Modes are: a - append to file; r - read from file; w - write to file.

max\_linesize

Maximum size of a line in characters. In read mode, an exception is thrown if an attempt is made to read a line exceeding max\_linesize. In write and append modes, an exception is thrown if an attempt is made to write a line exceeding max\_linesize. The end-of-line character(s) are not included in determining if the maximum line size is exceeded.

filetype

Variable of type FILE\_TYPE containing the file handle of the opened file.

### FREMOVE

The FREMOVE procedure removes a file from the system.

FREMOVE(location VARCHAR2, filename VARCHAR2)

An exception is thrown if the file to be removed does not exist.

Parameters

location

Directory name, as stored in pg\_catalog.edb\_dir.dirname, of the directory containing the file to be removed.

filename

Name of the file to be removed.

Examples

The following removes file empfile.csv.

DECLARE

v\_directory VARCHAR2(50) := 'empdir';

v\_filename VARCHAR2(20) := 'empfile.csv';

BEGIN

UTL\_FILE.FREMOVE(v\_directory,v\_filename);

DBMS\_OUTPUT.PUT\_LINE('Removed file: ' || v\_filename);

EXCEPTION

WHEN OTHERS THEN

DBMS\_OUTPUT.PUT\_LINE('SQLERRM: ' || SQLERRM);

DBMS\_OUTPUT.PUT\_LINE('SQLCODE: ' || SQLCODE);

END;

Removed file: empfile.csv

### FRENAME

The FRENAME procedure renames a given file. This effectively moves a file from one location to another.

FRENAME(location VARCHAR2, filename VARCHAR2,

dest\_dir VARCHAR2, dest\_file VARCHAR2, [ overwrite BOOLEAN ])

Parameters

location

Directory name, as stored in pg\_catalog.edb\_dir.dirname, of the directory containing the file to be renamed.

filename

Name of the source file to be renamed.

dest\_dir

Directory name, as stored in pg\_catalog.edb\_dir.dirname, of the directory to which the renamed file is to exist.

dest\_file

New name of the original file.

overwrite

Replaces any existing file named dest\_file in dest\_dir if set to TRUE, otherwise an exception is thrown if set to FALSE. This is the default.

Examples

The following renames a file, C:\TEMP\EMPDIR\empfile.csv, containing a comma-delimited list of employees from the emp table. The renamed file, C:\TEMP\NEWDIR\newemp.csv, is then listed.

CREATE DIRECTORY "newdir" AS 'C:/TEMP/NEWDIR';

DECLARE

v\_empfile UTL\_FILE.FILE\_TYPE;

v\_src\_dir VARCHAR2(50) := 'empdir';

v\_src\_file VARCHAR2(20) := 'empfile.csv';

v\_dest\_dir VARCHAR2(50) := 'newdir';

v\_dest\_file VARCHAR2(50) := 'newemp.csv';

v\_replace BOOLEAN := FALSE;

v\_emprec VARCHAR2(120);

v\_count INTEGER := 0;

BEGIN

UTL\_FILE.FRENAME(v\_src\_dir,v\_src\_file,v\_dest\_dir,

v\_dest\_file,v\_replace);

v\_empfile := UTL\_FILE.FOPEN(v\_dest\_dir,v\_dest\_file,'r');

DBMS\_OUTPUT.PUT\_LINE('The following is the renamed file, ''' ||

v\_dest\_file || '''');

LOOP

UTL\_FILE.GET\_LINE(v\_empfile,v\_emprec);

DBMS\_OUTPUT.PUT\_LINE(v\_emprec);

v\_count := v\_count + 1;

END LOOP;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

UTL\_FILE.FCLOSE(v\_empfile);

DBMS\_OUTPUT.PUT\_LINE(v\_count || ' records retrieved');

WHEN OTHERS THEN

DBMS\_OUTPUT.PUT\_LINE('SQLERRM: ' || SQLERRM);

DBMS\_OUTPUT.PUT\_LINE('SQLCODE: ' || SQLCODE);

END;

The following is the renamed file, 'newemp.csv'

7369,SMITH,CLERK,7902,17-DEC-80 00:00:00,800.00,,20

7499,ALLEN,SALESMAN,7698,20-FEB-81 00:00:00,1600.00,300.00,30

7521,WARD,SALESMAN,7698,22-FEB-81 00:00:00,1250.00,500.00,30

7566,JONES,MANAGER,7839,02-APR-81 00:00:00,2975.00,,20

7654,MARTIN,SALESMAN,7698,28-SEP-81 00:00:00,1250.00,1400.00,30

7698,BLAKE,MANAGER,7839,01-MAY-81 00:00:00,2850.00,,30

7782,CLARK,MANAGER,7839,09-JUN-81 00:00:00,2450.00,,10

7788,SCOTT,ANALYST,7566,19-APR-87 00:00:00,3000.00,,20

7839,KING,PRESIDENT,,17-NOV-81 00:00:00,5000.00,,10

7844,TURNER,SALESMAN,7698,08-SEP-81 00:00:00,1500.00,0.00,30

7876,ADAMS,CLERK,7788,23-MAY-87 00:00:00,1100.00,,20

7900,JAMES,CLERK,7698,03-DEC-81 00:00:00,950.00,,30

7902,FORD,ANALYST,7566,03-DEC-81 00:00:00,3000.00,,20

7934,MILLER,CLERK,7782,23-JAN-82 00:00:00,1300.00,,10

14 records retrieved

### GET\_LINE

The GET\_LINE procedure reads a line of text from a given file up to, but not including the end-of-line terminator. A NO\_DATA\_FOUND exception is thrown when there are no more lines to read.

GET\_LINE(file FILE\_TYPE, buffer OUT VARCHAR2)

Parameters

file

Variable of type FILE\_TYPE containing the file handle of the opened file.

buffer

Variable to receive a line from the file.

Examples

The following anonymous block reads through and displays the records in file empfile.csv.

DECLARE

v\_empfile UTL\_FILE.FILE\_TYPE;

v\_directory VARCHAR2(50) := 'empdir';

v\_filename VARCHAR2(20) := 'empfile.csv';

v\_emprec VARCHAR2(120);

v\_count INTEGER := 0;

BEGIN

v\_empfile := UTL\_FILE.FOPEN(v\_directory,v\_filename,'r');

LOOP

UTL\_FILE.GET\_LINE(v\_empfile,v\_emprec);

DBMS\_OUTPUT.PUT\_LINE(v\_emprec);

v\_count := v\_count + 1;

END LOOP;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

UTL\_FILE.FCLOSE(v\_empfile);

DBMS\_OUTPUT.PUT\_LINE('End of file ' || v\_filename || ' - ' ||

v\_count || ' records retrieved');

WHEN OTHERS THEN

DBMS\_OUTPUT.PUT\_LINE('SQLERRM: ' || SQLERRM);

DBMS\_OUTPUT.PUT\_LINE('SQLCODE: ' || SQLCODE);

END;

7369,SMITH,CLERK,7902,17-DEC-80 00:00:00,800.00,,20

7499,ALLEN,SALESMAN,7698,20-FEB-81 00:00:00,1600.00,300.00,30

7521,WARD,SALESMAN,7698,22-FEB-81 00:00:00,1250.00,500.00,30

7566,JONES,MANAGER,7839,02-APR-81 00:00:00,2975.00,,20

7654,MARTIN,SALESMAN,7698,28-SEP-81 00:00:00,1250.00,1400.00,30

7698,BLAKE,MANAGER,7839,01-MAY-81 00:00:00,2850.00,,30

7782,CLARK,MANAGER,7839,09-JUN-81 00:00:00,2450.00,,10

7788,SCOTT,ANALYST,7566,19-APR-87 00:00:00,3000.00,,20

7839,KING,PRESIDENT,,17-NOV-81 00:00:00,5000.00,,10

7844,TURNER,SALESMAN,7698,08-SEP-81 00:00:00,1500.00,0.00,30

7876,ADAMS,CLERK,7788,23-MAY-87 00:00:00,1100.00,,20

7900,JAMES,CLERK,7698,03-DEC-81 00:00:00,950.00,,30

7902,FORD,ANALYST,7566,03-DEC-81 00:00:00,3000.00,,20

7934,MILLER,CLERK,7782,23-JAN-82 00:00:00,1300.00,,10

End of file empfile.csv - 14 records retrieved

### IS\_OPEN

The IS\_OPEN function determines whether or not the given file is open.

status BOOLEAN IS\_OPEN(file FILE\_TYPE)

Parameters

file

Variable of type FILE\_TYPE containing the file handle of the file to be tested.

status

TRUE if the given file is open, FALSE otherwise.

### NEW\_LINE

The NEW\_LINE procedure writes an end-of-line character sequence in the file.

NEW\_LINE(file FILE\_TYPE [, lines INTEGER ])

Parameters

file

Variable of type FILE\_TYPE containing the file handle of the file to which end-of-line character sequences are to be written.

lines

Number of end-of-line character sequences to be written. The default is one.

Examples

A file containing a double-spaced list of employee records is written.

DECLARE

v\_empfile UTL\_FILE.FILE\_TYPE;

v\_directory VARCHAR2(50) := 'empdir';

v\_filename VARCHAR2(20) := 'empfile.csv';

CURSOR emp\_cur IS SELECT \* FROM emp ORDER BY empno;

BEGIN

v\_empfile := UTL\_FILE.FOPEN(v\_directory,v\_filename,'w');

FOR i IN emp\_cur LOOP

UTL\_FILE.PUT(v\_empfile,i.empno);

UTL\_FILE.PUT(v\_empfile,',');

UTL\_FILE.PUT(v\_empfile,i.ename);

UTL\_FILE.PUT(v\_empfile,',');

UTL\_FILE.PUT(v\_empfile,i.job);

UTL\_FILE.PUT(v\_empfile,',');

UTL\_FILE.PUT(v\_empfile,i.mgr);

UTL\_FILE.PUT(v\_empfile,',');

UTL\_FILE.PUT(v\_empfile,i.hiredate);

UTL\_FILE.PUT(v\_empfile,',');

UTL\_FILE.PUT(v\_empfile,i.sal);

UTL\_FILE.PUT(v\_empfile,',');

UTL\_FILE.PUT(v\_empfile,i.comm);

UTL\_FILE.PUT(v\_empfile,',');

UTL\_FILE.PUT(v\_empfile,i.deptno);

UTL\_FILE.NEW\_LINE(v\_empfile,2);

END LOOP;

DBMS\_OUTPUT.PUT\_LINE('Created file: ' || v\_filename);

UTL\_FILE.FCLOSE(v\_empfile);

END;

Created file: empfile.csv

This file is then displayed:

C:\TEMP\EMPDIR>TYPE empfile.csv

7369,SMITH,CLERK,7902,17-DEC-80 00:00:00,800.00,,20

7499,ALLEN,SALESMAN,7698,20-FEB-81 00:00:00,1600.00,300.00,30

7521,WARD,SALESMAN,7698,22-FEB-81 00:00:00,1250.00,500.00,30

7566,JONES,MANAGER,7839,02-APR-81 00:00:00,2975.00,,20

7654,MARTIN,SALESMAN,7698,28-SEP-81 00:00:00,1250.00,1400.00,30

7698,BLAKE,MANAGER,7839,01-MAY-81 00:00:00,2850.00,,30

7782,CLARK,MANAGER,7839,09-JUN-81 00:00:00,2450.00,,10

7788,SCOTT,ANALYST,7566,19-APR-87 00:00:00,3000.00,,20

7839,KING,PRESIDENT,,17-NOV-81 00:00:00,5000.00,,10

7844,TURNER,SALESMAN,7698,08-SEP-81 00:00:00,1500.00,0.00,30

7876,ADAMS,CLERK,7788,23-MAY-87 00:00:00,1100.00,,20

7900,JAMES,CLERK,7698,03-DEC-81 00:00:00,950.00,,30

7902,FORD,ANALYST,7566,03-DEC-81 00:00:00,3000.00,,20

7934,MILLER,CLERK,7782,23-JAN-82 00:00:00,1300.00,,10

### PUT

The PUT procedure writes a string to the given file. No end-of-line character sequence is written at the end of the string. Use the NEW\_LINE procedure to add an end-of-line character sequence.

PUT(file FILE\_TYPE, buffer { DATE | NUMBER | TIMESTAMP |

VARCHAR2 })

Parameters

file

Variable of type FILE\_TYPE containing the file handle of the file to which the given string is to be written.

buffer

Text to be written to the specified file.

Examples

The following example uses the PUT procedure to create a comma-delimited file of employees from the emp table.

DECLARE

v\_empfile UTL\_FILE.FILE\_TYPE;

v\_directory VARCHAR2(50) := 'empdir';

v\_filename VARCHAR2(20) := 'empfile.csv';

CURSOR emp\_cur IS SELECT \* FROM emp ORDER BY empno;

BEGIN

v\_empfile := UTL\_FILE.FOPEN(v\_directory,v\_filename,'w');

FOR i IN emp\_cur LOOP

UTL\_FILE.PUT(v\_empfile,i.empno);

UTL\_FILE.PUT(v\_empfile,',');

UTL\_FILE.PUT(v\_empfile,i.ename);

UTL\_FILE.PUT(v\_empfile,',');

UTL\_FILE.PUT(v\_empfile,i.job);

UTL\_FILE.PUT(v\_empfile,',');

UTL\_FILE.PUT(v\_empfile,i.mgr);

UTL\_FILE.PUT(v\_empfile,',');

UTL\_FILE.PUT(v\_empfile,i.hiredate);

UTL\_FILE.PUT(v\_empfile,',');

UTL\_FILE.PUT(v\_empfile,i.sal);

UTL\_FILE.PUT(v\_empfile,',');

UTL\_FILE.PUT(v\_empfile,i.comm);

UTL\_FILE.PUT(v\_empfile,',');

UTL\_FILE.PUT(v\_empfile,i.deptno);

UTL\_FILE.NEW\_LINE(v\_empfile);

END LOOP;

DBMS\_OUTPUT.PUT\_LINE('Created file: ' || v\_filename);

UTL\_FILE.FCLOSE(v\_empfile);

END;

Created file: empfile.csv

The following is the contents of empfile.csv created above:

C:\TEMP\EMPDIR>TYPE empfile.csv

7369,SMITH,CLERK,7902,17-DEC-80 00:00:00,800.00,,20

7499,ALLEN,SALESMAN,7698,20-FEB-81 00:00:00,1600.00,300.00,30

7521,WARD,SALESMAN,7698,22-FEB-81 00:00:00,1250.00,500.00,30

7566,JONES,MANAGER,7839,02-APR-81 00:00:00,2975.00,,20

7654,MARTIN,SALESMAN,7698,28-SEP-81 00:00:00,1250.00,1400.00,30

7698,BLAKE,MANAGER,7839,01-MAY-81 00:00:00,2850.00,,30

7782,CLARK,MANAGER,7839,09-JUN-81 00:00:00,2450.00,,10

7788,SCOTT,ANALYST,7566,19-APR-87 00:00:00,3000.00,,20

7839,KING,PRESIDENT,,17-NOV-81 00:00:00,5000.00,,10

7844,TURNER,SALESMAN,7698,08-SEP-81 00:00:00,1500.00,0.00,30

7876,ADAMS,CLERK,7788,23-MAY-87 00:00:00,1100.00,,20

7900,JAMES,CLERK,7698,03-DEC-81 00:00:00,950.00,,30

7902,FORD,ANALYST,7566,03-DEC-81 00:00:00,3000.00,,20

7934,MILLER,CLERK,7782,23-JAN-82 00:00:00,1300.00,,10

### PUT\_LINE

The PUT\_LINE procedure writes a single line to the given file including an end-of-line character sequence.

PUT\_LINE(file FILE\_TYPE, buffer { DATE | NUMBER | TIMESTAMP |

VARCHAR2 })

Parameters

file

Variable of type FILE\_TYPE containing the file handle of the file to which the given line is to be written.

buffer

Text to be written to the specified file.

Examples

The following example uses the PUT\_LINE procedure to create a comma-delimited file of employees from the emp table.

DECLARE

v\_empfile UTL\_FILE.FILE\_TYPE;

v\_directory VARCHAR2(50) := 'empdir';

v\_filename VARCHAR2(20) := 'empfile.csv';

v\_emprec VARCHAR2(120);

CURSOR emp\_cur IS SELECT \* FROM emp ORDER BY empno;

BEGIN

v\_empfile := UTL\_FILE.FOPEN(v\_directory,v\_filename,'w');

FOR i IN emp\_cur LOOP

v\_emprec := i.empno || ',' || i.ename || ',' || i.job || ',' ||

NVL(LTRIM(TO\_CHAR(i.mgr,'9999')),'') || ',' || i.hiredate ||

',' || i.sal || ',' ||

NVL(LTRIM(TO\_CHAR(i.comm,'9990.99')),'') || ',' || i.deptno;

UTL\_FILE.PUT\_LINE(v\_empfile,v\_emprec);

END LOOP;

DBMS\_OUTPUT.PUT\_LINE('Created file: ' || v\_filename);

UTL\_FILE.FCLOSE(v\_empfile);

END;

The following is the contents of empfile.csv created above:

C:\TEMP\EMPDIR>TYPE empfile.csv

7369,SMITH,CLERK,7902,17-DEC-80 00:00:00,800.00,,20

7499,ALLEN,SALESMAN,7698,20-FEB-81 00:00:00,1600.00,300.00,30

7521,WARD,SALESMAN,7698,22-FEB-81 00:00:00,1250.00,500.00,30

7566,JONES,MANAGER,7839,02-APR-81 00:00:00,2975.00,,20

7654,MARTIN,SALESMAN,7698,28-SEP-81 00:00:00,1250.00,1400.00,30

7698,BLAKE,MANAGER,7839,01-MAY-81 00:00:00,2850.00,,30

7782,CLARK,MANAGER,7839,09-JUN-81 00:00:00,2450.00,,10

7788,SCOTT,ANALYST,7566,19-APR-87 00:00:00,3000.00,,20

7839,KING,PRESIDENT,,17-NOV-81 00:00:00,5000.00,,10

7844,TURNER,SALESMAN,7698,08-SEP-81 00:00:00,1500.00,0.00,30

7876,ADAMS,CLERK,7788,23-MAY-87 00:00:00,1100.00,,20

7900,JAMES,CLERK,7698,03-DEC-81 00:00:00,950.00,,30

7902,FORD,ANALYST,7566,03-DEC-81 00:00:00,3000.00,,20

7934,MILLER,CLERK,7782,23-JAN-82 00:00:00,1300.00,,10

### PUTF

The PUTF procedure writes a formatted string to the given file.

PUTF(file FILE\_TYPE, format VARCHAR2 [, arg1 VARCHAR2]

[, ...])

Parameters

file

Variable of type FILE\_TYPE containing the file handle of the file to which the formatted line is to be written.

format

String to format the text written to the file. The special character sequence, %s, is substituted by the value of arg. The special character sequence, \n, indicates a new line. Note, however, in Advanced Server, a new line character must be specified with two consecutive backslashes instead of one - \\n.

arg1

Up to five arguments, arg1,...arg5, to be substituted in the format string for each occurrence of %s. The first arg is substituted for the first occurrence of %s, the second arg is substituted for the second occurrence of %s, etc.

Examples

The following anonymous block produces formatted output containing data from the emp table. Note the use of the E literal syntax and double backslashes for the new line character sequence in the format string.

DECLARE

v\_empfile UTL\_FILE.FILE\_TYPE;

v\_directory VARCHAR2(50) := 'empdir';

v\_filename VARCHAR2(20) := 'empfile.csv';

v\_format VARCHAR2(200);

CURSOR emp\_cur IS SELECT \* FROM emp ORDER BY empno;

BEGIN

v\_format := E'%s %s, %s\\nSalary: $%s Commission: $%s\\n\\n';

v\_empfile := UTL\_FILE.FOPEN(v\_directory,v\_filename,'w');

FOR i IN emp\_cur LOOP

UTL\_FILE.PUTF(v\_empfile,v\_format,i.empno,i.ename,i.job,i.sal,

NVL(i.comm,0));

END LOOP;

DBMS\_OUTPUT.PUT\_LINE('Created file: ' || v\_filename);

UTL\_FILE.FCLOSE(v\_empfile);

EXCEPTION

WHEN OTHERS THEN

DBMS\_OUTPUT.PUT\_LINE('SQLERRM: ' || SQLERRM);

DBMS\_OUTPUT.PUT\_LINE('SQLCODE: ' || SQLCODE);

END;

Created file: empfile.csv

The following is the contents of empfile.csv created above:

C:\TEMP\EMPDIR>TYPE empfile.csv

7369 SMITH, CLERK

Salary: $800.00 Commission: $0

7499 ALLEN, SALESMAN

Salary: $1600.00 Commission: $300.00

7521 WARD, SALESMAN

Salary: $1250.00 Commission: $500.00

7566 JONES, MANAGER

Salary: $2975.00 Commission: $0

7654 MARTIN, SALESMAN

Salary: $1250.00 Commission: $1400.00

7698 BLAKE, MANAGER

Salary: $2850.00 Commission: $0

7782 CLARK, MANAGER

Salary: $2450.00 Commission: $0

7788 SCOTT, ANALYST

Salary: $3000.00 Commission: $0

7839 KING, PRESIDENT

Salary: $5000.00 Commission: $0

7844 TURNER, SALESMAN

Salary: $1500.00 Commission: $0.00

7876 ADAMS, CLERK

Salary: $1100.00 Commission: $0

7900 JAMES, CLERK

Salary: $950.00 Commission: $0

7902 FORD, ANALYST

Salary: $3000.00 Commission: $0

7934 MILLER, CLERK

Salary: $1300.00 Commission: $0

## UTL\_HTTP

The UTL\_HTTP package provides a way to use the HTTP or HTTPS protocol to retrieve information found at an URL.

Table 7.7.2 UTL\_HTTP Functions and Procedures

| Function/Procedure | Return Type | Description |
| --- | --- | --- |
| BEGIN\_REQUEST(url, method, http\_version) | UTL\_HTTP.REQ | Initiates a new HTTP request. |
| END\_REQUEST(r IN OUT) | n/a | Ends an HTTP request before allowing it to complete. |
| END\_RESPONSE(r IN OUT) | n/a | Ends the HTTP response. |
| GET\_BODY\_CHARSET | VARCHAR2 | Returns the default character set of the body of future HTTP requests. |
| GET\_BODY\_CHARSET(charset OUT) | n/a | Returns the default character set of the body of future HTTP requests. |
| GET\_FOLLOW\_REDIRECT(max\_redirects OUT) | n/a | Current setting for the maximum number of redirections allowed. |
| GET\_HEADER(r IN OUT, n, name OUT, value OUT) | n/a | Returns the nth header of the HTTP response. |
| GET\_HEADER\_BY\_NAME(r IN OUT, name, value OUT, n) | n/a | Returns the HTTP response header for the specified name. |
| GET\_HEADER\_COUNT(r IN OUT) | INTEGER | Returns the number of HTTP response headers. |
| GET\_RESPONSE(r IN OUT) | UTL\_HTTP.RESP | Returns the HTTP response. |
| GET\_RESPONSE\_ERROR\_CHECK(enable OUT) | n/a | Returns whether or not response error check is set. |
| GET\_TRANSFER\_TIMEOUT(timeout OUT) | n/a | Returns the transfer timeout setting for HTTP requests. |
| READ\_LINE(r IN OUT, data OUT, remove\_crlf) | n/a | Returns the HTTP response body in text form until the end of line. |
| READ\_RAW(r IN OUT, data OUT, len) | n/a | Returns the HTTP response body in binary form for a specified number of bytes. |
| READ\_TEXT(r IN OUT, data OUT, len) | n/a | Returns the HTTP response body in text form for a specified number of characters. |
| REQUEST(url) | VARCHAR2 | Returns the content of a web page. |
| REQUEST\_PIECES(url, max\_pieces) | UTL\_HTTP. HTML\_PIECES | Returns a table of 2000-byte segments retrieved from an URL. |
| SET\_BODY\_CHARSET(charset) | n/a | Sets the default character set of the body of future HTTP requests. |
| SET\_FOLLOW\_REDIRECT(max\_redirects) | n/a | Sets the maximum number of times to follow the redirect instruction. |
| SET\_FOLLOW\_REDIRECT(r IN OUT, max\_redirects) | n/a | Sets the maximum number of times to follow the redirect instruction for an individual request. |
| SET\_HEADER(r IN OUT, name, value) | n/a | Sets the HTTP request header. |
| SET\_RESPONSE\_ERROR\_CHECK(enable) | n/a | Determines whether or not HTTP 4xx and 5xx status codes are to be treated as errors. |
| SET\_TRANSFER\_TIMEOUT(timeout) | n/a | Sets the default, transfer timeout value for HTTP requests. |
| SET\_TRANSFER\_TIMEOUT(r IN OUT, timeout) | n/a | Sets the transfer timeout value for an individual HTTP request. |
| WRITE\_LINE(*r* IN OUT, *data*) | n/a | Writes CRLF terminated data to the HTTP request body in TEXT form. |
| WRITE\_RAW(r IN OUT, data) | n/a | Writes data to the HTTP request body in BINARY form. |
| WRITE\_TEXT(*r* IN OUT, *data*) | n/a | Writes data to the HTTP request body in TEXT form. |

Advanced Server's implementation of UTL\_HTTP is a partial implementation when compared to Oracle's version. Only those functions and procedures listed in the table above are supported.

Please Note:

In Advanced Server, an HTTP 4xx or HTTP 5xx response produces a database error; in Oracle, this is configurable but FALSE by default.

In Advanced Server, the UTL\_HTTP text interfaces expect the downloaded data to be in the database encoding. All currently-available interfaces are text interfaces. In Oracle, the encoding is detected from HTTP headers; in the absence of the header, the default is configurable and defaults to ISO-8859-1.

Advanced Server ignores all cookies it receives.

The UTL\_HTTP exceptions that can be raised in Oracle are not recognized by Advanced Server. In addition, the error codes returned by Advanced Server are not the same as those returned by Oracle.

There are various public constants available with UTL\_HTTP. These are listed in the following tables.

The following table contains UTL\_HTTP public constants defining HTTP versions and port assignments.

|  |  |
| --- | --- |
| HTTP VERSIONS | |
| HTTP\_VERSION\_1\_0 | CONSTANT VARCHAR2(64) := 'HTTP/1.0'; |
| HTTP\_VERSION\_1\_1 | CONSTANT VARCHAR2(64) := 'HTTP/1.1'; |
| STANDARD PORT ASSIGNMENTS | |
| DEFAULT\_HTTP\_PORT | CONSTANT INTEGER := 80; |
| DEFAULT\_HTTPS\_PORT | CONSTANT INTEGER := 443; |

The following table contains UTL\_HTTP public status code constants.

|  |  |
| --- | --- |
| 1XX INFORMATIONAL | |
| HTTP\_CONTINUE | CONSTANT INTEGER := 100; |
| HTTP\_SWITCHING\_PROTOCOLS | CONSTANT INTEGER := 101; |
| HTTP\_PROCESSING | CONSTANT INTEGER := 102; |
| 2XX SUCCESS | |
| HTTP\_OK | CONSTANT INTEGER := 200; |
| HTTP\_CREATED | CONSTANT INTEGER := 201; |
| HTTP\_ACCEPTED | CONSTANT INTEGER := 202; |
| HTTP\_NON\_AUTHORITATIVE\_INFO | CONSTANT INTEGER := 203; |
| HTTP\_NO\_CONTENT | CONSTANT INTEGER := 204; |
| HTTP\_RESET\_CONTENT | CONSTANT INTEGER := 205; |
| HTTP\_PARTIAL\_CONTENT | CONSTANT INTEGER := 206; |
| HTTP\_MULTI\_STATUS | CONSTANT INTEGER := 207; |
| HTTP\_ALREADY\_REPORTED | CONSTANT INTEGER := 208; |
| HTTP\_IM\_USED | CONSTANT INTEGER := 226; |
| 3XX REDIRECTION | |
| HTTP\_MULTIPLE\_CHOICES | CONSTANT INTEGER := 300; |
| HTTP\_MOVED\_PERMANENTLY | CONSTANT INTEGER := 301; |
| HTTP\_FOUND | CONSTANT INTEGER := 302; |
| HTTP\_SEE\_OTHER | CONSTANT INTEGER := 303; |
| HTTP\_NOT\_MODIFIED | CONSTANT INTEGER := 304; |
| HTTP\_USE\_PROXY | CONSTANT INTEGER := 305; |
| HTTP\_SWITCH\_PROXY | CONSTANT INTEGER := 306; |
| HTTP\_TEMPORARY\_REDIRECT | CONSTANT INTEGER := 307; |
| HTTP\_PERMANENT\_REDIRECT | CONSTANT INTEGER := 308; |
| 4XX CLIENT ERROR | |
| HTTP\_BAD\_REQUEST | CONSTANT INTEGER := 400; |
| HTTP\_UNAUTHORIZED | CONSTANT INTEGER := 401; |
| HTTP\_PAYMENT\_REQUIRED | CONSTANT INTEGER := 402; |
| HTTP\_FORBIDDEN | CONSTANT INTEGER := 403; |
| HTTP\_NOT\_FOUND | CONSTANT INTEGER := 404; |
| HTTP\_METHOD\_NOT\_ALLOWED | CONSTANT INTEGER := 405; |
| HTTP\_NOT\_ACCEPTABLE | CONSTANT INTEGER := 406; |
| HTTP\_PROXY\_AUTH\_REQUIRED | CONSTANT INTEGER := 407; |
| HTTP\_REQUEST\_TIME\_OUT | CONSTANT INTEGER := 408; |
| HTTP\_CONFLICT | CONSTANT INTEGER := 409; |
| HTTP\_GONE | CONSTANT INTEGER := 410; |
| HTTP\_LENGTH\_REQUIRED | CONSTANT INTEGER := 411; |
| HTTP\_PRECONDITION\_FAILED | CONSTANT INTEGER := 412; |
| HTTP\_REQUEST\_ENTITY\_TOO\_LARGE | CONSTANT INTEGER := 413; |
| HTTP\_REQUEST\_URI\_TOO\_LARGE | CONSTANT INTEGER := 414; |
| HTTP\_UNSUPPORTED\_MEDIA\_TYPE | CONSTANT INTEGER := 415; |
| HTTP\_REQ\_RANGE\_NOT\_SATISFIABLE | CONSTANT INTEGER := 416; |
| HTTP\_EXPECTATION\_FAILED | CONSTANT INTEGER := 417; |
| HTTP\_I\_AM\_A\_TEAPOT | CONSTANT INTEGER := 418; |
| HTTP\_AUTHENTICATION\_TIME\_OUT | CONSTANT INTEGER := 419; |
| HTTP\_ENHANCE\_YOUR\_CALM | CONSTANT INTEGER := 420; |
| HTTP\_UNPROCESSABLE\_ENTITY | CONSTANT INTEGER := 422; |
| HTTP\_LOCKED | CONSTANT INTEGER := 423; |
| HTTP\_FAILED\_DEPENDENCY | CONSTANT INTEGER := 424; |
| HTTP\_UNORDERED\_COLLECTION | CONSTANT INTEGER := 425; |
| HTTP\_UPGRADE\_REQUIRED | CONSTANT INTEGER := 426; |
| HTTP\_PRECONDITION\_REQUIRED | CONSTANT INTEGER := 428; |
| HTTP\_TOO\_MANY\_REQUESTS | CONSTANT INTEGER := 429; |
| HTTP\_REQUEST\_HEADER\_FIELDS\_TOO\_LARGE | CONSTANT INTEGER := 431; |
| HTTP\_NO\_RESPONSE | CONSTANT INTEGER := 444; |
| HTTP\_RETRY\_WITH | CONSTANT INTEGER := 449; |
| HTTP\_BLOCKED\_BY\_WINDOWS\_PARENTAL\_CONTROLS | CONSTANT INTEGER := 450; |
| HTTP\_REDIRECT | CONSTANT INTEGER := 451; |
| HTTP\_REQUEST\_HEADER\_TOO\_LARGE | CONSTANT INTEGER := 494; |
| HTTP\_CERT\_ERROR | CONSTANT INTEGER := 495; |
| HTTP\_NO\_CERT | CONSTANT INTEGER := 496; |
| HTTP\_HTTP\_TO\_HTTPS | CONSTANT INTEGER := 497; |
| HTTP\_CLIENT\_CLOSED\_REQUEST | CONSTANT INTEGER := 499; |
| 5XX SERVER ERROR | |
| HTTP\_INTERNAL\_SERVER\_ERROR | CONSTANT INTEGER := 500; |
| HTTP\_NOT\_IMPLEMENTED | CONSTANT INTEGER := 501; |
| HTTP\_BAD\_GATEWAY | CONSTANT INTEGER := 502; |
| HTTP\_SERVICE\_UNAVAILABLE | CONSTANT INTEGER := 503; |
| HTTP\_GATEWAY\_TIME\_OUT | CONSTANT INTEGER := 504; |
| HTTP\_VERSION\_NOT\_SUPPORTED | CONSTANT INTEGER := 505; |
| HTTP\_VARIANT\_ALSO\_NEGOTIATES | CONSTANT INTEGER := 506; |
| HTTP\_INSUFFICIENT\_STORAGE | CONSTANT INTEGER := 507; |
| HTTP\_LOOP\_DETECTED | CONSTANT INTEGER := 508; |
| HTTP\_BANDWIDTH\_LIMIT\_EXCEEDED | CONSTANT INTEGER := 509; |
| HTTP\_NOT\_EXTENDED | CONSTANT INTEGER := 510; |
| HTTP\_NETWORK\_AUTHENTICATION\_REQUIRED | CONSTANT INTEGER := 511; |
| HTTP\_NETWORK\_READ\_TIME\_OUT\_ERROR | CONSTANT INTEGER := 598; |
| HTTP\_NETWORK\_CONNECT\_TIME\_OUT\_ERROR | CONSTANT INTEGER := 599; |

### HTML\_PIECES

The UTL\_HTTP package declares a type named HTML\_PIECES, which is a table of type VARCHAR2 (2000) indexed by BINARY INTEGER. A value of this type is returned by the REQUEST\_PIECES function.

TYPE html\_pieces IS TABLE OF VARCHAR2(2000) INDEX BY BINARY\_INTEGER;

### REQ

The REQ record type holds information about each HTTP request.

TYPE req IS RECORD (

url VARCHAR2(32767), -- URL to be accessed

method VARCHAR2(64), -- HTTP method

http\_version VARCHAR2(64), -- HTTP version

private\_hndl INTEGER -- Holds handle for this request

);

### RESP

The RESP record type holds information about the response from each HTTP request.

TYPE resp IS RECORD (

status\_code INTEGER, -- HTTP status code

reason\_phrase VARCHAR2(256), -- HTTP response reason phrase

http\_version VARCHAR2(64), -- HTTP version

private\_hndl INTEGER -- Holds handle for this response

);

### BEGIN\_REQUEST

The BEGIN\_REQUEST function initiates a new HTTP request. A network connection is established to the web server with the specified URL. The signature is:

BEGIN\_REQUEST(url IN VARCHAR2, method IN VARCHAR2 DEFAULT 'GET ', http\_version IN VARCHAR2 DEFAULT NULL) RETURN UTL\_HTTP.REQ

The BEGIN\_REQUEST function returns a record of type UTL\_HTTP.REQ.

Parameters

url

url is the Uniform Resource Locator from which UTL\_HTTP will return content.

method

method is the HTTP method to be used. The default is GET.

http\_version

http\_version is the HTTP protocol version sending the request. The specified values should be either HTTP/1.0 or HTTP/1.1. The default is null in which case the latest HTTP protocol version supported by the UTL\_HTTP package is used which is 1.1.

### END\_REQUEST

The END\_REQUEST procedure terminates an HTTP request. Use the END\_REQUEST procedure to terminate an HTTP request without completing it and waiting for the response. The normal process is to begin the request, get the response, then close the response. The signature is:

END\_REQUEST(r IN OUT UTL\_HTTP.REQ)

Parameters

r

r is the HTTP request record.

### END\_RESPONSE

The END\_RESPONSE procedure terminates the HTTP response. The END\_RESPONSE procedure completes the HTTP request and response. This is the normal method to end the request and response process. The signature is:

END\_RESPONSE(r IN OUT UTL\_HTTP.RESP)

Parameters

r

r is the HTTP response record.

### GET\_BODY\_CHARSET

The GET\_BODY\_CHARSET program is available in the form of both a procedure and a function. A call to GET\_BODY\_CHARSET returns the default character set of the body of future HTTP requests.

The procedure signature is:

GET\_BODY\_CHARSET(charset OUT VARCHAR2)

The function signature is:

GET\_BODY\_CHARSET() RETURN VARCHAR2

This function returns a VARCHAR2 value.

Parameters

charset

charset is the character set of the body.

Examples

The following is an example of the GET\_BODY\_CHARSET function.

edb=# SELECT UTL\_HTTP.GET\_BODY\_CHARSET() FROM DUAL;

get\_body\_charset

------------------

ISO-8859-1

(1 row)

### GET\_FOLLOW\_REDIRECT

The GET\_FOLLOW\_REDIRECT procedure returns the current setting for the maximum number of redirections allowed. The signature is:

GET\_FOLLOW\_REDIRECT(max\_redirects OUT INTEGER)

Parameters

max\_redirects

max\_redirects is maximum number of redirections allowed.

### GET\_HEADER

The GET\_HEADER procedure returns the *n*th header of the HTTP response. The signature is:

GET\_HEADER(r IN OUT UTL\_HTTP.RESP, n INTEGER, name OUT VARCHAR2, value OUT VARCHAR2)

Parameters

r

r is the HTTP response record.

n

n is the nth header of the HTTP response record to retrieve.

name

name is the name of the response header.

value

value is the value of the response header.

Examples

The following example retrieves the header count, then the headers.

DECLARE

v\_req UTL\_HTTP.REQ;

v\_resp UTL\_HTTP.RESP;

v\_name VARCHAR2(30);

v\_value VARCHAR2(200);

v\_header\_cnt INTEGER;

BEGIN

-- Initiate request and get response

v\_req := UTL\_HTTP.BEGIN\_REQUEST('www.enterprisedb.com');

v\_resp := UTL\_HTTP.GET\_RESPONSE(v\_req);

-- Get header count

v\_header\_cnt := UTL\_HTTP.GET\_HEADER\_COUNT(v\_resp);

DBMS\_OUTPUT.PUT\_LINE('Header Count: ' || v\_header\_cnt);

-- Get all headers

FOR i IN 1 .. v\_header\_cnt LOOP

UTL\_HTTP.GET\_HEADER(v\_resp, i, v\_name, v\_value);

DBMS\_OUTPUT.PUT\_LINE(v\_name || ': ' || v\_value);

END LOOP;

-- Terminate request

UTL\_HTTP.END\_RESPONSE(v\_resp);

END;

The following is the output from the example.

Header Count: 23

Age: 570

Cache-Control: must-revalidate

Content-Type: text/html; charset=utf-8

Date: Wed, 30 Apr 2014 14:57:52 GMT

ETag: "aab02f2bd2d696eed817ca89ef411dda"

Expires: Sun, 19 Nov 1978 05:00:00 GMT

Last-Modified: Wed, 30 Apr 2014 14:15:49 GMT

RTSS: 1-1307-3

Server: Apache/2.2.3 (Red Hat)

Set-Cookie: SESS2771d0952de2a1a84d322a262e0c173c=jn1u1j1etmdi5gg4lh8hakvs01; expires=Fri, 23-May-2014 18:21:43 GMT; path=/; domain=.enterprisedb.com

Vary: Accept-Encoding

Via: 1.1 varnish

X-EDB-Backend: ec

X-EDB-Cache: HIT

X-EDB-Cache-Address: 10.31.162.212

X-EDB-Cache-Server: ip-10-31-162-212

X-EDB-Cache-TTL: 600.000

X-EDB-Cacheable: MAYBE: The user has a cookie of some sort. Maybe it's double choc-chip!

X-EDB-Do-GZIP: false

X-Powered-By: PHP/5.2.17

X-Varnish: 484508634 484506789

transfer-encoding: chunked

Connection: keep-alive

### GET\_HEADER\_BY\_NAME

The GET\_HEADER\_BY\_NAME procedure returns the header of the HTTP response according to the specified name. The signature is:

GET\_HEADER\_BY\_NAME(r IN OUT UTL\_HTTP.RESP, name VARCHAR2, value OUT VARCHAR2, n INTEGER DEFAULT 1)

Parameters

r

r is the HTTP response record.

name

name is the name of the response header to retrieve.

value

value is the value of the response header.

n

n is the nth header of the HTTP response record to retrieve according to the values specified by name. The default is 1.

Examples

The following example retrieves the header for Content-Type.

DECLARE

v\_req UTL\_HTTP.REQ;

v\_resp UTL\_HTTP.RESP;

v\_name VARCHAR2(30) := 'Content-Type';

v\_value VARCHAR2(200);

BEGIN

v\_req := UTL\_HTTP.BEGIN\_REQUEST('www.enterprisedb.com');

v\_resp := UTL\_HTTP.GET\_RESPONSE(v\_req);

UTL\_HTTP.GET\_HEADER\_BY\_NAME(v\_resp, v\_name, v\_value);

DBMS\_OUTPUT.PUT\_LINE(v\_name || ': ' || v\_value);

UTL\_HTTP.END\_RESPONSE(v\_resp);

END;

Content-Type: text/html; charset=utf-8

### GET\_HEADER\_COUNT

The GET\_HEADER\_COUNT function returns the number of HTTP response headers. The signature is:

GET\_HEADER\_COUNT(r IN OUT UTL\_HTTP.RESP) RETURN INTEGER

This function returns an INTEGER value.

Parameters

r

r is the HTTP response record.

### GET\_RESPONSE

The GET\_RESPONSE function sends the network request and returns any HTTP response. The signature is:

GET\_RESPONSE(r IN OUT UTL\_HTTP.REQ) RETURN UTL\_HTTP.RESP

This function returns a UTL\_HTTP.RESP record.

Parameters

r

r is the HTTP request record.

### GET\_RESPONSE\_ERROR\_CHECK

The GET\_RESPONSE\_ERROR\_CHECK procedure returns whether or not response error check is set. The signature is:

GET\_RESPONSE\_ERROR\_CHECK(enable OUT BOOLEAN)

Parameters

enable

enable returns TRUE if response error check is set, otherwise it returns FALSE.

### GET\_TRANSFER\_TIMEOUT

The GET\_TRANSFER\_TIMEOUT procedure returns the current, default transfer timeout setting for HTTP requests. The signature is:

GET\_TRANSFER\_TIMEOUT(timeout OUT INTEGER)

Parameters

timeout

timeout is the transfer timeout setting in seconds.

### READ\_LINE

The READ\_LINE procedure returns the data from the HTTP response body in text form until the end of line is reached. A CR character, a LF character, a CR LF sequence, or the end of the response body constitutes the end of line. The signature is:

READ\_LINE(r IN OUT UTL\_HTTP.RESP, data OUT VARCHAR2, remove\_crlf BOOLEAN DEFAULT FALSE)

Parameters

r

r is the HTTP response record.

data

data is the response body in text form.

remove\_crlf

Set remove\_crlf to TRUE to remove new line characters, otherwise set to FALSE. The default is FALSE.

Examples

The following example retrieves and displays the body of the specified website.

DECLARE

v\_req UTL\_HTTP.REQ;

v\_resp UTL\_HTTP.RESP;

v\_value VARCHAR2(1024);

BEGIN

v\_req := UTL\_HTTP.BEGIN\_REQUEST('http://www.enterprisedb.com');

v\_resp := UTL\_HTTP.GET\_RESPONSE(v\_req);

LOOP

UTL\_HTTP.READ\_LINE(v\_resp, v\_value, TRUE);

DBMS\_OUTPUT.PUT\_LINE(v\_value);

END LOOP;

EXCEPTION

WHEN OTHERS THEN

UTL\_HTTP.END\_RESPONSE(v\_resp);

END;

The following is the output.

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"

"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en" dir="ltr">

<!-- \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ HEAD \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ -->

<head>

<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />

<title>EnterpriseDB | The Postgres Database Company</title>

<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />

<meta name="keywords" content="postgres, postgresql, postgresql installer, mysql migration, open source database, training, replication" />

<meta name="description" content="The leader in open source database products, services, support, training and expertise based on PostgreSQL. Free downloads, documentation, and tutorials." />

<meta name="abstract" content="The Enterprise PostgreSQL Company" />

<link rel="EditURI" type="application/rsd+xml" title="RSD" href="http://www.enterprisedb.com/blogapi/rsd" />

<link rel="alternate" type="application/rss+xml" title="EnterpriseDB RSS" href="http://www.enterprisedb.com/rss.xml" />

<link rel="shortcut icon" href="/sites/all/themes/edb\_pixelcrayons/favicon.ico" type="image/x-icon" />

<link type="text/css" rel="stylesheet" media="all" href="/sites/default/files/css/css\_db11adabae0aed6b79a2c3c52def4754.css" />

<!--[if IE 6]>

<link type="text/css" rel="stylesheet" media="all" href="/sites/all/themes/oho\_basic/css/ie6.css?g" />

<![endif]-->

<!--[if IE 7]>

<link type="text/css" rel="stylesheet" media="all" href="/sites/all/themes/oho\_basic/css/ie7.css?g" />

<![endif]-->

<script type="text/javascript" src="/sites/default/files/js/js\_74d97b1176812e2fd6e43d62503a5204.js"></script>

<script type="text/javascript">

<!--//--><![CDATA[//><!--

### READ\_RAW

The READ\_RAW procedure returns the data from the HTTP response body in binary form. The number of bytes returned is specified by the *len* parameter. The signature is:

READ\_RAW(r IN OUT UTL\_HTTP.RESP, data OUT RAW, len INTEGER)

Parameters

r

r is the HTTP response record.

data

data is the response body in binary form.

len

Set len to the number of bytes of data to be returned.

Examples

The following example retrieves and displays the first 150 bytes in binary form.

DECLARE

v\_req UTL\_HTTP.REQ;

v\_resp UTL\_HTTP.RESP;

v\_data RAW;

BEGIN

v\_req := UTL\_HTTP.BEGIN\_REQUEST('http://www.enterprisedb.com');

v\_resp := UTL\_HTTP.GET\_RESPONSE(v\_req);

UTL\_HTTP.READ\_RAW(v\_resp, v\_data, 150);

DBMS\_OUTPUT.PUT\_LINE(v\_data);

UTL\_HTTP.END\_RESPONSE(v\_resp);

END;

The following is the output from the example.

\x3c21444f43545950452068746d6c205055424c494320222d2f2f5733432f2f445444205848544d4c20312e30205374726963742f2f454e220d0a202022687474703a2f2f7777772e77332e6f72672f54522f7868746d6c312f4454442f7868746d6c312d7374726963742e647464223e0d0a3c68746d6c20786d6c6e733d22687474703a2f2f7777772e77332e6f72672f313939392f

### READ\_TEXT

The READ\_TEXT procedure returns the data from the HTTP response body in text form. The maximum number of characters returned is specified by the *len* parameter. The signature is:

READ\_TEXT(r IN OUT UTL\_HTTP.RESP, data OUT VARCHAR2, len INTEGER)

Parameters

r

r is the HTTP response record.

data

data is the response body in text form.

len

Set len to the maximum number of characters to be returned.

Examples

The following example retrieves the first 150 characters.

DECLARE

v\_req UTL\_HTTP.REQ;

v\_resp UTL\_HTTP.RESP;

v\_data VARCHAR2(150);

BEGIN

v\_req := UTL\_HTTP.BEGIN\_REQUEST('http://www.enterprisedb.com');

v\_resp := UTL\_HTTP.GET\_RESPONSE(v\_req);

UTL\_HTTP.READ\_TEXT(v\_resp, v\_data, 150);

DBMS\_OUTPUT.PUT\_LINE(v\_data);

UTL\_HTTP.END\_RESPONSE(v\_resp);

END;

The following is the output.

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"

"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<html xmlns="http://www.w3.org/1999/

### REQUEST

The REQUEST function returns the first 2000 bytes retrieved from a user-specified URL. The signature is:

REQUEST(url IN VARCHAR2) RETURN VARCHAR2

If the data found at the given URL is longer than 2000 bytes, the remainder will be discarded. If the data found at the given URL is shorter than 2000 bytes, the result will be shorter than 2000 bytes.

Parameters

url

url is the Uniform Resource Locator from which UTL\_HTTP will return content.

Example

The following command returns the first 2000 bytes retrieved from the EnterpriseDB website:

SELECT UTL\_HTTP.REQUEST('http://www.enterprisedb.com/') FROM DUAL;

### REQUEST\_PIECES

The REQUEST\_PIECES function returns a table of 2000-byte segments retrieved from an URL. The signature is:

REQUEST\_PIECES(url IN VARCHAR2, max\_pieces NUMBER IN DEFAULT 32767) RETURN UTL\_HTTP.HTML\_PIECES

Parameters

url

url is the Uniform Resource Locator from which UTL\_HTTP will return content.

max\_pieces

max\_pieces specifies the maximum number of 2000-byte segments that the REQUEST\_PIECES function will return. If max\_pieces specifies more units than are available at the specified url, the final unit will contain fewer bytes.

Example

The following example returns the first four 2000 byte segments retrieved from the EnterpriseDB website:

DECLARE

result UTL\_HTTP.HTML\_PIECES;

BEGIN

result := UTL\_HTTP.REQUEST\_PIECES('http://www.enterprisedb.com/', 4);

END;

### SET\_BODY\_CHARSET

The SET\_BODY\_CHARSET procedure sets the default character set of the body of future HTTP requests. The signature is:

SET\_BODY\_CHARSET(charset VARCHAR2 DEFAULT NULL)

Parameters

charset

charset is the character set of the body of future requests. The default is null in which case the database character set is assumed.

### SET\_FOLLOW\_REDIRECT

The SET\_FOLLOW\_REDIRECT procedure sets the maximum number of times the HTTP redirect instruction is to be followed in the response to this request or future requests. This procedures has two signatures:

SET\_FOLLOW\_REDIRECT(max\_redirects IN INTEGER DEFAULT 3)

and

SET\_FOLLOW\_REDIRECT(r IN OUT UTL\_HTTP.REQ, max\_redirects IN INTEGER DEFAULT 3)

Use the second form to change the maximum number of redirections for an individual request that a request inherits from the session default settings.

Parameters

r

r is the HTTP request record.

max\_redirects

max\_redirects is maximum number of redirections allowed. Set to 0 to disable redirections. The default is 3.

### SET\_HEADER

The SET\_HEADER procedure sets the HTTP request header. The signature is:

SET\_HEADER(r IN OUT UTL\_HTTP.REQ, name IN VARCHAR2, value IN VARCHAR2 DEFAULT NULL)

Parameters

r

r is the HTTP request record.

name

name is the name of the request header.

value

value is the value of the request header. The default is null.

### SET\_RESPONSE\_ERROR\_CHECK

The SET\_RESPONSE\_ERROR\_CHECK procedure determines whether or not HTTP 4xx and 5xx status codes returned by the GET\_RESPONSE function should be interpreted as errors. The signature is:

SET\_RESPONSE\_ERROR\_CHECK(enable IN BOOLEAN DEFAULT FALSE)

Parameters

enable

Set enable to TRUE if HTTP 4xx and 5xx status codes are to be treated as errors, otherwise set to FALSE. The default is FALSE.

### SET\_TRANSFER\_TIMEOUT

The SET\_TRANSFER\_TIMEOUT procedure sets the default, transfer timeout setting for waiting for a response from an HTTP request. This procedure has two signatures:

SET\_TRANSFER\_TIMEOUT(timeout IN INTEGER DEFAULT 60)

and

SET\_TRANSFER\_TIMEOUT(r IN OUT UTL\_HTTP.REQ, timeout IN INTEGER DEFAULT 60)

Use the second form to change the transfer timeout setting for an individual request that a request inherits from the session default settings.

Parameters

r

r is the HTTP request record.

timeout

timeout is the transfer timeout setting in seconds for HTTP requests. The default is 60 seconds.

### WRITE\_LINE

The WRITE\_LINE procedure writes data to the HTTP request body in text form; the text is terminated with a CRLF character pair. The signature is:

WRITE\_LINE(*r* IN OUT UTL\_HTTP.REQ, *data* IN VARCHAR2)

Parameters

*r*

*r* is the HTTP request record.

*data*

*data* is the request body in TEXT form.

Example

The following example writes data (Account balance $500.00) in text form to the request body to be sent using the HTTP POST method. The data is sent to a hypothetical web application (post.php) that accepts and processes data.

DECLARE

v\_req UTL\_HTTP.REQ;

v\_resp UTL\_HTTP.RESP;

BEGIN

v\_req := UTL\_HTTP.BEGIN\_REQUEST('http://www.example.com/post.php',

'POST');

UTL\_HTTP.SET\_HEADER(v\_req, 'Content-Length', '23');

UTL\_HTTP.WRITE\_LINE(v\_req, 'Account balance $500.00');

v\_resp := UTL\_HTTP.GET\_RESPONSE(v\_req);

DBMS\_OUTPUT.PUT\_LINE('Status Code: ' || v\_resp.status\_code);

DBMS\_OUTPUT.PUT\_LINE('Reason Phrase: ' || v\_resp.reason\_phrase);

UTL\_HTTP.END\_RESPONSE(v\_resp);

END;

Assuming the web application successfully processed the POST method, the following output would be displayed:

Status Code: 200

Reason Phrase: OK

### WRITE\_RAW

The WRITE\_RAW procedure writes data to the HTTP request body in binary form. The signature is:

WRITE\_RAW(*r* IN OUT UTL\_HTTP.REQ, *data* IN RAW)

Parameters

*r*

*r* is the HTTP request record.

*data*

*data* is the request body in binary form.

Example

The following example writes data in binary form to the request body to be sent using the HTTP POST method to a hypothetical web application that accepts and processes such data.

DECLARE

v\_req UTL\_HTTP.REQ;

v\_resp UTL\_HTTP.RESP;

BEGIN

v\_req := UTL\_HTTP.BEGIN\_REQUEST('http://www.example.com/post.php',

'POST');

UTL\_HTTP.SET\_HEADER(v\_req, 'Content-Length', '23');

UTL\_HTTP.WRITE\_RAW(v\_req, HEXTORAW

('54657374696e6720504f5354206d6574686f6420696e20485454502072657175657374'));

v\_resp := UTL\_HTTP.GET\_RESPONSE(v\_req);

DBMS\_OUTPUT.PUT\_LINE('Status Code: ' || v\_resp.status\_code);

DBMS\_OUTPUT.PUT\_LINE('Reason Phrase: ' || v\_resp.reason\_phrase);

UTL\_HTTP.END\_RESPONSE(v\_resp);

END;

The text string shown in the HEXTORAW function is the hexadecimal translation of the text Testing POST method in HTTP request.

Assuming the web application successfully processed the POST method, the following output would be displayed:

Status Code: 200

Reason Phrase: OK

### WRITE\_TEXT

The WRITE\_TEXT procedure writes data to the HTTP request body in text form. The signature is:

WRITE\_TEXT(*r* IN OUT UTL\_HTTP.REQ, *data* IN VARCHAR2)

Parameters

*r*

*r* is the HTTP request record.

*data*

*data* is the request body in text form.

Example

The following example writes data (Account balance $500.00) in text form to the request body to be sent using the HTTP POST method. The data is sent to a hypothetical web application (post.php) that accepts and processes data.

DECLARE

v\_req UTL\_HTTP.REQ;

v\_resp UTL\_HTTP.RESP;

BEGIN

v\_req := UTL\_HTTP.BEGIN\_REQUEST('http://www.example.com/post.php',

'POST');

UTL\_HTTP.SET\_HEADER(v\_req, 'Content-Length', '23');

UTL\_HTTP.WRITE\_TEXT(v\_req, 'Account balance $500.00');

v\_resp := UTL\_HTTP.GET\_RESPONSE(v\_req);

DBMS\_OUTPUT.PUT\_LINE('Status Code: ' || v\_resp.status\_code);

DBMS\_OUTPUT.PUT\_LINE('Reason Phrase: ' || v\_resp.reason\_phrase);

UTL\_HTTP.END\_RESPONSE(v\_resp);

END;

Assuming the web application successfully processed the POST method, the following output would be displayed:

Status Code: 200

Reason Phrase: OK

## UTL\_MAIL

The UTL\_MAIL package provides the capability to manage e-mail.

Note: An administrator must grant execute privileges to each user or group before they can use this package.

Table ‑ UTL\_MAIL Functions/Procedures

| Function/Procedure | Function or Procedure | Return Type | Description |
| --- | --- | --- | --- |
| SEND(sender, recipients, cc, bcc, subject, message [, mime\_type [, priority ]]) | Procedure | n/a | Packages and sends an e-mail to an SMTP server. |
| SEND\_ATTACH\_RAW(sender, recipients, cc, bcc, subject, message, mime\_type, priority, attachment [, att\_inline [, att\_mime\_type [, att\_filename ]]]) | Procedure | n/a | Same as the SEND procedure, but with BYTEA or large object attachments. |
| SEND\_ATTACH\_VARCHAR2(sender, recipients, cc, bcc, subject, message, mime\_type, priority, attachment [, att\_inline [, att\_mime\_type [, att\_filename ]]]) | Procedure | n/a | Same as the SEND procedure, but with VARCHAR2 attachments. |

### SEND

The SEND procedure provides the capability to send an e-mail to an SMTP server.

SEND(sender VARCHAR2, recipients VARCHAR2, cc VARCHAR2,

bcc VARCHAR2, subject VARCHAR2, message VARCHAR2

[, mime\_type VARCHAR2 [, priority PLS\_INTEGER ]])

Parameters

sender

E-mail address of the sender.

recipients

Comma-separated e-mail addresses of the recipients.

cc

Comma-separated e-mail addresses of copy recipients.

bcc

Comma-separated e-mail addresses of blind copy recipients.

subject

Subject line of the e-mail.

message

Body of the e-mail.

mime\_type

Mime type of the message. The default is text/plain; charset=us-ascii.

priority

Priority of the e-mail The default is 3.

Examples

The following anonymous block sends a simple e-mail message.

DECLARE

v\_sender VARCHAR2(30);

v\_recipients VARCHAR2(60);

v\_subj VARCHAR2(20);

v\_msg VARCHAR2(200);

BEGIN

v\_sender := 'jsmith@enterprisedb.com';

v\_recipients := 'ajones@enterprisedb.com,rrogers@enterprisedb.com';

v\_subj := 'Holiday Party';

v\_msg := 'This year''s party is scheduled for Friday, Dec. 21 at ' ||

'6:00 PM. Please RSVP by Dec. 15.';

UTL\_MAIL.SEND(v\_sender,v\_recipients,NULL,NULL,v\_subj,v\_msg);

END;

### SEND\_ATTACH\_RAW

The SEND\_ATTACH\_RAW procedure provides the capability to send an e-mail to an SMTP server with an attachment containing either BYTEA data or a large object (identified by the large object's OID). The call to SEND\_ATTACH\_RAW can be written in two ways:

SEND\_ATTACH\_RAW(sender VARCHAR2, recipients VARCHAR2,

cc VARCHAR2, bcc VARCHAR2, subject VARCHAR2, message VARCHAR2,

mime\_type VARCHAR2, priority PLS\_INTEGER,

attachment BYTEA[, att\_inline BOOLEAN

[, att\_mime\_type VARCHAR2[, att\_filename VARCHAR2 ]]])

SEND\_ATTACH\_RAW(sender VARCHAR2, recipients VARCHAR2,

cc VARCHAR2, bcc VARCHAR2, subject VARCHAR2, message VARCHAR2,

mime\_type VARCHAR2, priority PLS\_INTEGER, attachment OID

[, att\_inline BOOLEAN [, att\_mime\_type VARCHAR2

[, att\_filename VARCHAR2 ]]])

Parameters

sender

E-mail address of the sender.

recipients

Comma-separated e-mail addresses of the recipients.

cc

Comma-separated e-mail addresses of copy recipients.

bcc

Comma-separated e-mail addresses of blind copy recipients.

subject

Subject line of the e-mail.

message

Body of the e-mail.

mime\_type

Mime type of the message. The default is text/plain; charset=us-ascii.

priority

Priority of the e-mail. The default is 3.

attachment

The attachment.

att\_inline

If set to TRUE, then the attachment is viewable inline, FALSE otherwise. The default is TRUE.

att\_mime\_type

Mime type of the attachment. The default is application/octet.

att\_filename

The file name containing the attachment. The default is NULL.

### SEND\_ATTACH\_VARCHAR2

The SEND\_ATTACH\_VARCHAR2 procedure provides the capability to send an e-mail to an SMTP server with a text attachment.

SEND\_ATTACH\_VARCHAR2(sender VARCHAR2, recipients VARCHAR2,

cc VARCHAR2, bcc VARCHAR2, subject VARCHAR2, message VARCHAR2,

mime\_type VARCHAR2, priority PLS\_INTEGER, attachment VARCHAR2

[, att\_inline BOOLEAN [, att\_mime\_type VARCHAR2

[, att\_filename VARCHAR2 ]]])

Parameters

sender

E-mail address of the sender.

recipients

Comma-separated e-mail addresses of the recipients.

cc

Comma-separated e-mail addresses of copy recipients.

bcc

Comma-separated e-mail addresses of blind copy recipients.

subject

Subject line of the e-mail.

message

Body of the e-mail.

mime\_type

Mime type of the message. The default is text/plain; charset=us-ascii.

priority

Priority of the e-mail The default is 3.

attachment

The VARCHAR2 attachment.

att\_inline

If set to TRUE, then the attachment is viewable inline, FALSE otherwise. The default is TRUE.

att\_mime\_type

Mime type of the attachment. The default is text/plain; charset=us-ascii.

att\_filename

The file name containing the attachment. The default is NULL.

## UTL\_RAW

The UTL\_RAW package allows you to manipulate or retrieve the length of raw data types.

Note: An administrator must grant execute privileges to each user or group before they can use this package.

| Function/Procedure | Function or Procedure | Return Type | Description |
| --- | --- | --- | --- |
| CAST\_TO\_RAW(*c* IN VARCHAR2) | Function | RAW | Converts a VARCHAR2 string to a RAW value. |
| CAST\_TO\_VARCHAR2(*r* IN RAW) | Function | VARCHAR2 | Converts a RAW value to a VARCHAR2 string. |
| CONCAT(*r1* IN RAW, *r2* IN RAW, *r3* IN RAW,…) | Function | RAW | Concatenate multiple RAW values into a single RAW value. |
| CONVERT(*r* IN RAW, *to*\_*charset* IN VARCHAR2, *from*\_*charset* IN VARCHAR2 | Function | RAW | Converts encoded data from one encoding to another, and returns the result as a RAW value. |
| LENGTH(*r* IN RAW) | Function | NUMBER | Returns the length of a RAW value. |
| SUBSTR(*r* IN RAW, *pos* IN INTEGER, *len* IN INTEGER) | Function | RAW | Returns a portion of a RAW value. |

Advanced Server's implementation of UTL\_RAW is a partial implementation when compared to Oracle's version. Only those functions and procedures listed in the table above are supported.

### CAST\_TO\_RAW

The CAST\_TO\_RAW function converts a VARCHAR2 string to a RAW value. The signature is:

CAST\_TO\_RAW(*c* VARCHAR2)

The function returns a RAW value if you pass a non-NULL value; if you pass a NULL value, the function will return NULL.

**Parameters**

c

The VARCHAR2 value that will be converted to RAW.

**Example**

The following example uses the CAST\_TO\_RAW function to convert a VARCHAR2 string to a RAW value:

DECLARE

v VARCHAR2;

r RAW;

BEGIN

v := 'Accounts';

dbms\_output.put\_line(v);

r := UTL\_RAW.CAST\_TO\_RAW(v);

dbms\_output.put\_line(r);

END;

The result set includes the content of the original string and the converted RAW value:

Accounts

\x4163636f756e7473

### CAST\_TO\_VARCHAR2

The CAST\_TO\_VARCHAR2 function converts RAW data to VARCHAR2 data. The signature is:

CAST\_TO\_VARCHAR2(*r* RAW)

The function returns a VARCHAR2 value if you pass a non-NULL value; if you pass a NULL value, the function will return NULL.

**Parameters**

r

The RAW value that will be converted to a VARCHAR2 value.

**Example**

The following example uses the CAST\_TO\_VARCHAR2 function to convert a RAW value to a VARCHAR2 string:

DECLARE

r RAW;

v VARCHAR2;

BEGIN

r := '\x4163636f756e7473'

dbms\_output.put\_line(v);

v := UTL\_RAW.CAST\_TO\_VARCHAR2(r);

dbms\_output.put\_line(r);

END;

The result set includes the content of the original string and the converted RAW value:

\x4163636f756e7473

Accounts

### CONCAT

The CONCAT function concatenates multiple RAW values into a single RAW value. The signature is:

CONCAT(*r1* RAW, *r2* RAW, *r3* RAW,…)

The function returns a RAW value. Unlike the Oracle implementation, the Advanced Server implementation is a variadic function, and does not place a restriction on the number of values that can be concatenated.

**Parameters**

r1, r2, r3,…

The RAW values that CONCAT will concatenate.

**Example**

The following example uses the CONCAT function to concatenate multiple RAW values into a single RAW value:

SELECT UTL\_RAW.CAST\_TO\_VARCHAR2(UTL\_RAW.CONCAT('\x61', '\x62', '\x63')) FROM DUAL;  
 concat

--------  
 abc  
(1 row)

The result (the concatenated values) is then converted to VARCHAR2 format by the CAST\_TO\_VARCHAR2 function.

### CONVERT

The CONVERT function converts a string from one encoding to another encoding and returns the result as a RAW value. The signature is:

CONVERT(r RAW, to\_charset VARCHAR2, from\_charset VARCHAR2)

The function returns a RAW value.

**Parameters**

r

The RAW value that will be converted.

*to\_charset*

The name of the encoding to which *r* will be converted.

*from\_charset*

The name of the encoding from which *r* will be converted.

**Example**

The following example uses the UTL\_RAW.CAST\_TO\_RAW function to convert a VARCHAR2 string (Accounts) to a raw value, and then convert the value from UTF8 to LATIN7, and then from LATIN7 to UTF8:

DECLARE  
 r RAW;  
 v VARCHAR2;  
BEGIN  
 v:= 'Accounts';  
 dbms\_output.put\_line(v);

r:= UTL\_RAW.CAST\_TO\_RAW(v);  
 dbms\_output.put\_line(r);  
 r:= UTL\_RAW.CONVERT(r, 'UTF8', 'LATIN7');  
 dbms\_output.put\_line(r);  
 r:= UTL\_RAW.CONVERT(r, 'LATIN7', 'UTF8');  
 dbms\_output.put\_line(r);

The example returns the VARCHAR2 value, the RAW value, and the converted values:

Accounts

\x4163636f756e7473

\x4163636f756e7473

\x4163636f756e7473

### LENGTH

The LENGTH function returns the length of a RAW value. The signature is:

LENGTH(*r* RAW)

The function returns a RAW value.

**Parameters**

r

The RAW value that LENGTH will evaluate.

**Example**

The following example uses the LENGTH function to return the length of a RAW value:

SELECT UTL\_RAW.LENGTH(UTL\_RAW.CAST\_TO\_RAW('Accounts')) FROM DUAL;  
 length

--------  
8  
(1 row)

The following example uses the LENGTH function to return the length of a RAW value that includes multi-byte characters:

SELECT UTL\_RAW.LENGTH(UTL\_RAW.CAST\_TO\_RAW('独孤求败'));

length

--------

12

(1 row)

### SUBSTR

The SUBSTR function returns a substring of a RAW value. The signature is:

SUBSTR (*r* RAW, *pos* INTEGER, *len* INTEGER)

This function returns a RAW value.

**Parameters**

r

The RAW value from which the substring will be returned.

*pos*

The position within the RAW value of the first byte of the returned substring.

* If *pos* is 0 or 1, the substring begins at the first byte of the RAW value.
* If *pos* is greater than one, the substring begins at the first byte specified by *pos*. For example, if *pos* is 3, the substring begins at the third byte of the value.
* If *pos* is negative, the substring begins at *pos* bytes from the end of the source value. For example, if *pos* is -3, the substring begins at the third byte from the end of the value.

*len*

The maximum number of bytes that will be returned.

**Example**

The following example uses the SUBSTR function to select a substring that begins 3 bytes from the start of a RAW value:

SELECT UTL\_RAW.SUBSTR(UTL\_RAW.CAST\_TO\_RAW('Accounts'), 3, 5) FROM DUAL;

substr  
--------  
 count  
(1 row)

The following example uses the SUBSTR function to select a substring that starts 5 bytes from the end of a RAW value:

SELECT UTL\_RAW.SUBSTR(UTL\_RAW.CAST\_TO\_RAW('Accounts'), -5 , 3) FROM DUAL;

substr

--------

oun

(1 row)

## UTL\_SMTP

The UTL\_SMTP package provides the capability to send e-mails over the Simple Mail Transfer Protocol (SMTP).

Note: An administrator must grant execute privileges to each user or group before they can use this package.

Table ‑ UTL\_SMTP Functions/Procedures

| Function/Procedure | Function or Procedure | Return Type | Description |
| --- | --- | --- | --- |
| CLOSE\_DATA(c IN OUT) | Procedure | n/a | Ends an e-mail message. |
| COMMAND(c IN OUT, cmd [, arg ]) | Both | REPLY | Execute an SMTP command. |
| COMMAND\_REPLIES(c IN OUT, cmd [, arg ]) | Function | REPLIES | Execute an SMTP command where multiple reply lines are expected. |
| DATA(c IN OUT, body VARCHAR2) | Procedure | n/a | Specify the body of an e-mail message. |
| EHLO(c IN OUT, domain) | Procedure | n/a | Perform initial handshaking with an SMTP server and return extended information. |
| HELO(c IN OUT, domain) | Procedure | n/a | Perform initial handshaking with an SMTP server |
| HELP(c IN OUT [, command ]) | Function | REPLIES | Send the HELP command. |
| MAIL(c IN OUT, sender [, parameters ]) | Procedure | n/a | Start a mail transaction. |
| NOOP(c IN OUT) | Both | REPLY | Send the null command. |
| OPEN\_CONNECTION(host [, port [, tx\_timeout ]]) | Function | CONNECTION | Open a connection. |
| OPEN\_DATA(c IN OUT) | Both | REPLY | Send the DATA command. |
| QUIT(c IN OUT) | Procedure | n/a | Terminate the SMTP session and disconnect. |
| RCPT(c IN OUT, recipient [, parameters ]) | Procedure | n/a | Specify the recipient of an e-mail message. |
| RSET(c IN OUT) | Procedure | n/a | Terminate the current mail transaction. |
| VRFY(c IN OUT, recipient) | Function | REPLY | Validate an e-mail address. |
| WRITE\_DATA(c IN OUT, data) | Procedure | n/a | Write a portion of the e-mail message. |

The following table lists the public variables available in the UTL\_SMTP package.

Table ‑ UTL\_SMTP Public Variables

| Public Variables | Data Type | Value | Description |
| --- | --- | --- | --- |
| connection | RECORD |  | Description of an SMTP connection. |
| reply | RECORD |  | SMTP reply line. |

### CONNECTION

The CONNECTION record type provides a description of an SMTP connection.

TYPE connection IS RECORD (

host VARCHAR2(255),

port PLS\_INTEGER,

tx\_timeout PLS\_INTEGER

);

### REPLY/REPLIES

The REPLY record type provides a description of an SMTP reply line. REPLIES is a table of multiple SMTP reply lines.

TYPE reply IS RECORD (

code INTEGER,

text VARCHAR2(508)

);

TYPE replies IS TABLE OF reply INDEX BY BINARY\_INTEGER;

### CLOSE\_DATA

The CLOSE\_DATA procedure terminates an e-mail message by sending the following sequence:

<CR><LF>.<CR><LF>

This is a single period at the beginning of a line.

CLOSE\_DATA(c IN OUT CONNECTION)

Parameters

c

The SMTP connection to be closed.

### COMMAND

The COMMAND procedure provides the capability to execute an SMTP command. If you are expecting multiple reply lines, use COMMAND\_REPLIES.

reply REPLY COMMAND(c IN OUT CONNECTION, cmd VARCHAR2

[, arg VARCHAR2 ])

COMMAND(c IN OUT CONNECTION, cmd VARCHAR2 [, arg VARCHAR2 ])

Parameters

c

The SMTP connection to which the command is to be sent.

cmd

The SMTP command to be processed.

arg

An argument to the SMTP command. The default is null.

reply

SMTP reply to the command. If SMTP returns multiple replies, only the last one is returned in reply.

See Section 9.21.2 for a description of REPLY and REPLIES.

### COMMAND\_REPLIES

The COMMAND\_REPLIES function processes an SMTP command that returns multiple reply lines. Use COMMAND if only a single reply line is expected.

replies REPLIES COMMAND(c IN OUT CONNECTION, cmd VARCHAR2

[, arg VARCHAR2 ])

Parameters

c

The SMTP connection to which the command is to be sent.

cmd

The SMTP command to be processed.

arg

An argument to the SMTP command. The default is null.

replies

SMTP reply lines to the command. See Section 9.21.2 for a description of REPLY and REPLIES.

### DATA

The DATA procedure provides the capability to specify the body of the e-mail message. The message is terminated with a <CR><LF>.<CR><LF> sequence.

DATA(c IN OUT CONNECTION, body VARCHAR2)

Parameters

c

The SMTP connection to which the command is to be sent.

body

Body of the e-mail message to be sent.

### EHLO

The EHLO procedure performs initial handshaking with the SMTP server after establishing the connection. The EHLO procedure allows the client to identify itself to the SMTP server according to RFC 821. RFC 1869 specifies the format of the information returned in the server’s reply. The HELO procedure performs the equivalent functionality, but returns less information about the server.

EHLO(c IN OUT CONNECTION, domain VARCHAR2)

Parameters

c

The connection to the SMTP server over which to perform handshaking.

domain

Domain name of the sending host.

### HELO

The HELO procedure performs initial handshaking with the SMTP server after establishing the connection. The HELO procedure allows the client to identify itself to the SMTP server according to RFC 821. The EHLO procedure performs the equivalent functionality, but returns more information about the server.

HELO(c IN OUT, domain VARCHAR2)

Parameters

c

The connection to the SMTP server over which to perform handshaking.

domain

Domain name of the sending host.

### HELP

The HELP function provides the capability to send the HELP command to the SMTP server.

replies REPLIES HELP(c IN OUT CONNECTION [, command VARCHAR2 ])

Parameters

c

The SMTP connection to which the command is to be sent.

command

Command on which help is requested.

replies

SMTP reply lines to the command. See Section 9.21.2 for a description of REPLY and REPLIES.

### MAIL

The MAIL procedure initiates a mail transaction.

MAIL(c IN OUT CONNECTION, sender VARCHAR2

[, parameters VARCHAR2 ])

Parameters

c

Connection to SMTP server on which to start a mail transaction.

sender

The sender’s e-mail address.

parameters

Mail command parameters in the format, key=value as defined in RFC 1869, Section 6.

### NOOP

The NOOP function/procedure sends the null command to the SMTP server. The NOOP has no effect upon the server except to obtain a successful response.

reply REPLY NOOP(c IN OUT CONNECTION)

NOOP(c IN OUT CONNECTION)

Parameters

c

The SMTP connection on which to send the command.

reply

SMTP reply to the command. If SMTP returns multiple replies, only the last one is returned in reply. See Section 9.21.2 for a description of REPLY and REPLIES.

### OPEN\_CONNECTION

The OPEN\_CONNECTION functions open a connection to an SMTP server.

c CONNECTION OPEN\_CONNECTION(host VARCHAR2 [, port PLS\_INTEGER [, tx\_timeout PLS\_INTEGER DEFAULT NULL]])

Parameters

host

Name of the SMTP server.

port

Port number on which the SMTP server is listening. The default is 25.

tx\_timeout

Time out value in seconds. Do not wait is indicated by specifying 0. Wait indefinitely is indicated by setting timeout to null. The default is null.

c

Connection handle returned by the SMTP server.

### OPEN\_DATA

The OPEN\_DATA procedure sends the DATA command to the SMTP server.

OPEN\_DATA(c IN OUT CONNECTION)

Parameters

c

SMTP connection on which to send the command.

### QUIT

The QUIT procedure closes the session with an SMTP server.

QUIT(c IN OUT CONNECTION)

Parameters

c

SMTP connection to be terminated.

### RCPT

The RCPT procedure provides the e-mail address of the recipient. To schedule multiple recipients, invoke RCPT multiple times.

RCPT(c IN OUT CONNECTION, recipient VARCHAR2

[, parameters VARCHAR2 ])

Parameters

c

Connection to SMTP server on which to add a recipient.

recipient

The recipient’s e-mail address.

parameters

Mail command parameters in the format, key=value as defined in RFC 1869, Section 6.

### RSET

The RSET procedure provides the capability to terminate the current mail transaction.

RSET(c IN OUT CONNECTION)

Parameters

c

SMTP connection on which to cancel the mail transaction.

### VRFY

The VRFY function provides the capability to validate and verify the recipient’s e-mail address. If valid, the recipient’s full name and fully qualified mailbox is returned.

reply REPLY VRFY(c IN OUT CONNECTION, recipient VARCHAR2)

Parameters

c

The SMTP connection on which to verify the e-mail address.

recipient

The recipient’s e-mail address to be verified.

reply

SMTP reply to the command. If SMTP returns multiple replies, only the last one is returned in reply. See Section 9.21.2 for a description of REPLY and REPLIES.

### WRITE\_DATA

The WRITE\_DATA procedure provides the capability to add VARCHAR2 data to an e-mail message. The WRITE\_DATA procedure may be repetitively called to add data.

WRITE\_DATA(c IN OUT CONNECTION, data VARCHAR2)

Parameters

c

The SMTP connection on which to add data.

data

Data to be added to the e-mail message. The data must conform to the RFC 822 specification.

### Comprehensive Example

The following procedure constructs and sends a text e-mail message using the UTL\_SMTP package.

CREATE OR REPLACE PROCEDURE send\_mail (

p\_sender VARCHAR2,

p\_recipient VARCHAR2,

p\_subj VARCHAR2,

p\_msg VARCHAR2,

p\_mailhost VARCHAR2

)

IS

v\_conn UTL\_SMTP.CONNECTION;

v\_crlf CONSTANT VARCHAR2(2) := CHR(13) || CHR(10);

v\_port CONSTANT PLS\_INTEGER := 25;

BEGIN

v\_conn := UTL\_SMTP.OPEN\_CONNECTION(p\_mailhost,v\_port);

UTL\_SMTP.HELO(v\_conn,p\_mailhost);

UTL\_SMTP.MAIL(v\_conn,p\_sender);

UTL\_SMTP.RCPT(v\_conn,p\_recipient);

UTL\_SMTP.DATA(v\_conn, SUBSTR(

'Date: ' || TO\_CHAR(SYSDATE,

'Dy, DD Mon YYYY HH24:MI:SS') || v\_crlf

|| 'From: ' || p\_sender || v\_crlf

|| 'To: ' || p\_recipient || v\_crlf

|| 'Subject: ' || p\_subj || v\_crlf

|| p\_msg

, 1, 32767));

UTL\_SMTP.QUIT(v\_conn);

END;

EXEC send\_mail('asmith@enterprisedb.com','pjones@enterprisedb.com','Holiday Party','Are you planning to attend?','smtp.enterprisedb.com');

The following example uses the OPEN\_DATA, WRITE\_DATA, and CLOSE\_DATA procedures instead of the DATA procedure.

CREATE OR REPLACE PROCEDURE send\_mail\_2 (

p\_sender VARCHAR2,

p\_recipient VARCHAR2,

p\_subj VARCHAR2,

p\_msg VARCHAR2,

p\_mailhost VARCHAR2

)

IS

v\_conn UTL\_SMTP.CONNECTION;

v\_crlf CONSTANT VARCHAR2(2) := CHR(13) || CHR(10);

v\_port CONSTANT PLS\_INTEGER := 25;

BEGIN

v\_conn := UTL\_SMTP.OPEN\_CONNECTION(p\_mailhost,v\_port);

UTL\_SMTP.HELO(v\_conn,p\_mailhost);

UTL\_SMTP.MAIL(v\_conn,p\_sender);

UTL\_SMTP.RCPT(v\_conn,p\_recipient);

UTL\_SMTP.OPEN\_DATA(v\_conn);

UTL\_SMTP.WRITE\_DATA(v\_conn,'From: ' || p\_sender || v\_crlf);

UTL\_SMTP.WRITE\_DATA(v\_conn,'To: ' || p\_recipient || v\_crlf);

UTL\_SMTP.WRITE\_DATA(v\_conn,'Subject: ' || p\_subj || v\_crlf);

UTL\_SMTP.WRITE\_DATA(v\_conn,v\_crlf || p\_msg);

UTL\_SMTP.CLOSE\_DATA(v\_conn);

UTL\_SMTP.QUIT(v\_conn);

END;

EXEC send\_mail\_2('asmith@enterprisedb.com','pjones@enterprisedb.com','Holiday Party','Are you planning to attend?','smtp.enterprisedb.com');

## UTL\_URL

The UTL\_URL package provides a way to escape illegal and reserved characters within an URL.

Table 7.7.2 UTL\_HTTP Functions and Procedures

| Function/Procedure | Return Type | Description |
| --- | --- | --- |
| ESCAPE(url, escape\_reserved\_chars, url\_charset) | VARCHAR2 | Use the ESCAPE function to escape any illegal and reserved characters in a URL. |
| UNESCAPE(url, url\_charset) | VARCHAR2 | The UNESCAPE function to convert an URL to its original form. |

The UTL\_URL package will return the BAD\_URL exception if the call to a function includes an incorrectly-formed URL.

### ESCAPE

Use the ESCAPE function to escape illegal and reserved characters within an URL. The signature is:

ESCAPE(url VARCHAR2, escape\_reserved\_chars BOOLEAN, url\_charset VARCHAR2)

Reserved characters are replaced with a percent sign, followed by the two-digit hex code of the ascii value for the escaped character.

Parameters

url

url specifies the Uniform Resource Locator that UTL\_URL will escape.

escape\_reserved\_chars

escape\_reserved\_chars is a BOOLEAN value that instructs the ESCAPE function to escape reserved characters as well as illegal characters:

* If escaped\_reserved\_chars is FALSE, ESCAPE will escape only the illegal characters in the specified URL.
* If escape\_reserved\_chars is TRUE, ESCAPE will escape both the illegal characters and the reserved characters in the specified URL.

By default, escape\_reserved\_chars is FALSE.

Within an URL, legal characters are:

|  |  |  |
| --- | --- | --- |
| Uppercase A through Z | Lowercase a through z | 0 through 9 |
| asterisk (\*) | exclamation point (!) | hyphen (-) |
| left parenthesis (() | period (.) | right parenthesis ()) |
| single-quote (') | tilde (~) | underscore (\_) |

Some characters are legal in some parts of an URL, while illegal in others; to review comprehensive rules about illegal characters, please refer to RFC 2396. Some examples of characters that are considered illegal in any part of an URL are:

|  |  |
| --- | --- |
| Illegal Character | Escape Sequence |
| a blank space ( ) | %20 |
| curly braces ({ or }) | %7b and %7d |
| hash mark (#) | %23 |

The ESCAPE function considers the following characters to be reserved, and will escape them if escape\_reserved\_chars is set to TRUE:

|  |  |
| --- | --- |
| Reserved Character | Escape Sequence |
| ampersand (&) | %5C |
| at sign (@) | %25 |
| colon (:) | %3a |
| comma (,) | %2c |
| dollar sign ($) | %24 |
| equal sign (=) | %3d |
| plus sign (+) | %2b |
| question mark (?) | %3f |
| semi-colon (;) | %3b |
| slash (/) | %2f |

url\_charset

url\_charset specifies a character set to which a given character will be converted before it is escaped. If url\_charset is NULL, the character will not be converted. The default value of url\_charset is ISO-8859-1.

Examples

The following anonymous block uses the ESCAPE function to escape the blank spaces in the URL:

DECLARE  
 result varchar2(400);  
BEGIN  
 result := UTL\_URL.ESCAPE('http://www.example.com/Using the ESCAPE function.html');  
 DBMS\_OUTPUT.PUT\_LINE(result);  
END;

The resulting (escaped) URL is:

http://www.example.com/Using%20the%20ESCAPE%20function.html

If you include a value of TRUE for the escape\_reserved\_chars parameter when invoking the function:

DECLARE  
 result varchar2(400);  
BEGIN  
 result := UTL\_URL.ESCAPE('http://www.example.com/Using the ESCAPE function.html', TRUE);  
 DBMS\_OUTPUT.PUT\_LINE(result);  
END;

The ESCAPE function escapes the reserved characters as well as the illegal characters in the URL:

http%3A%2F%2Fwww.example.com%2FUsing%20the%20ESCAPE%20function.html

### UNESCAPE

The UNESCAPE function removes escape characters added to an URL by the ESCAPE function, converting the URL to its original form.

The signature is:

UNESCAPE(url VARCHAR2, url\_charset VARCHAR2)

Parameters

url

url specifies the Uniform Resource Locator that UTL\_URL will unescape.

url\_charset

After unescaping a character, the character is assumed to be in url\_charset encoding, and will be converted from that encoding to database encoding before being returned. If url\_charset is NULL, the character will not be converted. The default value of url\_charset is ISO-8859-1.

Examples

The following anonymous block uses the ESCAPE function to escape the blank spaces in the URL:

DECLARE  
 result varchar2(400);  
BEGIN  
 result := UTL\_URL.UNESCAPE('http://www.example.com/Using%20the%20UNESCAPE%20function.html');  
 DBMS\_OUTPUT.PUT\_LINE(result);  
END;

The resulting (unescaped) URL is:

http://www.example.com/Using the UNESCAPE function.html

# Expanded Catalog Views

The Expanded Catalog Views provide comprehensive information from another perspective about database objects.

## ALL\_ALL\_TABLES

The ALL\_ALL\_TABLES view provides information about the tables accessible by the current user.

| Name | Type | Description |
| --- | --- | --- |
| owner | TEXT | User name of the table’s owner. |
| schema\_name | TEXT | Name of the schema in which the table belongs. |
| table\_name | TEXT | The name of the table. |
| tablespace\_name | TEXT | Name of the tablespace in which the table resides if other than the default tablespace. |
| status | CHARACTER VARYING (5) | Included for compatibility only; always set to VALID. |
| temporary | TEXT | Y if the table is temporary; N if the table is permanent. |

## ALL\_CONS\_COLUMNS

The ALL\_CONS\_COLUMNS view provides information about the columns specified in constraints placed on tables accessible by the current user.

| Name | Type | Description |
| --- | --- | --- |
| owner | TEXT | User name of the constraint’s owner. |
| schema\_name | TEXT | Name of the schema in which the constraint belongs. |
| constraint\_name | TEXT | The name of the constraint. |
| table\_name | TEXT | The name of the table to which the constraint belongs. |
| column\_name | TEXT | The name of the column referenced in the constraint. |
| position | SMALLINT | The position of the column within the object definition. |
| constraint\_def | TEXT | The definition of the constraint. |

## ALL\_CONSTRAINTS

The ALL\_CONSTRAINTS view provides information about the constraints placed on tables accessible by the current user.

| Name | Type | Description |
| --- | --- | --- |
| owner | TEXT | User name of the constraint’s owner. |
| schema\_name | TEXT | Name of the schema in which the constraint belongs. |
| constraint\_name | TEXT | The name of the constraint. |
| constraint\_type | TEXT | The constraint type. Possible values are:  C – check constraint  F – foreign key constraint  P – primary key constraint  U – unique key constraint  R – referential integrity constraint  V – constraint on a view  O – with read-only, on a view |
| table\_name | TEXT | Name of the table to which the constraint belongs. |
| search\_condition | TEXT | Search condition that applies to a check constraint. |
| r\_owner | TEXT | Owner of a table referenced by a referential constraint. |
| r\_constraint\_name | TEXT | Name of the constraint definition for a referenced table. |
| delete\_rule | TEXT | The delete rule for a referential constraint. Possible values are:  C – cascade  R – restrict  N – no action |
| deferrable | BOOLEAN | Specified if the constraint is deferrable (T or F). |
| deferred | BOOLEAN | Specifies if the constraint has been deferred (T or F). |
| index\_owner | TEXT | User name of the index owner. |
| index\_name | TEXT | The name of the index. |
| constraint\_def | TEXT | The definition of the constraint. |

## ALL\_DB\_LINKS

The ALL\_DB\_LINKS view provides information about the database links accessible by the current user.

| Name | Type | Description |
| --- | --- | --- |
| owner | TEXT | User name of the database link’s owner. |
| db\_link | TEXT | The name of the database link. |
| type | CHARACTER VARYING | Type of remote server. Value will be either REDWOOD or EDB |
| username | TEXT | User name of the user logging in. |
| host | TEXT | Name or IP address of the remote server. |

## ALL\_IND\_COLUMNS

The ALL\_IND\_COLUMNS view provides information about columns included in indexes on the tables accessible by the current user.

| Name | Type | Description |
| --- | --- | --- |
| index\_owner | TEXT | User name of the index’s owner. |
| schema\_name | TEXT | Name of the schema in which the index belongs. |
| index\_name | TEXT | The name of the index. |
| table\_owner | TEXT | User name of the table owner. |
| table\_name | TEXT | The name of the table to which the index belongs. |
| column\_name | TEXT | The name of the column. |
| column\_position | SMALLINT | The position of the column within the index. |
| column\_length | SMALLINT | The length of the column (in bytes). |
| char\_length | NUMERIC | The length of the column (in characters). |
| descend | CHARACTER(1) | Always set to Y (descending); included for compatibility only. |

## ALL\_INDEXES

The ALL\_INDEXES view provides information about the indexes on tables that may be accessed by the current user.

| Name | Type | Description |
| --- | --- | --- |
| owner | TEXT | User name of the index’s owner. |
| schema\_name | TEXT | Name of the schema in which the index belongs. |
| index\_name | TEXT | The name of the index. |
| index\_type | TEXT | The index type is always BTREE. Included for compatibility only. |
| table\_owner | TEXT | User name of the owner of the indexed table. |
| table\_name | TEXT | The name of the indexed table. |
| table\_type | TEXT | Included for compatibility only. Always set to TABLE. |
| uniqueness | TEXT | Indicates if the index is UNIQUE or NONUNIQUE. |
| compression | CHARACTER(1) | Always set to N (not compressed). Included for compatibility only. |
| tablespace\_name | TEXT | Name of the tablespace in which the table resides if other than the default tablespace. |
| logging | TEXT | Always set to LOGGING. Included for compatibility only. |
| status | TEXT | Included for compatibility only; always set to VALID. |
| partitioned | CHARACTER(3) | Indicates that the index is partitioned. Currently, always set to NO. |
| temporary | CHARACTER(1) | Indicates that an index is on a temporary table. Always set to N; included for compatibility only. |
| secondary | CHARACTER(1) | Included for compatibility only. Always set to N. |
| join\_index | CHARACTER(3) | Included for compatibility only. Always set to NO. |
| dropped | CHARACTER(3) | Included for compatibility only. Always set to NO. |

## ALL\_JOBS

The ALL\_JOBS view provides information about all jobs that reside in the database.

| Name | Type | Description |
| --- | --- | --- |
| job | INTEGER | The identifier of the job (Job ID). |
| log\_user | TEXT | The name of the user that submitted the job. |
| priv\_user | TEXT | Same as log\_user. Included for compatibility only. |
| schema\_user | TEXT | The name of the schema used to parse the job. |
| last\_date | TIMESTAMP WITH TIME ZONE | The last date that this job executed successfully. |
| last\_sec | TEXT | Same as last\_date. |
| this\_date | TIMESTAMP WITH TIME ZONE | The date that the job began executing. |
| this\_sec | TEXT | Same as this\_date |
| next\_date | TIMESTAMP WITH TIME ZONE | The next date that this job will be executed. |
| next\_sec | TEXT | Same as next\_date. |
| total\_time | INTERVAL | The execution time of this job (in seconds). |
| broken | TEXT | If Y, no attempt will be made to run this job.  If N, this job will attempt to execute. |
| interval | TEXT | Determines how often the job will repeat. |
| failures | BIGINT | The number of times that the job has failed to complete since it’s last successful execution. |
| what | TEXT | The job definition (PL/SQL code block) that runs when the job executes. |
| nls\_env | CHARACTER VARYING(4000) | Always NULL. Provided for compatibility only. |
| misc\_env | BYTEA | Always NULL. Provided for compatibility only. |
| instance | NUMERIC | Always 0. Provided for compatibility only. |

## ALL\_OBJECTS

The ALL\_OBJECTS view provides information about all objects that reside in the database.

| Name | Type | Description |
| --- | --- | --- |
| owner | TEXT | User name of the object’s owner. |
| schema\_name | TEXT | Name of the schema in which the object belongs. |
| object\_name | TEXT | Name of the object. |
| object\_type | TEXT | Type of the object – possible values are: INDEX, FUNCTION, PACKAGE, PACKAGE BODY, PROCEDURE, SEQUENCE, SYNONYM, TABLE, TRIGGER, and VIEW. |
| status | CHARACTER VARYING | Whether or not the state of the object is valid. Currently, Included for compatibility only; always set to VALID. |
| temporary | TEXT | Y if a temporary object; N if this is a permanent object. |

## ALL\_PART\_KEY\_COLUMNS

The ALL\_PART\_KEY\_COLUMNS view provides information about the key columns of the partitioned tables that reside in the database.

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| owner | TEXT | The owner of the table. |
| schema\_name | TEXT | The name of the schema in which the table resides. |
| name | TEXT | The name of the table in which the column resides. |
| object\_type | CHARACTER(5) | For compatibility only; always TABLE. |
| column\_name | TEXT | The name of the column on which the key is defined. |
| column\_position | INTEGER | 1 for the first column; 2 for the second column, etc. |

## ALL\_PART\_TABLES

The ALL\_PART\_TABLES view provides information about all of the partitioned tables that reside in the database.

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| owner | TEXT | The owner of the partitioned table. |
| schema\_name | TEXT | The name of the schema in which the table resides. |
| table\_name | TEXT | The name of the table. |
| partitioning\_type | TEXT | The partitioning type used to define table partitions. |
| subpartitioning\_type | TEXT | The subpartitioning type used to define table subpartitions. |
| partition\_count | BIGINT | The number of partitions in the table. |
| def\_subpartition\_count | INTEGER | The number of subpartitions in the table. |
| partitioning\_key\_count | INTEGER | The number of partitioning keys specified. |
| subpartitioning\_key\_count | INTEGER | The number of subpartitioning keys specified. |
| status | CHARACTER VARYING(8) | Provided for compatibility only. Always VALID. |
| def\_tablespace\_name | CHARACTER VARYING(30) | Provided for compatibility only. Always NULL. |
| def\_pct\_free | NUMERIC | Provided for compatibility only. Always NULL. |
| def\_pct\_used | NUMERIC | Provided for compatibility only. Always NULL. |
| def\_ini\_trans | NUMERIC | Provided for compatibility only. Always NULL. |
| def\_max\_trans | NUMERIC | Provided for compatibility only. Always NULL. |
| def\_initial\_extent | CHARACTER VARYING(40) | Provided for compatibility only. Always NULL. |
| def\_next\_extent | CHARACTER VARYING(40) | Provided for compatibility only. Always NULL. |
| def\_min\_extents | CHARACTER VARYING(40) | Provided for compatibility only. Always NULL. |
| def\_max\_extents | CHARACTER VARYING(40) | Provided for compatibility only. Always NULL. |
| def\_pct\_increase | CHARACTER VARYING(40) | Provided for compatibility only. Always NULL. |
| def\_freelists | NUMERIC | Provided for compatibility only. Always NULL. |
| def\_freelist\_groups | NUMERIC | Provided for compatibility only. Always NULL. |
| def\_logging | CHARACTER VARYING(7) | Provided for compatibility only. Always YES. |
| def\_compression | CHARACTER VARYING(8) | Provided for compatibility only. Always NONE |
| def\_buffer\_pool | CHARACTER VARYING(7) | Provided for compatibility only. Always DEFAULT |
| ref\_ptn\_constraint\_name | CHARACTER VARYING(30) | Provided for compatibility only. Always NULL |
| interval | CHARACTER VARYING(1000) | Provided for compatibility only. Always NULL |

## ALL\_POLICIES

The ALL\_POLICIES view provides information on all policies in the database. This view is accessible only to superusers.

| Name | Type | Description |
| --- | --- | --- |
| object\_owner | TEXT | Name of the owner of the object. |
| schema\_name | TEXT | Name of the schema in which the object belongs. |
| object\_name | TEXT | Name of the object on which the policy applies. |
| policy\_group | TEXT | Included for compatibility only; always set to an empty string. |
| policy\_name | TEXT | Name of the policy. |
| pf\_owner | TEXT | Name of the schema containing the policy function, or the schema containing the package that contains the policy function. |
| package | TEXT | Name of the package containing the policy function if the function belongs to a package. |
| function | TEXT | Name of the policy function. |
| sel | TEXT | Whether or not the policy applies to SELECT commands. Possible values are YES or NO. |
| ins | TEXT | Whether or not the policy applies to INSERT commands. Possible values are YES or NO. |
| upd | TEXT | Whether or not the policy applies to UPDATE commands. Possible values are YES or NO. |
| del | TEXT | Whether or not the policy applies to DELETE commands. Possible values are YES or NO. |
| idx | TEXT | Whether or not the policy applies to index maintenance. Possible values are YES or NO. |
| chk\_option | TEXT | Whether or not the check option is in force for INSERT and UPDATE commands. Possible values are YES or NO. |
| enable | TEXT | Whether or not the policy is enabled on the object. Possible values are YES or NO. |
| static\_policy | TEXT | Included for compatibility only; always set to NO. |
| policy\_type | TEXT | Included for compatibility only; always set to UNKNOWN. |
| long\_predicate | TEXT | Included for compatibility only; always set to YES. |

## ALL\_SEQUENCES

The ALL\_SEQUENCES view provides information about all user-defined sequences on which the user has select, or update privileges.

| Name | Type | Description |
| --- | --- | --- |
| sequence\_owner | TEXT | User name of the sequence's owner. |
| schema\_name | TEXT | Name of the schema in which the sequence resides. |
| sequence\_name | TEXT | Name of the sequence. |
| min\_value | NUMERIC | The lowest value that the server will assign to the sequence. |
| max\_value | NUMERIC | The highest value that the server will assign to the sequence. |
| increment\_by | NUMERIC | The value added to the current sequence number to create the next sequent number. |
| cycle\_flag | CHARACTER VARYING | Specifies if the sequence should wrap when it reaches min\_value or max\_value. |
| order\_flag | CHARACTER VARYING | This will always return Y. |
| cache\_size | NUMERIC | The number of pre-allocated sequence numbers stored in memory. |
| last\_number | NUMERIC | The value of the last sequence number saved to disk. |

## ALL\_SOURCE

The ALL\_SOURCE view provides a source code listing of the following program types: functions, procedures, triggers, package specifications, and package bodies.

| Name | Type | Description |
| --- | --- | --- |
| owner | TEXT | User name of the program’s owner. |
| schema\_name | TEXT | Name of the schema in which the program belongs. |
| name | TEXT | Name of the program. |
| type | TEXT | Type of program – possible values are: FUNCTION, PACKAGE, PACKAGE BODY, PROCEDURE, and TRIGGER. |
| line | INTEGER | Source code line number relative to a given program. |
| text | TEXT | Line of source code text. |

## ALL\_SUBPART\_KEY\_COLUMNS

The ALL\_SUBPART\_KEY\_COLUMNS view provides information about the key columns of those partitioned tables which are subpartitioned that reside in the database.

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| owner | TEXT | The owner of the table. |
| schema\_name | TEXT | The name of the schema in which the table resides. |
| name | TEXT | The name of the table in which the column resides. |
| object\_type | CHARACTER(5) | For compatibility only; always TABLE. |
| column\_name | TEXT | The name of the column on which the key is defined. |
| column\_position | INTEGER | 1 for the first column; 2 for the second column, etc. |

## ALL\_SYNONYMS

The ALL\_SYNONYMS view provides information on all synonyms that may be referenced by the current user.

| Name | Type | Description |
| --- | --- | --- |
| owner | TEXT | User name of the synonym’s owner. |
| schema\_name | TEXT | The name of the schema in which the synonym resides. |
| synonym\_name | TEXT | Name of the synonym. |
| table\_owner | TEXT | User name of the object’s owner. |
| table\_schema\_name | TEXT | The name of the schema in which the table resides. |
| table\_name | TEXT | The name of the object that the synonym refers to. |
| db\_link | TEXT | The name of any associated database link. |

## ALL\_TAB\_COLUMNS

The ALL\_TAB\_COLUMNS view provides information on all columns in all user-defined tables and views.

| Name | Type | Description |
| --- | --- | --- |
| owner | CHARACTER VARYING | User name of the owner of the table or view in which the column resides. |
| schema\_name | CHARACTER VARYING | Name of the schema in which the table or view resides. |
| table\_name | CHARACTER VARYING | Name of the table or view. |
| column\_name | CHARACTER VARYING | Name of the column. |
| data\_type | CHARACTER VARYING | Data type of the column. |
| data\_length | NUMERIC | Length of text columns. |
| data\_precision | NUMERIC | Precision (number of digits) for NUMBER columns. |
| data\_scale | NUMERIC | Scale of NUMBER columns. |
| nullable | CHARACTER(1) | Whether or not the column is nullable. Possible values are:  Y – column is nullable; N – column does not allow null. |
| column\_id | NUMERIC | Relative position of the column within the table or view. |
| data\_default | CHARACTER VARYING | Default value assigned to the column. |

## ALL\_TAB\_PARTITIONS

The ALL\_TAB\_PARTITIONS view provides information about all of the partitions that reside in the database.

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| table\_owner | TEXT | The owner of the table in which the partition resides. |
| schema\_name | TEXT | The name of the schema in which the table resides. |
| table\_name | TEXT | The name of the table. |
| composite | TEXT | YES if the table is subpartitioned; NO if the table is not subpartitioned. |
| partition\_name | TEXT | The name of the partition. |
| subpartition\_count | BIGINT | The number of subpartitions in the partition. |
| high\_value | TEXT | The high partitioning value specified in the CREATE TABLE statement. |
| high\_value\_length | INTEGER | The length of the partitioning value. |
| partition\_position | INTEGER | 1 for the first partition; 2 for the second partition, etc. |
| tablespace\_name | TEXT | The name of the tablespace in which the partition resides. |
| pct\_free | NUMERIC | Included for compatibility only; always 0 |
| pct\_used | NUMERIC | Included for compatibility only; always 0 |
| ini\_trans | NUMERIC | Included for compatibility only; always 0 |
| max\_trans | NUMERIC | Included for compatibility only; always 0 |
| initial\_extent | NUMERIC | Included for compatibility only; always NULL |
| next\_extent | NUMERIC | Included for compatibility only; always NULL |
| min\_extent | NUMERIC | Included for compatibility only; always 0 |
| max\_extent | NUMERIC | Included for compatibility only; always 0 |
| pct\_increase | NUMERIC | Included for compatibility only; always 0 |
| freelists | NUMERIC | Included for compatibility only; always NULL |
| freelist\_groups | NUMERIC | Included for compatibility only; always NULL |
| logging | CHARACTER VARYING(7) | Included for compatibility only; always YES |
| compression | CHARACTER VARYING(8) | Included for compatibility only; always NONE |
| num\_rows | NUMERIC | Same as pg\_class.reltuples. |
| blocks | INTEGER | Same as pg\_class.relpages. |
| empty\_blocks | NUMERIC | Included for compatibility only; always NULL |
| avg\_space | NUMERIC | Included for compatibility only; always NULL |
| chain\_cnt | NUMERIC | Included for compatibility only; always NULL |
| avg\_row\_len | NUMERIC | Included for compatibility only; always NULL |
| sample\_size | NUMERIC | Included for compatibility only; always NULL |
| last\_analyzed | TIMESTAMP WITHOUT TIME ZONE | Included for compatibility only; always NULL |
| buffer\_pool | CHARACTER VARYING(7) | Included for compatibility only; always NULL |
| global\_stats | CHARACTER VARYING(3) | Included for compatibility only; always YES |
| user\_stats | CHARACTER VARYING(3) | Included for compatibility only; always NO |
| backing\_table | REGCLASS | Name of the partition backing table. |

## ALL\_TAB\_SUBPARTITIONS

The ALL\_TAB\_SUBPARTITIONS view provides information about all of the subpartitions that reside in the database.

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| table\_owner | TEXT | The owner of the table in which the subpartition resides. |
| schema\_name | TEXT | The name of the schema in which the table resides. |
| table\_name | TEXT | The name of the table. |
| partition\_name | TEXT | The name of the partition. |
| subpartition\_name | TEXT | The name of the subpartition. |
| high\_value | TEXT | The high subpartitioning value specified in the CREATE TABLE statement. |
| high\_value\_length | INTEGER | The length of the subpartitioning value. |
| subpartition\_position | INTEGER | 1 for the first subpartition; 2 for the second subpartition, etc. |
| tablespace\_name | TEXT | The name of the tablespace in which the subpartition resides. |
| pct\_free | NUMERIC | Included for compatibility only; always 0 |
| pct\_used | NUMERIC | Included for compatibility only; always 0 |
| ini\_trans | NUMERIC | Included for compatibility only; always 0 |
| max\_trans | NUMERIC | Included for compatibility only; always 0 |
| initial\_extent | NUMERIC | Included for compatibility only; always NULL |
| next\_extent | NUMERIC | Included for compatibility only; always NULL |
| min\_extent | NUMERIC | Included for compatibility only; always 0 |
| max\_extent | NUMERIC | Included for compatibility only; always 0 |
| pct\_increase | NUMERIC | Included for compatibility only; always 0 |
| freelists | NUMERIC | Included for compatibility only; always NULL |
| freelist\_groups | NUMERIC | Included for compatibility only; always NULL |
| logging | CHARACTER VARYING(7) | Included for compatibility only; always YES |
| compression | CHARACTER VARYING(8) | Included for compatibility only; always NONE |
| num\_rows | NUMERIC | Same as pg\_class.reltuples. |
| blocks | INTEGER | Same as pg\_class.relpages. |
| empty\_blocks | NUMERIC | Included for compatibility only; always NULL |
| avg\_space | NUMERIC | Included for compatibility only; always NULL |
| chain\_cnt | NUMERIC | Included for compatibility only; always NULL |
| avg\_row\_len | NUMERIC | Included for compatibility only; always NULL |
| sample\_size | NUMERIC | Included for compatibility only; always NULL |
| last\_analyzed | TIMESTAMP WITHOUT TIME ZONE | Included for compatibility only; always NULL |
| buffer\_pool | CHARACTER VARYING(7) | Included for compatibility only; always NULL |
| global\_stats | CHARACTER VARYING(3) | Included for compatibility only; always YES |
| user\_stats | CHARACTER VARYING(3) | Included for compatibility only; always NO |
| backing\_table | REGCLASS | Name of the subpartition backing table. |

## ALL\_TABLES

The ALL\_TABLES view provides information on all user-defined tables.

| Name | Type | Description |
| --- | --- | --- |
| owner | TEXT | User name of the table’s owner. |
| schema\_name | TEXT | Name of the schema in which the table belongs. |
| table\_name | TEXT | Name of the table. |
| tablespace\_name | TEXT | Name of the tablespace in which the table resides if other than the default tablespace. |
| status | CHARACTER VARYING(5) | Whether or not the state of the table is valid. Currently, Included for compatibility only; always set to VALID. |
| temporary | CHARACTER(1) | Y if this is a temporary table; N if this is not a temporary table. |

## ALL\_TRIGGERS

The ALL\_TRIGGERS view provides information about the triggers on tables that may be accessed by the current user.

| Name | Type | Description |
| --- | --- | --- |
| owner | TEXT | User name of the trigger’s owner. |
| schema\_name | TEXT | The name of the schema in which the trigger resides. |
| trigger\_name | TEXT | The name of the trigger. |
| trigger\_type | TEXT | The type of the trigger. Possible values are:  BEFORE ROW  BEFORE STATEMENT  AFTER ROW  AFTER STATEMENT |
| triggering\_event | TEXT | The event that fires the trigger. |
| table\_owner | TEXT | The user name of the owner of the table on which the trigger is defined. |
| base\_object\_type | TEXT | Included for compatibility only. Value will always be TABLE. |
| table\_name | TEXT | The name of the table on which the trigger is defined. |
| referencing\_name | TEXT | Included for compatibility only. Value will always be REFERENCING NEW AS NEW OLD AS OLD. |
| status | TEXT | Status indicates if the trigger is enabled (VALID) or disabled (NOTVALID). |
| description | TEXT | Included for compatibility only. Value will always be SEE TRIGGER BODY FOR TEXT. |
| trigger\_body | TEXT | The body of the trigger. |
| action\_statement | TEXT | The SQL command that executes when the trigger fires. |

## ALL\_TYPES

The ALL\_TYPES view provides information about the object types available to the current user.

| Name | Type | Description |
| --- | --- | --- |
| owner | TEXT | The owner of the object type. |
| schema\_name | TEXT | The name of the schema in which the type is defined. |
| type\_name | TEXT | The name of the type. |
| type\_oid | OID | The object identifier (OID) of the type. |
| typecode | TEXT | The typecode of the type. Possible values are:  OBJECT  COLLECTION  OTHER |
| attributes | INTEGER | The number of attributes in the type. |

## ALL\_USERS

The ALL\_USERS view provides information on all user names.

| Name | Type | Description |
| --- | --- | --- |
| username | TEXT | Name of the user. |
| user\_id | OID | Numeric user id assigned to the user. |
| created | TIMESTAMP WITHOUT TIME ZONE | Included for compatibility only; always NULL. |

## ALL\_VIEW\_COLUMNS

The ALL\_VIEW\_COLUMNS view provides information on all columns in all user-defined views.

| Name | Type | Description |
| --- | --- | --- |
| owner | CHARACTER VARYING | User name of the view’s owner. |
| schema\_name | CHARACTER VARYING | Name of the schema in which the view belongs. |
| view\_name | CHARACTER VARYING | Name of the view. |
| column\_name | CHARACTER VARYING | Name of the column. |
| data\_type | CHARACTER VARYING | Data type of the column. |
| data\_length | NUMERIC | Length of text columns. |
| data\_precision | NUMERIC | Precision (number of digits) for NUMBER columns. |
| data\_scale | NUMERIC | Scale of NUMBER columns. |
| nullable | CHARACTER(1) | Whether or not the column is nullable – possible values are: Y – column is nullable; N – column does not allow null. |
| column\_id | NUMERIC | Relative position of the column within the view. |
| data\_default | CHARACTER VARYING | Default value assigned to the column. |

## ALL\_VIEWS

The ALL\_VIEWS view provides information about all user-defined views.

| Name | Type | Description |
| --- | --- | --- |
| owner | TEXT | User name of the view’s owner. |
| schema\_name | TEXT | Name of the schema in which the view belongs. |
| view\_name | TEXT | Name of the view. |
| text | TEXT | The SELECT statement that defines the view. |

## DBA\_ALL\_TABLES

The DBA\_ALL\_TABLES view provides information about all tables in the database.

| Name | Type | Description |
| --- | --- | --- |
| owner | TEXT | User name of the table’s owner. |
| schema\_name | TEXT | Name of the schema in which the table belongs. |
| table\_name | TEXT | Name of the table. |
| tablespace\_name | TEXT | Name of the tablespace in which the table resides if other than the default tablespace. |
| status | CHARACTER VARYING(5) | Included for compatibility only; always set to VALID. |
| temporary | TEXT | Y if the table is temporary; N if the table is permanent. |

## DBA\_CONS\_COLUMNS

The DBA\_CONS\_COLUMNS view provides information about all columns that are included in constraints that are specified in on all tables in the database.

| Name | Type | Description |
| --- | --- | --- |
| owner | TEXT | User name of the constraint’s owner. |
| schema\_name | TEXT | Name of the schema in which the constraint belongs. |
| constraint\_name | TEXT | The name of the constraint. |
| table\_name | TEXT | The name of the table to which the constraint belongs. |
| column\_name | TEXT | The name of the column referenced in the constraint. |
| position | SMALLINT | The position of the column within the object definition. |
| constraint\_def | TEXT | The definition of the constraint. |

## DBA\_CONSTRAINTS

The DBA\_CONSTRAINTS view provides information about all constraints on tables in the database.

| Name | Type | Description |
| --- | --- | --- |
| owner | TEXT | User name of the constraint’s owner. |
| schema\_name | TEXT | Name of the schema in which the constraint belongs. |
| constraint\_name | TEXT | The name of the constraint. |
| constraint\_type | TEXT | The constraint type. Possible values are:  C – check constraint  F – foreign key constraint  P – primary key constraint  U – unique key constraint  R – referential integrity constraint  V – constraint on a view  O – with read-only, on a view |
| table\_name | TEXT | Name of the table to which the constraint belongs. |
| search\_condition | TEXT | Search condition that applies to a check constraint. |
| r\_owner | TEXT | Owner of a table referenced by a referential constraint. |
| r\_constraint\_name | TEXT | Name of the constraint definition for a referenced table. |
| delete\_rule | TEXT | The delete rule for a referential constraint. Possible values are:  C – cascade  R - restrict  N – no action |
| deferrable | BOOLEAN | Specified if the constraint is deferrable (T or F). |
| deferred | BOOLEAN | Specifies if the constraint has been deferred (T or F). |
| index\_owner | TEXT | User name of the index owner. |
| index\_name | TEXT | The name of the index. |
| constraint\_def | TEXT | The definition of the constraint. |

## DBA\_DB\_LINKS

The DBA\_DB\_LINKS view provides information about all database links in the database.

| Name | Type | Description |
| --- | --- | --- |
| owner | TEXT | User name of the database link’s owner. |
| db\_link | TEXT | The name of the database link. |
| type | CHARACTER VARYING | Type of remote server. Value will be either REDWOOD or EDB |
| username | TEXT | User name of the user logging in. |
| host | TEXT | Name or IP address of the remote server. |

## DBA\_IND\_COLUMNS

The DBA\_IND\_COLUMNS view provides information about all columns included in indexes, on all tables in the database.

| Name | Type | Description |
| --- | --- | --- |
| index\_owner | TEXT | User name of the index’s owner. |
| schema\_name | TEXT | Name of the schema in which the index belongs. |
| index\_name | TEXT | Name of the index. |
| table\_owner | TEXT | User name of the table’s owner. |
| table\_name | TEXT | Name of the table in which the index belongs. |
| column\_name | TEXT | Name of column or attribute of object column. |
| column\_position | SMALLINT | The position of the column in the index. |
| column\_length | SMALLINT | The length of the column (in bytes). |
| char\_length | NUMERIC | The length of the column (in characters). |
| descend | CHARACTER(1) | Always set to Y (descending); included for compatibility only. |

## DBA\_INDEXES

The DBA\_INDEXES view provides information about all indexes in the database.

| Name | Type | Description |
| --- | --- | --- |
| owner | TEXT | User name of the index’s owner. |
| schema\_name | TEXT | Name of the schema in which the index resides. |
| index\_name | TEXT | The name of the index. |
| index\_type | TEXT | The index type is always BTREE. Included for compatibility only. |
| table\_owner | TEXT | User name of the owner of the indexed table. |
| table\_name | TEXT | The name of the indexed table. |
| table\_type | TEXT | Included for compatibility only. Always set to TABLE. |
| uniqueness | TEXT | Indicates if the index is UNIQUE or NONUNIQUE. |
| compression | CHARACTER(1) | Always set to N (not compressed). Included for compatibility only. |
| tablespace\_name | TEXT | Name of the tablespace in which the table resides if other than the default tablespace. |
| logging | TEXT | Included for compatibility only. Always set to LOGGING. |
| status | TEXT | Whether or not the state of the object is valid. (VALID or INVALID). |
| partitioned | CHARACTER(3) | Indicates that the index is partitioned. Always set to NO. |
| temporary | CHARACTER(1) | Indicates that an index is on a temporary table. Always set to N. |
| secondary | CHARACTER(1) | Included for compatibility only. Always set to N. |
| join\_index | CHARACTER(3) | Included for compatibility only. Always set to NO. |
| dropped | CHARACTER(3) | Included for compatibility only. Always set to NO. |

## DBA\_JOBS

The DBA\_JOBS view provides information about all jobs in the database.

| Name | Type | Description |
| --- | --- | --- |
| job | INTEGER | The identifier of the job (Job ID). |
| log\_user | TEXT | The name of the user that submitted the job. |
| priv\_user | TEXT | Same as log\_user. Included for compatibility only. |
| schema\_user | TEXT | The name of the schema used to parse the job. |
| last\_date | TIMESTAMP WITH TIME ZONE | The last date that this job executed successfully. |
| last\_sec | TEXT | Same as last\_date. |
| this\_date | TIMESTAMP WITH TIME ZONE | The date that the job began executing. |
| this\_sec | TEXT | Same as this\_date |
| next\_date | TIMESTAMP WITH TIME ZONE | The next date that this job will be executed. |
| next\_sec | TEXT | Same as next\_date. |
| total\_time | INTERVAL | The execution time of this job (in seconds). |
| broken | TEXT | If Y, no attempt will be made to run this job.  If N, this job will attempt to execute. |
| interval | TEXT | Determines how often the job will repeat. |
| failures | BIGINT | The number of times that the job has failed to complete since it’s last successful execution. |
| what | TEXT | The job definition (PL/SQL code block) that runs when the job executes. |
| nls\_env | CHARACTER VARYING(4000) | Always NULL. Provided for compatibility only. |
| misc\_env | BYTEA | Always NULL. Provided for compatibility only. |
| instance | NUMERIC | Always 0. Provided for compatibility only. |

## DBA\_OBJECTS

The DBA\_OBJECTS view provides information about all objects in the database.

| Name | Type | Description |
| --- | --- | --- |
| owner | TEXT | User name of the object’s owner. |
| schema\_name | TEXT | Name of the schema in which the object belongs. |
| object\_name | TEXT | Name of the object. |
| object\_type | TEXT | Type of the object – possible values are: INDEX, FUNCTION, PACKAGE, PACKAGE BODY, PROCEDURE, SEQUENCE, SYNONYM, TABLE, TRIGGER, and VIEW. |
| status | CHARACTER VARYING | Included for compatibility only; always set to VALID. |
| temporary | TEXT | Y if the table is temporary; N if the table is permanent. |

## DBA\_PART\_KEY\_COLUMNS

The DBA\_PART\_KEY\_COLUMNS view provides information about the key columns of the partitioned tables that reside in the database.

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| owner | TEXT | The owner of the table. |
| schema\_name | TEXT | The name of the schema in which the table resides. |
| name | TEXT | The name of the table in which the column resides. |
| object\_type | CHARACTER(5) | For compatibility only; always TABLE. |
| column\_name | TEXT | The name of the column on which the key is defined. |
| column\_position | INTEGER | 1 for the first column; 2 for the second column, etc. |

## DBA\_PART\_TABLES

The DBA\_PART\_TABLES view provides information about all of the partitioned tables in the database.

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| owner | TEXT | The owner of the partitioned table. |
| schema\_name | TEXT | The schema in which the table resides. |
| table\_name | TEXT | The name of the table. |
| partitioning\_type | TEXT | The type used to define table partitions. |
| subpartitioning\_type | TEXT | The subpartitioning type used to define table subpartitions. |
| partition\_count | BIGINT | The number of partitions in the table. |
| def\_subpartition\_count | INTEGER | The number of subpartitions in the table. |
| partitioning\_key\_count | INTEGER | The number of partitioning keys specified. |
| subpartitioning\_key\_count | INTEGER | The number of subpartitioning keys specified. |
| status | CHARACTER VARYING(8) | Provided for compatibility only. Always VALID. |
| def\_tablespace\_name | CHARACTER VARYING(30) | Provided for compatibility only. Always NULL. |
| def\_pct\_free | NUMERIC | Provided for compatibility only. Always NULL. |
| def\_pct\_used | NUMERIC | Provided for compatibility only. Always NULL. |
| def\_ini\_trans | NUMERIC | Provided for compatibility only. Always NULL. |
| def\_max\_trans | NUMERIC | Provided for compatibility only. Always NULL. |
| def\_initial\_extent | CHARACTER VARYING(40) | Provided for compatibility only. Always NULL. |
| def\_next\_extent | CHARACTER VARYING(40) | Provided for compatibility only. Always NULL. |
| def\_min\_extents | CHARACTER VARYING(40) | Provided for compatibility only. Always NULL. |
| def\_max\_extents | CHARACTER VARYING(40) | Provided for compatibility only. Always NULL. |
| def\_pct\_increase | CHARACTER VARYING(40) | Provided for compatibility only. Always NULL. |
| def\_freelists | NUMERIC | Provided for compatibility only. Always NULL. |
| def\_freelist\_groups | NUMERIC | Provided for compatibility only. Always NULL. |
| def\_logging | CHARACTER VARYING(7) | Provided for compatibility only. Always YES. |
| def\_compression | CHARACTER VARYING(8) | Provided for compatibility only. Always NONE. |
| def\_buffer\_pool | CHARACTER VARYING(7) | Provided for compatibility only. Always DEFAULT. |
| ref\_ptn\_constraint\_name | CHARACTER VARYING(30) | Provided for compatibility only. Always NULL. |
| interval | CHARACTER VARYING(1000) | Provided for compatibility only. Always NULL. |

## DBA\_POLICIES

The DBA\_POLICIES view provides information on all policies in the database. This view is accessible only to superusers.

| Name | Type | Description |
| --- | --- | --- |
| object\_owner | TEXT | Name of the owner of the object. |
| schema\_name | TEXT | The name of the schema in which the object resides. |
| object\_name | TEXT | Name of the object to which the policy applies. |
| policy\_group | TEXT | Name of the policy group. Included for compatibility only; always set to an empty string. |
| policy\_name | TEXT | Name of the policy. |
| pf\_owner | TEXT | Name of the schema containing the policy function, or the schema containing the package that contains the policy function. |
| package | TEXT | Name of the package containing the policy function (if the function belongs to a package). |
| function | TEXT | Name of the policy function. |
| sel | TEXT | Whether or not the policy applies to SELECT commands. Possible values are YES or NO. |
| ins | TEXT | Whether or not the policy applies to INSERT commands. Possible values are YES or NO. |
| upd | TEXT | Whether or not the policy applies to UPDATE commands. Possible values are YES or NO. |
| del | TEXT | Whether or not the policy applies to DELETE commands. Possible values are YES or NO. |
| idx | TEXT | Whether or not the policy applies to index maintenance. Possible values are YES or NO. |
| chk\_option | TEXT | Whether or not the check option is in force for INSERT and UPDATE commands. Possible values are YES or NO. |
| enable | TEXT | Whether or not the policy is enabled on the object. Possible values are YES or NO. |
| static\_policy | TEXT | Included for compatibility only; always set to NO. |
| policy\_type | TEXT | Included for compatibility only; always set to UNKNOWN. |
| long\_predicate | TEXT | Included for compatibility only; always set to YES. |

## DBA\_PROFILES

The DBA\_PROFILES view provides information about existing profiles. The table includes a row for each profile/resource combination.

| Name | Type | Description |
| --- | --- | --- |
| profile | CHARACTER VARYING(128) | The name of the profile. |
| resource\_name | CHARACTER VARYING(32) | The name of the resource associated with the profile. |
| resource\_type | CHARACTER VARYING(8) | The type of resource governed by the profile; currently PASSWORD for all supported resources. |
| limit | CHARACTER VARYING(128) | The limit values of the resource. |
| common | CHARACTER VARYING(3) | YES for a user-created profile; NO for a system-defined profile. |

## DBA\_ROLE\_PRIVS

The DBA\_ROLE\_PRIVS view provides information on all roles that have been granted to users. A row is created for each role to which a user has been granted.

| Name | Type | Description |
| --- | --- | --- |
| grantee | TEXT | User name to whom the role was granted. |
| granted\_role | TEXT | Name of the role granted to the grantee. |
| admin\_option | TEXT | YES if the role was granted with the admin option, NO otherwise. |
| default\_role | TEXT | YES if the role is enabled when the grantee creates a session. |

## DBA\_ROLES

The DBA\_ROLES view provides information on all roles with the NOLOGIN attribute (groups).

| Name | Type | Description |
| --- | --- | --- |
| role | TEXT | Name of a role having the NOLOGIN attribute – i.e., a group. |
| password\_required | TEXT | Included for compatibility only; always N. |

## DBA\_SEQUENCES

The DBA\_SEQUENCES view provides information about all user-defined sequences.

| Name | Type | Description |
| --- | --- | --- |
| sequence\_owner | TEXT | User name of the sequence's owner. |
| schema\_name | TEXT | The name of the schema in which the sequence resides. |
| sequence\_name | TEXT | Name of the sequence. |
| min\_value | NUMERIC | The lowest value that the server will assign to the sequence. |
| max\_value | NUMERIC | The highest value that the server will assign to the sequence. |
| increment\_by | NUMERIC | The value added to the current sequence number to create the next sequent number. |
| cycle\_flag | CHARACTER VARYING | Specifies if the sequence should wrap when it reaches min\_value or max\_value. |
| order\_flag | CHARACTER VARYING | This will always return Y. |
| cache\_size | NUMERIC | The number of pre-allocated sequence numbers stored in memory. |
| last\_number | NUMERIC | The value of the last sequence number saved to disk. |

## DBA\_SOURCE

The DBA\_SOURCE view provides the source code listing of all objects in the database.

| Name | Type | Description |
| --- | --- | --- |
| owner | TEXT | User name of the program’s owner. |
| schema\_name | TEXT | Name of the schema in which the program belongs. |
| name | TEXT | Name of the program. |
| type | TEXT | Type of program – possible values are: FUNCTION, PACKAGE, PACKAGE BODY, PROCEDURE, and TRIGGER. |
| line | INTEGER | Source code line number relative to a given program. |
| text | TEXT | Line of source code text. |

## DBA\_SUBPART\_KEY\_COLUMNS

The DBA\_SUBPART\_KEY\_COLUMNS view provides information about the key columns of those partitioned tables which are subpartitioned that reside in the database.

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| owner | TEXT | The owner of the table. |
| schema\_name | TEXT | The name of the schema in which the table resides. |
| name | TEXT | The name of the table in which the column resides. |
| object\_type | CHARACTER(5) | For compatibility only; always TABLE. |
| column\_name | TEXT | The name of the column on which the key is defined. |
| column\_position | INTEGER | 1 for the first column; 2 for the second column, etc. |

## DBA\_SYNONYMS

The DBA\_SYNONYM view provides information about all synonyms in the database.

| Name | Type | Description |
| --- | --- | --- |
| owner | TEXT | User name of the synonym’s owner. |
| schema\_name | TEXT | Name of the schema in which the synonym belongs. |
| synonym\_name | TEXT | Name of the synonym. |
| table\_owner | TEXT | User name of the table’s owner on which the synonym is defined. |
| table\_schema\_name | TEXT | The name of the schema in which the table resides. |
| table\_name | TEXT | Name of the table on which the synonym is defined. |
| db\_link | TEXT | Name of any associated database link. |

## DBA\_TAB\_COLUMNS

The DBA\_TAB\_COLUMNS view provides information about all columns in the database.

| Name | Type | Description |
| --- | --- | --- |
| owner | CHARACTER VARYING | User name of the owner of the table or view in which the column resides. |
| schema\_name | CHARACTER VARYING | Name of the schema in which the table or view resides. |
| table\_name | CHARACTER VARYING | Name of the table or view in which the column resides. |
| column\_name | CHARACTER VARYING | Name of the column. |
| data\_type | CHARACTER VARYING | Data type of the column. |
| data\_length | NUMERIC | Length of text columns. |
| data\_precision | NUMERIC | Precision (number of digits) for NUMBER columns. |
| data\_scale | NUMERIC | Scale of NUMBER columns. |
| nullable | CHARACTER(1) | Whether or not the column is nullable – possible values are:   Y – column is nullable; N – column does not allow null. |
| column\_id | NUMERIC | Relative position of the column within the table or view. |
| data\_default | CHARACTER VARYING | Default value assigned to the column. |

## DBA\_TAB\_PARTITIONS

The DBA\_TAB\_PARTITIONS view provides information about all of the partitions that reside in the database.

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| table\_owner | TEXT | The owner of the table in which the partition resides. |
| schema\_name | TEXT | The name of the schema in which the table resides. |
| table\_name | TEXT | The name of the table. |
| composite | TEXT | YES if the table is subpartitioned; NO if the table is not subpartitioned. |
| partition\_name | TEXT | The name of the partition. |
| subpartition\_count | BIGINT | The number of subpartitions in the partition. |
| high\_value | TEXT | The high partitioning value specified in the CREATE TABLE statement. |
| high\_value\_length | INTEGER | The length of the partitioning value. |
| partition\_position | INTEGER | 1 for the first partition; 2 for the second partition, etc. |
| tablespace\_name | TEXT | The name of the tablespace in which the partition resides. |
| pct\_free | NUMERIC | Included for compatibility only; always 0 |
| pct\_used | NUMERIC | Included for compatibility only; always 0 |
| ini\_trans | NUMERIC | Included for compatibility only; always 0 |
| max\_trans | NUMERIC | Included for compatibility only; always 0 |
| initial\_extent | NUMERIC | Included for compatibility only; always NULL |
| next\_extent | NUMERIC | Included for compatibility only; always NULL |
| min\_extent | NUMERIC | Included for compatibility only; always 0 |
| max\_extent | NUMERIC | Included for compatibility only; always 0 |
| pct\_increase | NUMERIC | Included for compatibility only; always 0 |
| freelists | NUMERIC | Included for compatibility only; always NULL |
| freelist\_groups | NUMERIC | Included for compatibility only; always NULL |
| logging | CHARACTER VARYING(7) | Included for compatibility only; always YES |
| compression | CHARACTER VARYING(8) | Included for compatibility only; always NONE |
| num\_rows | NUMERIC | Same as pg\_class.reltuples. |
| blocks | INTEGER | Same as pg\_class.relpages. |
| empty\_blocks | NUMERIC | Included for compatibility only; always NULL |
| avg\_space | NUMERIC | Included for compatibility only; always NULL |
| chain\_cnt | NUMERIC | Included for compatibility only; always NULL |
| avg\_row\_len | NUMERIC | Included for compatibility only; always NULL |
| sample\_size | NUMERIC | Included for compatibility only; always NULL |
| last\_analyzed | TIMESTAMP WITHOUT TIME ZONE | Included for compatibility only; always NULL |
| buffer\_pool | CHARACTER VARYING(7) | Included for compatibility only; always NULL |
| global\_stats | CHARACTER VARYING(3) | Included for compatibility only; always YES |
| user\_stats | CHARACTER VARYING(3) | Included for compatibility only; always NO |
| backing\_table | REGCLASS | Name of the partition backing table. |

## DBA\_TAB\_SUBPARTITIONS

The DBA\_TAB\_SUBPARTITIONS view provides information about all of the subpartitions that reside in the database.

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| table\_owner | TEXT | The owner of the table in which the subpartition resides. |
| schema\_name | TEXT | The name of the schema in which the table resides. |
| table\_name | TEXT | The name of the table. |
| partition\_name | TEXT | The name of the subpartition. |
| subpartition\_name | TEXT | The name of the subpartition. |
| high\_value | TEXT | The high subpartitioning value specified in the CREATE TABLE statement. |
| high\_value\_length | INTEGER | The length of the subpartitioning value. |
| subpartition\_position | INTEGER | 1 for the first subpartition; 2 for the second subpartition, etc. |
| tablespace\_name | TEXT | The name of the tablespace in which the subpartition resides. |
| pct\_free | NUMERIC | Included for compatibility only; always 0 |
| pct\_used | NUMERIC | Included for compatibility only; always 0 |
| ini\_trans | NUMERIC | Included for compatibility only; always 0 |
| max\_trans | NUMERIC | Included for compatibility only; always 0 |
| initial\_extent | NUMERIC | Included for compatibility only; always NULL |
| next\_extent | NUMERIC | Included for compatibility only; always NULL |
| min\_extent | NUMERIC | Included for compatibility only; always 0 |
| max\_extent | NUMERIC | Included for compatibility only; always 0 |
| pct\_increase | NUMERIC | Included for compatibility only; always 0 |
| freelists | NUMERIC | Included for compatibility only; always NULL |
| freelist\_groups | NUMERIC | Included for compatibility only; always NULL |
| logging | CHARACTER VARYING(7) | Included for compatibility only; always YES |
| compression | CHARACTER VARYING(8) | Included for compatibility only; always NONE |
| num\_rows | NUMERIC | Same as pg\_class.reltuples. |
| blocks | INTEGER | Same as pg\_class.relpages. |
| empty\_blocks | NUMERIC | Included for compatibility only; always NULL |
| avg\_space | NUMERIC | Included for compatibility only; always NULL |
| chain\_cnt | NUMERIC | Included for compatibility only; always NULL |
| avg\_row\_len | NUMERIC | Included for compatibility only; always NULL |
| sample\_size | NUMERIC | Included for compatibility only; always NULL |
| last\_analyzed | TIMESTAMP WITHOUT TIME ZONE | Included for compatibility only; always NULL |
| buffer\_pool | CHARACTER VARYING(7) | Included for compatibility only; always NULL |
| global\_stats | CHARACTER VARYING(3) | Included for compatibility only; always YES |
| user\_stats | CHARACTER VARYING(3) | Included for compatibility only; always NO |
| backing\_table | REGCLASS | Name of the subpartition backing table. |

## DBA\_TABLES

The DBA\_TABLES view provides information about all tables in the database.

| Name | Type | Description |
| --- | --- | --- |
| owner | TEXT | User name of the table’s owner. |
| schema\_name | TEXT | Name of the schema in which the table belongs. |
| table\_name | TEXT | Name of the table. |
| tablespace\_name | TEXT | Name of the tablespace in which the table resides if other than the default tablespace. |
| status | CHARACTER VARYING(5) | Included for compatibility only; always set to VALID. |
| temporary | CHARACTER(1) | Y if the table is temporary; N if the table is permanent. |

## DBA\_TRIGGERS

The DBA\_TRIGGERS view provides information about all triggers in the database.

| Name | Type | Description |
| --- | --- | --- |
| owner | TEXT | User name of the trigger’s owner. |
| schema\_name | TEXT | The name of the schema in which the trigger resides. |
| trigger\_name | TEXT | The name of the trigger. |
| trigger\_type | TEXT | The type of the trigger. Possible values are:  BEFORE ROW  BEFORE STATEMENT  AFTER ROW  AFTER STATEMENT |
| triggering\_event | TEXT | The event that fires the trigger. |
| table\_owner | TEXT | The user name of the owner of the table on which the trigger is defined. |
| base\_object\_type | TEXT | Included for compatibility only. Value will always be TABLE. |
| table\_name | TEXT | The name of the table on which the trigger is defined. |
| referencing\_names | TEXT | Included for compatibility only. Value will always be REFERENCING NEW AS NEW OLD AS OLD. |
| status | TEXT | Status indicates if the trigger is enabled (VALID) or disabled (NOTVALID). |
| description | TEXT | Included for compatibility only. Value will always be SEE TRIGGER BODY FOR TEXT. |
| trigger\_body | TEXT | The body of the trigger. |
| action\_statement | TEXT | The SQL command that executes when the trigger fires. |

## DBA\_TYPES

The DBA\_TYPES view provides information about all object types in the database.

| Name | Type | Description |
| --- | --- | --- |
| owner | TEXT | The owner of the object type. |
| schema\_name | TEXT | The name of the schema in which the type is defined. |
| type\_name | TEXT | The name of the type. |
| type\_oid | OID | The object identifier (OID) of the type. |
| typecode | TEXT | The typecode of the type. Possible values are:  OBJECT  COLLECTION  OTHER |
| attributes | INTEGER | The number of attributes in the type. |

## DBA\_USERS

The DBA\_USERS view provides information about all users of the database.

| Name | Type | Description |
| --- | --- | --- |
| username | TEXT | User name of the user. |
| user\_id | OID | ID number of the user. |
| password | CHARACTER VARYING(30) | The password (encrypted) of the user. |
| account\_status | CHARACTER VARYING(32) | The current status of the account. Possible values are:  OPEN EXPIRED EXPIRED(GRACE) EXPIRED & LOCKED  EXPIRED & LOCKED(TIMED)  EXPIRED(GRACE) & LOCKED EXPIRED(GRACE) & LOCKED(TIMED) LOCKED LOCKED(TIMED)  Use the edb\_get\_role\_status(*role\_id*) function to get the current status of the account. |
| lock\_date | TIMESTAMP WITHOUT TIME ZONE | If the account status is LOCKED, lock\_date displays the date and time the account was locked. |
| expiry\_date | TIMESTAMP WITHOUT TIME ZONE | The expiration date of the password. Use the edb\_get\_password\_expiry\_date(*role\_id*) function to get the current password expiration date. |
| default\_tablespace | TEXT | The default tablespace associated with the account. |
| temporary\_tablespace | CHARACTER VARYING(30) | Included for compatibility only. The value will always be '' (an empty string). |
| created | TIMESTAMP WITHOUT TIME ZONE | Included for compatibility only. The value is always NULL. |
| profile | CHARACTER VARYING(30) | The profile associated with the user. |
| initial\_rsrc\_consumer\_group | CHARACTER VARYING(30) | Included for compatibility only. The value is always NULL. |
| external\_name | CHARACTER VARYING(4000) | Included for compatibility only. The value is always NULL. |

## DBA\_VIEW\_COLUMNS

The DBA\_VIEW\_COLUMNS view provides information on all columns in the database.

| Name | Type | Description |
| --- | --- | --- |
| owner | CHARACTER VARYING | User name of the view’s owner. |
| schema\_name | CHARACTER VARYING | Name of the schema in which the view belongs. |
| view\_name | CHARACTER VARYING | Name of the view. |
| column\_name | CHARACTER VARYING | Name of the column. |
| data\_type | CHARACTER VARYING | Data type of the column. |
| data\_length | NUMERIC | Length of text columns. |
| data\_precision | NUMERIC | Precision (number of digits) for NUMBER columns. |
| data\_scale | NUMERIC | Scale of NUMBER columns. |
| nullable | CHARACTER(1) | Whether or not the column is nullable – possible values are:  Y – column is nullable; N – column does not allow null. |
| column\_id | NUMERIC | Relative position of the column within the view. |
| data\_default | CHARACTER VARYING | Default value assigned to the column. |

## DBA\_VIEWS

The DBA\_VIEWS view provides information about all views in the database.

| Name | Type | Description |
| --- | --- | --- |
| owner | TEXT | User name of the view’s owner. |
| schema\_name | TEXT | Name of the schema in which the view belongs. |
| view\_name | TEXT | Name of the view. |
| text | TEXT | The text of the SELECT statement that defines the view. |

## USER\_ALL\_TABLES

The USER\_ALL\_TABLES view provides information about all tables owned by the current user.

| Name | Type | Description |
| --- | --- | --- |
| schema\_name | TEXT | Name of the schema in which the table belongs. |
| table\_name | TEXT | Name of the table. |
| tablespace\_name | TEXT | Name of the tablespace in which the table resides if other than the default tablespace. |
| status | CHARACTER VARYING(5) | Included for compatibility only; always set to VALID.. |
| temporary | TEXT | Y if the table is temporary; N if the table is permanent. |

## USER\_CONS\_COLUMNS

The USER\_CONS\_COLUMNS view provides information about all columns that are included in constraints in tables that are owned by the current user.

| Name | Type | Description |
| --- | --- | --- |
| owner | TEXT | User name of the constraint’s owner. |
| schema\_name | TEXT | Name of the schema in which the constraint belongs. |
| constraint\_name | TEXT | The name of the constraint. |
| table\_name | TEXT | The name of the table to which the constraint belongs. |
| column\_name | TEXT | The name of the column referenced in the constraint. |
| position | SMALLINT | The position of the column within the object definition. |
| constraint\_def | TEXT | The definition of the constraint. |

## USER\_CONSTRAINTS

The USER\_CONSTRAINTS view provides information about all constraints placed on tables that are owned by the current user.

| Name | Type | Description |
| --- | --- | --- |
| owner | TEXT | The name of the owner of the constraint. |
| schema\_name | TEXT | Name of the schema in which the constraint belongs. |
| constraint\_name | TEXT | The name of the constraint. |
| constraint\_type | TEXT | The constraint type. Possible values are:  C – check constraint  F – foreign key constraint  P – primary key constraint  U – unique key constraint  R – referential integrity constraint  V – constraint on a view  O – with read-only, on a view |
| table\_name | TEXT | Name of the table to which the constraint belongs. |
| search\_condition | TEXT | Search condition that applies to a check constraint. |
| r\_owner | TEXT | Owner of a table referenced by a referential constraint. |
| r\_constraint\_name | TEXT | Name of the constraint definition for a referenced table. |
| delete\_rule | TEXT | The delete rule for a referential constraint. Possible values are:  C – cascade  R – restrict  N – no action |
| deferrable | BOOLEAN | Specified if the constraint is deferrable (T or F). |
| deferred | BOOLEAN | Specifies if the constraint has been deferred (T or F). |
| index\_owner | TEXT | User name of the index owner. |
| index\_name | TEXT | The name of the index. |
| constraint\_def | TEXT | The definition of the constraint. |

## USER\_DB\_LINKS

The USER\_DB\_LINKS view provides information about all database links that are owned by the current user.

| Name | Type | Description |
| --- | --- | --- |
| db\_link | TEXT | The name of the database link. |
| type | CHARACTER VARYING | Type of remote server. Value will be either REDWOOD or EDB |
| username | TEXT | User name of the user logging in. |
| password | TEXT | Password used to authenticate on the remote server. |
| host | TEXT | Name or IP address of the remote server. |

## USER\_IND\_COLUMNS

The USER\_IND\_COLUMNS view provides information about all columns referred to in indexes on tables that are owned by the current user.

| Name | Type | Description |
| --- | --- | --- |
| schema\_name | TEXT | Name of the schema in which the index belongs. |
| index\_name | TEXT | The name of the index. |
| table\_name | TEXT | The name of the table to which the index belongs. |
| column\_name | TEXT | The name of the column. |
| column\_position | SMALLINT | The position of the column within the index. |
| column\_length | SMALLINT | The length of the column (in bytes). |
| char\_length | NUMERIC | The length of the column (in characters). |
| descend | CHARACTER(1) | Always set to Y (descending); included for compatibility only. |

## USER\_INDEXES

The USER\_INDEXES view provides information about all indexes on tables that are owned by the current user.

| Name | Type | Description |
| --- | --- | --- |
| schema\_name | TEXT | Name of the schema in which the index belongs. |
| index\_name | TEXT | The name of the index. |
| index\_type | TEXT | Included for compatibility only. The index type is always BTREE. |
| table\_owner | TEXT | User name of the owner of the indexed table. |
| table\_name | TEXT | The name of the indexed table. |
| table\_type | TEXT | Included for compatibility only. Always set to TABLE. |
| uniqueness | TEXT | Indicates if the index is UNIQUE or NONUNIQUE. |
| compression | CHARACTER(1) | Included for compatibility only. Always set to N (not compressed). |
| tablespace\_name | TEXT | Name of the tablespace in which the table resides if other than the default tablespace. |
| logging | TEXT | Included for compatibility only. Always set to LOGGING. |
| status | TEXT | Whether or not the state of the object is valid. (VALID or INVALID). |
| partitioned | CHARACTER(3) | Included for compatibility only. Always set to NO. |
| temporary | CHARACTER(1) | Included for compatibility only. Always set to N. |
| secondary | CHARACTER(1) | Included for compatibility only. Always set to N. |
| join\_index | CHARACTER(3) | Included for compatibility only. Always set to NO. |
| dropped | CHARACTER(3) | Included for compatibility only. Always set to NO. |

## USER\_JOBS

The USER\_JOBS view provides information about all jobs owned by the current user.

| Name | Type | Description |
| --- | --- | --- |
| job | INTEGER | The identifier of the job (Job ID). |
| log\_user | TEXT | The name of the user that submitted the job. |
| priv\_user | TEXT | Same as log\_user. Included for compatibility only. |
| schema\_user | TEXT | The name of the schema used to parse the job. |
| last\_date | TIMESTAMP WITH TIME ZONE | The last date that this job executed successfully. |
| last\_sec | TEXT | Same as last\_date. |
| this\_date | TIMESTAMP WITH TIME ZONE | The date that the job began executing. |
| this\_sec | TEXT | Same as this\_date. |
| next\_date | TIMESTAMP WITH TIME ZONE | The next date that this job will be executed. |
| next\_sec | TEXT | Same as next\_date. |
| total\_time | INTERVAL | The execution time of this job (in seconds). |
| broken | TEXT | If Y, no attempt will be made to run this job.  If N, this job will attempt to execute. |
| interval | TEXT | Determines how often the job will repeat. |
| failures | BIGINT | The number of times that the job has failed to complete since it’s last successful execution. |
| what | TEXT | The job definition (PL/SQL code block) that runs when the job executes. |
| nls\_env | CHARACTER VARYING(4000) | Always NULL. Provided for compatibility only. |
| misc\_env | BYTEA | Always NULL. Provided for compatibility only. |
| instance | NUMERIC | Always 0. Provided for compatibility only. |

## USER\_OBJECTS

The USER\_OBJECTS view provides information about all objects that are owned by the current user.

| Name | Type | Description |
| --- | --- | --- |
| schema\_name | TEXT | Name of the schema in which the object belongs. |
| object\_name | TEXT | Name of the object. |
| object\_type | TEXT | Type of the object – possible values are: INDEX, FUNCTION, PACKAGE, PACKAGE BODY, PROCEDURE, SEQUENCE, SYNONYM, TABLE, TRIGGER, and VIEW. |
| status | CHARACTER VARYING | Included for compatibility only; always set to VALID. |
| temporary | TEXT | Y if the object is temporary; N if the object is not temporary. |

## USER\_PART\_KEY\_COLUMNS

The USER\_PART\_KEY\_COLUMNS view provides information about the key columns of the partitioned tables that reside in the database.

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| schema\_name | TEXT | The name of the schema in which the table resides. |
| name | TEXT | The name of the table in which the column resides. |
| object\_type | CHARACTER(5) | For compatibility only; always TABLE. |
| column\_name | TEXT | The name of the column on which the key is defined. |
| column\_position | INTEGER | 1 for the first column; 2 for the second column, etc. |

## USER\_PART\_TABLES

The USER\_PART\_TABLES view provides information about all of the partitioned tables in the database that are owned by the current user.

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| schema\_name | TEXT | The name of the schema in which the table resides. |
| table\_name | TEXT | The name of the table. |
| partitioning\_type | TEXT | The partitioning type used to define table partitions. |
| subpartitioning\_type | TEXT | The subpartitioning type used to define table subpartitions. |
| partition\_count | BIGINT | The number of partitions in the table. |
| def\_subpartition\_count | INTEGER | The number of subpartitions in the table. |
| partitioning\_key\_count | INTEGER | The number of partitioning keys specified. |
| subpartitioning\_key\_count | INTEGER | The number of subpartitioning keys specified. |
| status | CHARACTER VARYING(8) | Provided for compatibility only. Always VALID. |
| def\_tablespace\_name | CHARACTER VARYING(30) | Provided for compatibility only. Always NULL. |
| def\_pct\_free | NUMERIC | Provided for compatibility only. Always NULL. |
| def\_pct\_used | NUMERIC | Provided for compatibility only. Always NULL. |
| def\_ini\_trans | NUMERIC | Provided for compatibility only. Always NULL. |
| def\_max\_trans | NUMERIC | Provided for compatibility only. Always NULL. |
| def\_initial\_extent | CHARACTER VARYING(40) | Provided for compatibility only. Always NULL. |
| def\_min\_extents | CHARACTER VARYING(40) | Provided for compatibility only. Always NULL. |
| def\_max\_extents | CHARACTER VARYING(40) | Provided for compatibility only. Always NULL. |
| def\_pct\_increase | CHARACTER VARYING(40) | Provided for compatibility only. Always NULL. |
| def\_freelists | NUMERIC | Provided for compatibility only. Always NULL. |
| def\_freelist\_groups | NUMERIC | Provided for compatibility only. Always NULL. |
| def\_logging | CHARACTER VARYING(7) | Provided for compatibility only. Always YES. |
| def\_compression | CHARACTER VARYING(8) | Provided for compatibility only. Always NONE. |
| def\_buffer\_pool | CHARACTER VARYING(7) | Provided for compatibility only. Always DEFAULT. |
| ref\_ptn\_constraint\_name | CHARACTER VARYING(30) | Provided for compatibility only. Always NULL. |
| interval | CHARACTER VARYING(1000) | Provided for compatibility only. Always NULL. |

## USER\_POLICIES

The USER\_POLICIES view provides information on policies where the schema containing the object on which the policy applies has the same name as the current session user. This view is accessible only to superusers.

| Name | Type | Description |
| --- | --- | --- |
| schema\_name | TEXT | The name of the schema in which the object resides. |
| object\_name | TEXT | Name of the object on which the policy applies. |
| policy\_group | TEXT | Name of the policy group. Included for compatibility only; always set to an empty string. |
| policy\_name | TEXT | Name of the policy. |
| pf\_owner | TEXT | Name of the schema containing the policy function, or the schema containing the package that contains the policy function. |
| package | TEXT | Name of the package containing the policy function if the function belongs to a package. |
| function | TEXT | Name of the policy function. |
| sel | TEXT | Whether or not the policy applies to SELECT commands. Possible values are YES or NO. |
| ins | TEXT | Whether or not the policy applies to INSERT commands. Possible values are YES or NO. |
| upd | TEXT | Whether or not the policy applies to UPDATE commands. Possible values are YES or NO. |
| del | TEXT | Whether or not the policy applies to DELETE commands. Possible values are YES or NO. |
| idx | TEXT | Whether or not the policy applies to index maintenance. Possible values are YES or NO. |
| chk\_option | TEXT | Whether or not the check option is in force for INSERT and UPDATE commands. Possible values are YES or NO. |
| enable | TEXT | Whether or not the policy is enabled on the object. Possible values are YES or NO. |
| static\_policy | TEXT | Whether or not the policy is static. Included for compatibility only; always set to NO. |
| policy\_type | TEXT | Policy type. Included for compatibility only; always set to UNKNOWN. |
| long\_predicate | TEXT | Included for compatibility only; always set to YES. |

## USER\_ROLE\_PRIVS

The USER\_ROLE\_PRIVS view provides information about the privileges that have been granted to the current user. A row is created for each role to which a user has been granted.

| Name | Type | Description |
| --- | --- | --- |
| username | TEXT | The name of the user to which the role was granted. |
| granted\_role | TEXT | Name of the role granted to the grantee. |
| admin\_option | TEXT | YES if the role was granted with the admin option, NO otherwise. |
| default\_role | TEXT | YES if the role is enabled when the grantee creates a session. |
| os\_granted | CHARACTER VARYING(3) | Included for compatibility only; always NO. |

## USER\_SEQUENCES

The USER\_SEQUENCES view provides information about all user-defined sequences that belong to the current user.

| Name | Type | Description |
| --- | --- | --- |
| schema\_name | TEXT | The name of the schema in which the sequence resides. |
| sequence\_name | TEXT | Name of the sequence. |
| min\_value | NUMERIC | The lowest value that the server will assign to the sequence. |
| max\_value | NUMERIC | The highest value that the server will assign to the sequence. |
| increment\_by | NUMERIC | The value added to the current sequence number to create the next sequent number. |
| cycle\_flag | CHARACTER VARYING | Specifies if the sequence should wrap when it reaches min\_value or max\_value. |
| order\_flag | CHARACTER VARYING | Included for compatibility only; always Y. |
| cache\_size | NUMERIC | The number of pre-allocated sequence numbers in memory. |
| last\_number | NUMERIC | The value of the last sequence number saved to disk. |

## USER\_SOURCE

The USER\_SOURCE view provides information about all programs owned by the current user.

| Name | Type | Description |
| --- | --- | --- |
| schema\_name | TEXT | Name of the schema in which the program belongs. |
| name | TEXT | Name of the program. |
| type | TEXT | Type of program – possible values are: FUNCTION, PACKAGE, PACKAGE BODY, PROCEDURE, and TRIGGER. |
| line | INTEGER | Source code line number relative to a given program. |
| text | TEXT | Line of source code text. |

## USER\_SUBPART\_KEY\_COLUMNS

The USER\_SUBPART\_KEY\_COLUMNS view provides information about the key columns of those partitioned tables which are subpartitioned that belong to the current user.

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| schema\_name | TEXT | The name of the schema in which the table resides. |
| name | TEXT | The name of the table in which the column resides. |
| object\_type | CHARACTER(5) | For compatibility only; always TABLE. |
| column\_name | TEXT | The name of the column on which the key is defined. |
| column\_position | INTEGER | 1 for the first column; 2 for the second column, etc. |

## USER\_SYNONYMS

The USER\_SYNONYMS view provides information about all synonyms owned by the current user.

| Name | Type | Description |
| --- | --- | --- |
| schema\_name | TEXT | The name of the schema in which the synonym resides. |
| synonym\_name | TEXT | Name of the synonym. |
| table\_owner | TEXT | User name of the table’s owner on which the synonym is defined. |
| table\_schema\_name | TEXT | The name of the schema in which the table resides. |
| table\_name | TEXT | Name of the table on which the synonym is defined. |
| db\_link | TEXT | Name of any associated database link. |

## USER\_TAB\_COLUMNS

The USER\_TAB\_COLUMNS view displays information about all columns in tables and views owned by the current user.

| Name | Type | Description |
| --- | --- | --- |
| schema\_name | CHARACTER VARYING | Name of the schema in which the table or view resides. |
| table\_name | CHARACTER VARYING | Name of the table or view in which the column resides. |
| column\_name | CHARACTER VARYING | Name of the column. |
| data\_type | CHARACTER VARYING | Data type of the column. |
| data\_length | NUMERIC | Length of text columns. |
| data\_precision | NUMERIC | Precision (number of digits) for NUMBER columns. |
| data\_scale | NUMERIC | Scale of NUMBER columns. |
| nullable | CHARACTER(1) | Whether or not the column is nullable – possible values are: Y Y – column is nullable; N – column does not allow null. |
| column\_id | NUMERIC | Relative position of the column within the table. |
| data\_default | CHARACTER VARYING | Default value assigned to the column. |

## USER\_TAB\_PARTITIONS

The USER\_TAB\_PARTITIONS view provides information about all of the partitions that are owned by the current user.

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| schema\_name | TEXT | The name of the schema in which the table resides. |
| table\_name | TEXT | The name of the table. |
| composite | TEXT | YES if the table is subpartitioned; NO if the table is not subpartitioned. |
| partition\_name | TEXT | The name of the partition. |
| subpartition\_count | BIGINT | The number of subpartitions in the partition. |
| high\_value | TEXT | The high partitioning value specified in the CREATE TABLE statement. |
| high\_value\_length | INTEGER | The length of the partitioning value. |
| partition\_position | INTEGER | 1 for the first partition; 2 for the second partition, etc. |
| tablespace\_name | TEXT | The name of the tablespace in which the partition resides. |
| pct\_free | NUMERIC | Included for compatibility only; always 0 |
| pct\_used | NUMERIC | Included for compatibility only; always 0 |
| ini\_trans | NUMERIC | Included for compatibility only; always 0 |
| max\_trans | NUMERIC | Included for compatibility only; always 0 |
| initial\_extent | NUMERIC | Included for compatibility only; always NULL |
| next\_extent | NUMERIC | Included for compatibility only; always NULL |
| min\_extent | NUMERIC | Included for compatibility only; always 0 |
| max\_extent | NUMERIC | Included for compatibility only; always 0 |
| pct\_increase | NUMERIC | Included for compatibility only; always 0 |
| freelists | NUMERIC | Included for compatibility only; always NULL |
| freelist\_groups | NUMERIC | Included for compatibility only; always NULL |
| logging | CHARACTER VARYING(7) | Included for compatibility only; always YES |
| compression | CHARACTER VARYING(8) | Included for compatibility only; always NONE |
| num\_rows | NUMERIC | Same as pg\_class.reltuples. |
| blocks | INTEGER | Same as pg\_class.relpages. |
| empty\_blocks | NUMERIC | Included for compatibility only; always NULL |
| avg\_space | NUMERIC | Included for compatibility only; always NULL |
| chain\_cnt | NUMERIC | Included for compatibility only; always NULL |
| avg\_row\_len | NUMERIC | Included for compatibility only; always NULL |
| sample\_size | NUMERIC | Included for compatibility only; always NULL |
| last\_analyzed | TIMESTAMP WITHOUT TIME ZONE | Included for compatibility only; always NULL |
| buffer\_pool | CHARACTER VARYING(7) | Included for compatibility only; always NULL |
| global\_stats | CHARACTER VARYING(3) | Included for compatibility only; always YES |
| user\_stats | CHARACTER VARYING(3) | Included for compatibility only; always NO |
| backing\_table | REGCLASS | Name of the partition backing table. |

## USER\_TAB\_SUBPARTITIONS

The USER\_TAB\_SUBPARTITIONS view provides information about all of the subpartitions owned by the current user.

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| schema\_name | TEXT | The name of the schema in which the table resides. |
| table\_name | TEXT | The name of the table. |
| partition\_name | TEXT | The name of the subpartition. |
| subpartition\_name | TEXT | The name of the subpartition. |
| high\_value | TEXT | The high subpartitioning value specified in the CREATE TABLE statement. |
| high\_value\_length | INTEGER | The length of the subpartitioning value. |
| subpartition\_position | INTEGER | 1 for the first subpartition; 2 for the second subpartition, etc. |
| tablespace\_name | TEXT | The name of the tablespace in which the subpartition resides. |
| pct\_free | NUMERIC | Included for compatibility only; always 0 |
| pct\_used | NUMERIC | Included for compatibility only; always 0 |
| ini\_trans | NUMERIC | Included for compatibility only; always 0 |
| max\_trans | NUMERIC | Included for compatibility only; always 0 |
| initial\_extent | NUMERIC | Included for compatibility only; always NULL |
| next\_extent | NUMERIC | Included for compatibility only; always NULL |
| min\_extent | NUMERIC | Included for compatibility only; always 0 |
| max\_extent | NUMERIC | Included for compatibility only; always 0 |
| pct\_increase | NUMERIC | Included for compatibility only; always 0 |
| freelists | NUMERIC | Included for compatibility only; always NULL |
| freelist\_groups | NUMERIC | Included for compatibility only; always NULL |
| logging | CHARACTER VARYING(7) | Included for compatibility only; always YES |
| compression | CHARACTER VARYING(8) | Included for compatibility only; always NONE |
| num\_rows | NUMERIC | Same as pg\_class.reltuples. |
| blocks | INTEGER | Same as pg\_class.relpages. |
| empty\_blocks | NUMERIC | Included for compatibility only; always NULL |
| avg\_space | NUMERIC | Included for compatibility only; always NULL |
| chain\_cnt | NUMERIC | Included for compatibility only; always NULL |
| avg\_row\_len | NUMERIC | Included for compatibility only; always NULL |
| sample\_size | NUMERIC | Included for compatibility only; always NULL |
| last\_analyzed | TIMESTAMP WITHOUT TIME ZONE | Included for compatibility only; always NULL |
| buffer\_pool | CHARACTER VARYING(7) | Included for compatibility only; always NULL |
| global\_stats | CHARACTER VARYING(3) | Included for compatibility only; always YES |
| user\_stats | CHARACTER VARYING(3) | Included for compatibility only; always NO |
| backing\_table | REGCLASS | Name of the partition backing table. |

## USER\_TABLES

The USER\_TABLES view displays information about all tables owned by the current user.

| Name | Type | Description |
| --- | --- | --- |
| schema\_name | TEXT | Name of the schema in which the table belongs. |
| table\_name | TEXT | Name of the table. |
| tablespace\_name | TEXT | Name of the tablespace in which the table resides if other than the default tablespace. |
| status | CHARACTER VARYING(5) | Included for compatibility only; always set to VALID.. |
| temporary | CHARACTER(1) | Y if the table is temporary; N if the table is not temporary. |

## USER\_TRIGGERS

The USER\_TRIGGERS view displays information about all triggers on tables owned by the current user.

| Name | Type | Description |
| --- | --- | --- |
| schema\_name | TEXT | The name of the schema in which the trigger resides. |
| trigger\_name | TEXT | The name of the trigger. |
| trigger\_type | TEXT | The type of the trigger. Possible values are:  BEFORE ROW  BEFORE STATEMENT  AFTER ROW  AFTER STATEMENT |
| triggering\_event | TEXT | The event that fires the trigger. |
| table\_owner | TEXT | The user name of the owner of the table on which the trigger is defined. |
| base\_object\_type | TEXT | Included for compatibility only. Value will always be TABLE. |
| table\_name | TEXT | The name of the table on which the trigger is defined. |
| referencing\_names | TEXT | Included for compatibility only. Value will always be REFERENCING NEW AS NEW OLD AS OLD. |
| status | TEXT | Status indicates if the trigger is enabled (VALID) or disabled (NOTVALID). |
| description | TEXT | Included for compatibility only. Value will always be SEE TRIGGER BODY FOR TEXT. |
| trigger\_body | TEXT | The body of the trigger. |
| action\_statement | TEXT | The SQL command that executes when the trigger fires. |

## USER\_TYPES

The USER\_TYPES view provides information about all object types owned by the current user.

| Name | Type | Description |
| --- | --- | --- |
| schema\_name | TEXT | The name of the schema in which the type is defined. |
| type\_name | TEXT | The name of the type. |
| type\_oid | OID | The object identifier (OID) of the type. |
| typecode | TEXT | The typecode of the type. Possible values are:  OBJECT  COLLECTION  OTHER |
| attributes | INTEGER | The number of attributes in the type. |

## USER\_USERS

The USER\_USERS view provides information about the current user.

| Name | Type | Description |
| --- | --- | --- |
| username | TEXT | User name of the user. |
| user\_id | OID | ID number of the user. |
| account\_status | CHARACTER VARYING(32) | The current status of the account. Possible values are:  EXPIRED & LOCKED  OPEN  LOCKED |
| lock\_date | TIMESTAMP WITHOUT TIME ZONE | Included for compatibility only. The value is always NULL. |
| expiry\_date | TIMESTAMP WITHOUT TIME ZONE | The expiration date of the account. |
| default\_tablespace | CHARACTER VARYING(30) | The default tablespace associated with the account. |
| temporary\_tablespace | CHARACTER VARYING(30) | Included for compatibility only. The value will always be '' (an empty string). |
| created | TIMESTAMP WITHOUT TIME ZONE | Included for compatibility only. The value will always be NULL. |
| initial\_rsrc\_consumer\_group | CHARACTER VARYING(30) | Included for compatibility only. The value will always be NULL. |
| external\_name | CHARACTER VARYING(4000) | Included for compatibility only; always set to NULL. |

## USER\_VIEW\_COLUMNS

The USER\_VIEW\_COLUMNS view provides information about all columns in views owned by the current user.

| Name | Type | Description |
| --- | --- | --- |
| schema\_name | CHARACTER VARYING | Name of the schema in which the view belongs. |
| view\_name | CHARACTER VARYING | Name of the view. |
| column\_name | CHARACTER VARYING | Name of the column. |
| data\_type | CHARACTER VARYING | Data type of the column. |
| data\_length | NUMERIC | Length of text columns. |
| data\_precision | NUMERIC | Precision (number of digits) for NUMBER columns. |
| data\_scale | NUMERIC | Scale of NUMBER columns. |
| nullable | CHARACTER(1) | Whether or not the column is nullable – possible values are: Y – column is nullable; N – column does not allow null. |
| column\_id | NUMERIC | Relative position of the column within the view. |
| data\_default | CHARACTER VARYING | Default value assigned to the column. |

## USER\_VIEWS

The USER\_VIEWS view provides information about all views owned by the current user.

| Name | Type | Description |
| --- | --- | --- |
| schema\_name | TEXT | Name of the schema in which the view resides. |
| view\_name | TEXT | Name of the view. |
| text | TEXT | The SELECT statement that defines the view. |

## V$VERSION

The V$VERSION view provides information about product compatibility.

| Name | Type | Description |
| --- | --- | --- |
| banner | TEXT | Displays product compatibility information. |

## PRODUCT\_COMPONENT\_VERSION

The PRODUCT\_COMPONENT\_VERSION view provides version information about product version compatibility.

| Name | Type | Description |
| --- | --- | --- |
| product | CHARACTER VARYING(74) | The name of the product. |
| version | CHARACTER VARYING(74 | The version number of the product. |
| status | CHARACTER VARYING(74) | Included for compatibility; always Available. |

# System Catalog Tables

The following system catalog tables contain definitions of database objects. The layout of the system tables is subject to change; if you are writing an application that depends on information stored in the system tables, it would be prudent to use an existing catalog view, or create a catalog view to isolate the application from changes to the system table.

## dual

dual is a single-row, single-column table that is provided for compatibility with Oracle.

| Column | Type | Modifiers | Description |
| --- | --- | --- | --- |
| dummy | VARCHAR2(1) |  | Provided for compatibility only. |

## edb\_dir

The edb\_dir table contains one row for each alias that points to a directory created with the CREATE DIRECTORY command. A directory is an alias for a pathname that allows a user limited access to the host file system.

You can use a directory to fence a user into a specific directory tree within the file system. For example, the UTL\_FILE package offers functions that permit a user to read and write files and directories in the host file system, but only allows access to paths that the database administrator has granted access to via a CREATE DIRECTORY command.

| Column | Type | Modifiers | Description |
| --- | --- | --- | --- |
| dirname | "name" | not null | The name of the alias. |
| dirowner | oid | not null | The OID of the user that owns the alias. |
| dirpath | text |  | The directory name to which access is granted. |
| diracl | aclitem[] |  | The access control list that determines which users may access the alias. |

## edb\_all\_resource\_groups

The edb\_all\_resource\_groups table contains one row for each resource group created with the CREATE RESOURCE GROUP command and displays the number of active processes in each resource group.

| Column | Type | Modifiers | Description |
| --- | --- | --- | --- |
| group\_name | "name" |  | The name of the resource group. |
| active\_processes | integer |  | Number of currently active processes in the resource group. |
| cpu\_rate\_limit | float8 |  | Maximum CPU rate limit for the resource group. 0 means no limit. |
| per\_process\_cpu\_rate\_limit | float8 |  | Maximum CPU rate limit per currently active process in the resource group. |
| dirty\_rate\_limit | float8 |  | Maximum dirty rate limit for a resource group. 0 means no limit. |
| per\_process\_dirty\_rate\_limit | float8 |  | Maximum dirty rate limit per currently active process in the resource group. |

## edb\_partdef

The edb\_partdef table contains one row for each

| Column | Type | Modifiers | Description |
| --- | --- | --- | --- |
| pdefrel | oid | not null | The OID of the partitioning root (comes from pg\_class). |
| pdeftype | char | not null | The partitioning type:  'r' for range  'l' for list  'h' for hash. |
| pdefsubtype | char | not null | The subpartitioning type:  'r' for range  'l' for list  'h' for hash. |
| pdefcols | int2vector | not null | The partitioning key columns (a vector of pg\_attribute OIDs). |
| pdefsubcols | int2vector | not null | The subpartitioning key columns (a vector of pg\_attribute OIDs). |
| pdefkeyexpr | pg\_node\_tree |  | Currently unused. |
| pdefinsertexpr | pg\_node\_tree |  | Currently unused. |

## edb\_partition

The edb\_partition table contains one row for each partition or subpartition.

| Column | Type | Modifiers | Description |
| --- | --- | --- | --- |
| partname | name | not\_null | The partition or subpartition name. |
| partpos | integer | not\_null | The partition or subpartition position. |
| partpdefid | oid | not\_null | The OID of the edb\_partdef tuple (points to edb\_partdef). |
| partrelid | oid | not\_null | The OID of the partition backing table (points to pg\_class). |
| partparent | oid | not\_null | The OID of the parent edb\_partition tuple (for subpartitions). |
| partcons | oid | not\_null | The OID of the CHECK constraint for the partition (points to pg\_constraint). |
| parttablespace | oid | not\_null | The OID of the TABLESPACE (points to pg\_tablespace). |
| partistemplate | boolean | not\_null | Identifies this partition as a template partition (currently unused). |
| partvals | pg\_node\_tree |  | A list of partition key values in pg\_getexpr() form. |

## edb\_password\_history

The edb\_password\_history table contains one row for each password change. The table is shared across all databases within a cluster.

| Column | Type | References | Description |
| --- | --- | --- | --- |
| passhistroleid | oid | pg\_authid.oid | The ID of a role. |
| passhistpassword | text |  | Role password in md5 encrypted form. |
| passhistpasswordsetat | timestamptz |  | The time the password was set. |

## edb\_policy

The edb\_partition table contains one row for each policy.

| Column | Type | Modifiers | Description |
| --- | --- | --- | --- |
| policyname | name | not null | The policy name. |
| policygroup | oid | not null | Currently unused. |
| policyobject | oid | not null | The OID of the table secured by this policy (the object\_schema plus the object\_name). |
| policykind | char | not null | The kind of object secured by this policy:  'r' for a table  'v' for a view  = for a synonym  Currently always 'r'. |
| policyproc | oid | not null | The OID of the policy function (function\_schema plus policy\_function). |
| policyinsert | boolean | not null | True if the policy is enforced by INSERT statements. |
| policyselect | boolean | not null | True if the policy is enforced by SELECT statements. |
| policydelete | boolean | not null | True if the policy is enforced by DELETE statements. |
| policyupdate | boolean | not null | True if the policy is enforced by UPDATE statements. |
| policyindex | boolean | not null | Currently unused. |
| policyenabled | boolean | not null | True if the policy is enabled. |
| policyupdatecheck | boolean | not null | True if rows updated by an UPDATE statement must satisfy the policy. |
| policystatic | boolean | not null | Currently unused. |
| policytype | integer | not null | Currently unused. |
| policyopts | integer | not null | Currently unused. |
| policyseccols | int2vector | not null | The column numbers for columns listed in sec\_relevant\_cols. |

## edb\_profile

The edb\_profile table stores information about the available profiles. edb\_profiles is shared across all databases within a cluster.

| Column | Type | References | Description |
| --- | --- | --- | --- |
| oid | oid |  | Row identifier (hidden attribute; must be explicitly selected). |
| prfname | name |  | The name of the profile. |
| prffailedloginattempts | integer |  | The number of failed login attempts allowed by the profile. -1 indicates that the value from the default profile should be used. -2 indicates no limit on failed login attempts. |
| prfpasswordlocktime | integer |  | The password lock time associated with the profile (in seconds). -1 indicates that the value from the default profile should be used. -2 indicates that the account should be locked permanently. |
| prfpasswordlifetime | integer |  | The password life time associated with the profile (in seconds). -1 indicates that the value from the default profile should be used. -2 indicates that the password never expires. |
| prfpasswordgracetime | integer |  | The password grace time associated with the profile (in seconds). -1 indicates that the value from the default profile should be used. -2 indicates that the password never expires. |
| prfpasswordreusetime | integer |  | The number of seconds that a user must wait before reusing a password. -1 indicates that the value from the default profile should be used. -2 indicates that the old passwords can never be reused. |
| prfpasswordreusemax | integer |  | The number of password changes that have to occur before a password can be reused. -1 indicates that the value from the default profile should be used. -2 indicates that the old passwords can never be reused. |
| prfpasswordverifyfuncdb | oid | pg\_database.oid | The OID of the database in which the password verify function exists. |
| prfpasswordverifyfunc | oid | pg\_proc.oid | The OID of the password verify function associated with the profile. |

## edb\_resource\_group

The edb\_resource\_group table contains one row for each resource group created with the CREATE RESOURCE GROUP command.

| Column | Type | Modifiers | Description |
| --- | --- | --- | --- |
| rgrpname | "name" | not null | The name of the resource group. |
| rgrpcpuratelimit | float8 | not null | Maximum CPU rate limit for a resource group. 0 means no limit. |
| rgrpdirtyratelimit | float8 | not null | Maximum dirty rate limit for a resource group. 0 means no limit. |

## edb\_variable

The edb\_variable table contains one row for each package level variable (each variable declared within a package).

| Column | Type | Modifiers | Description |
| --- | --- | --- | --- |
| varname | "name" | not null | The name of the variable. |
| varpackage | oid | not null | The OID of the pg\_namespace row that stores the package. |
| vartype | oid | not null | The OID of the pg\_type row that defines the type of the variable. |
| varaccess | "char" | not null | + if the variable is visible outside of the package.  - if the variable is only visible within the package.  Note: Public variables are declared within the package header; private variables are declared within the package body. |
| varsrc | text |  | Contains the source of the variable declaration, including any default value expressions for the variable. |
| varseq | smallint | not null | The order in which the variable was declared in the package. |

## pg\_synonym

The pg\_synonym table contains one row for each synonym created with the CREATE SYNONYM command or CREATE PUBLIC SYNONYM command.

| Column | Type | Modifiers | Description |
| --- | --- | --- | --- |
| synname | "name" | not null | The name of the synonym. |
| synnamespace | oid | not null | Replaces synowner. Contains the OID of the pg\_namespace row where the synonym is stored |
| synowner | oid | not null | The OID of the user that owns the synonym. |
| synobjschema | "name" | not null | The schema in which the referenced object is defined. |
| synobjname | "name" | not null | The name of the referenced object. |
| synlink | text |  | The (optional) name of the database link in which the referenced object is defined. |

## product\_component\_version

The product\_component\_version table contains information about feature compatibility; an application can query this table at installation or run time to verify that features used by the application are available with this deployment.

| Column | Type | Description |
| --- | --- | --- |
| product | character varying (74) | The name of the product. |
| version | character varying (74) | The version number of the product. |
| status | character varying (74) | The status of the release. |

# Appendix

This chapter contains various miscellaneous topics.

## Advanced Server Database Limits

This section lists the Advanced Server database limits.

Table ‑ - Advanced Server Database Limits

| Limit | Value |
| --- | --- |
| Maximum Database Size | Unlimited |
| Maximum Table Size | 32 TB |
| Maximum Row Size | 1.6 TB |
| Maximum Field Size | 1 GB |
| Maximum Rows per Table | Unlimited |
| Maximum Columns per Table | 250 - 1600 depending on column types |
| Maximum Indexes per Table | Unlimited |

## Advanced Server Keywords

A keyword is a word that is recognized by the Advanced Server parser as having a special meaning or association. You can use the pg\_get\_keywords() function to retrieve an up-to-date list of the Advanced Server keywords:

acctg=#   
acctg=# SELECT \* FROM pg\_get\_keywords();  
 word | catcode | catdesc   
---------------------+---------+---------------------------------  
 abort | U | unreserved  
 absolute | U | unreserved  
 access | U | unreserved  
 action | U | unreserved  
 add | U | unreserved  
...

pg\_get\_keywords returns a table containing the keywords recognized by Advanced Server:

* The word column displays the keyword.
* The catcode column displays a category code.
* The catdesc column displays a brief description of the category to which the keyword belongs.

Note that any character can be used in an identifier if the name is enclosed in double quotes. You can selectively query the pg\_get\_keywords() function to retrieve an up-to-date list of the Advanced Server keywords that belong to a specific category:

SELECT \* FROM pg\_get\_keywords() WHERE catcode = *'code'*;

Where *code* is:

R - The word is reserved. Reserved keywords may never be used as an identifier; they are reserved for use by the server.

U - The word is unreserved. Unreserved words are used internally in some contexts, but may be used as a name for a database object.

T - The word is used internally, but may be used as a name for a function or type.

C - The word is used internally, and may not be used as a name for a function or type.

For more information about Advanced Server identifiers and keywords, please refer to the PostgreSQL core documentation at:

<http://www.postgresql.org/docs/9.5/static/sql-syntax-lexical.html>