# Table of Contents

1. **Introduction** .......................................................... 16
   1.1 What’s New .......................................................... 17
   1.2 Typographical Conventions Used in this Guide ................. 18
   1.3 Oracle Compatible Configuration Parameters .................... 19
       1.3.1 edb_redwood_date ........................................ 19
       1.3.2 edb_redwood_strings ..................................... 20
       1.3.3 edb_stmt_level_tx ...................................... 21
       1.3.4 oracle_home ............................................ 22
   1.4 About the Examples Used in this Guide ........................... 23

2. **SQL Tutorial** .......................................................... 24
   2.1 Getting Started ....................................................... 24
       2.1.1 Sample Database .......................................... 25
           2.1.1.1 Sample Database Installation ....................... 25
           2.1.1.2 Sample Database Description ...................... 25
       2.1.2 Creating a New Table ....................................... 36
       2.1.3 Populating a Table With Rows .............................. 37
       2.1.4 Querying a Table ........................................... 38
       2.1.5 Joins Between Tables ....................................... 40
       2.1.6 Aggregate Functions ....................................... 44
       2.1.7 Updates ...................................................... 46
       2.1.8 Deletions ..................................................... 47
   2.2 Advanced Concepts .................................................. 48
       2.2.1 Views ........................................................ 48
       2.2.2 Foreign Keys ............................................... 49
       2.2.3 The ROWNUM Pseudo-Column ............................... 50
       2.2.4 Synonyms .................................................... 51
       2.2.5 Hierarchical Queries ....................................... 55
           2.2.5.1 Defining the Parent/Child Relationship ............ 55
           2.2.5.2 Selecting the Root Nodes ............................ 56
           2.2.5.3 Organization Tree in the Sample Application ....... 56
           2.2.5.4 Node Level ........................................... 58
           2.2.5.5 Ordering the Siblings ............................... 59

3. **The SQL Language** .................................................... 61
   3.1 SQL Syntax .......................................................... 61
       3.1.1 Lexical Structure ........................................... 61
       3.1.2 Identifiers and Key Words ................................ 62
       3.1.3 Constants ................................................... 64
           3.1.3.1 String Constants .................................... 64
           3.1.3.2 Numeric Constants .................................. 64
           3.1.3.3 Constants of Other Types ............................ 65
       3.1.4 Comments .................................................... 65
   3.2 Data Types ........................................................... 66
       3.2.1 Numeric Types ............................................... 67
           3.2.1.1 Integer Types ....................................... 67
           3.2.1.2 Arbitrary Precision Numbers ......................... 67
           3.2.1.3 Floating-Point Types ................................ 68
       3.2.2 Character Types ............................................... 70
       3.2.3 Binary Data .................................................. 72
   3.2.4 Date/Time Types .................................................. 73
       3.2.4.1 INTERVAL Types ....................................... 74
       3.2.4.2 Date/Time Input ....................................... 75
           3.2.4.2.1 Dates .............................................. 75
<table>
<thead>
<tr>
<th>Section</th>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2.4.2.2</td>
<td>Times</td>
<td>76</td>
</tr>
<tr>
<td>3.2.4.2.3</td>
<td>Time Stamps</td>
<td>76</td>
</tr>
<tr>
<td>3.2.4.3</td>
<td>Date/Time Output</td>
<td>77</td>
</tr>
<tr>
<td>3.2.4.4</td>
<td>Internals</td>
<td>77</td>
</tr>
<tr>
<td>3.2.5</td>
<td>Boolean Type</td>
<td>78</td>
</tr>
<tr>
<td>3.2.6</td>
<td>XML Type</td>
<td>79</td>
</tr>
<tr>
<td>3.3</td>
<td>SQL Commands</td>
<td>80</td>
</tr>
<tr>
<td>3.3.1</td>
<td>ALTER INDEX</td>
<td>81</td>
</tr>
<tr>
<td>3.3.2</td>
<td>ALTER PROCEDURE</td>
<td>82</td>
</tr>
<tr>
<td>3.3.3</td>
<td>ALTER ROLE</td>
<td>83</td>
</tr>
<tr>
<td>3.3.4</td>
<td>ALTER SEQUENCE</td>
<td>87</td>
</tr>
<tr>
<td>3.3.5</td>
<td>ALTER SESSION</td>
<td>89</td>
</tr>
<tr>
<td>3.3.6</td>
<td>ALTER TABLE</td>
<td>91</td>
</tr>
<tr>
<td>3.3.7</td>
<td>ALTER TABLESPACE</td>
<td>95</td>
</tr>
<tr>
<td>3.3.8</td>
<td>ALTER USER</td>
<td>96</td>
</tr>
<tr>
<td>3.3.9</td>
<td>CALL</td>
<td>97</td>
</tr>
<tr>
<td>3.3.10</td>
<td>COMMENT</td>
<td>98</td>
</tr>
<tr>
<td>3.3.11</td>
<td>COMMIT</td>
<td>100</td>
</tr>
<tr>
<td>3.3.12</td>
<td>CREATE DATABASE</td>
<td>101</td>
</tr>
<tr>
<td>3.3.13</td>
<td>CREATE [PUBLIC] DATABASE LINK</td>
<td>102</td>
</tr>
<tr>
<td>3.3.14</td>
<td>CREATE DIRECTORY</td>
<td>107</td>
</tr>
<tr>
<td>3.3.15</td>
<td>CREATE FUNCTION</td>
<td>109</td>
</tr>
<tr>
<td>3.3.16</td>
<td>CREATE INDEX</td>
<td>113</td>
</tr>
<tr>
<td>3.3.17</td>
<td>CREATE PACKAGE</td>
<td>115</td>
</tr>
<tr>
<td>3.3.18</td>
<td>CREATE PACKAGE BODY</td>
<td>118</td>
</tr>
<tr>
<td>3.3.19</td>
<td>CREATE PROCEDURE</td>
<td>121</td>
</tr>
<tr>
<td>3.3.20</td>
<td>CREATE ROLE</td>
<td>125</td>
</tr>
<tr>
<td>3.3.21</td>
<td>CREATE SCHEMA</td>
<td>127</td>
</tr>
<tr>
<td>3.3.22</td>
<td>CREATE SEQUENCE</td>
<td>129</td>
</tr>
<tr>
<td>3.3.23</td>
<td>CREATE SYNONYM</td>
<td>132</td>
</tr>
<tr>
<td>3.3.24</td>
<td>CREATE TABLE</td>
<td>134</td>
</tr>
<tr>
<td>3.3.25</td>
<td>CREATE TABLE AS</td>
<td>142</td>
</tr>
<tr>
<td>3.3.26</td>
<td>CREATE TRIGGER</td>
<td>144</td>
</tr>
<tr>
<td>3.3.27</td>
<td>CREATE TYPE</td>
<td>147</td>
</tr>
<tr>
<td>3.3.28</td>
<td>CREATE TYPE BODY</td>
<td>152</td>
</tr>
<tr>
<td>3.3.29</td>
<td>CREATE USER</td>
<td>156</td>
</tr>
<tr>
<td>3.3.30</td>
<td>CREATE VIEW</td>
<td>157</td>
</tr>
<tr>
<td>3.3.31</td>
<td>DELETE</td>
<td>159</td>
</tr>
<tr>
<td>3.3.32</td>
<td>DROP DATABASE LINK</td>
<td>162</td>
</tr>
<tr>
<td>3.3.33</td>
<td>DROP DIRECTORY</td>
<td>163</td>
</tr>
<tr>
<td>3.3.34</td>
<td>DROP FUNCTION</td>
<td>164</td>
</tr>
<tr>
<td>3.3.35</td>
<td>DROP INDEX</td>
<td>166</td>
</tr>
<tr>
<td>3.3.36</td>
<td>DROP PACKAGE</td>
<td>167</td>
</tr>
<tr>
<td>3.3.37</td>
<td>DROP PROCEDURE</td>
<td>168</td>
</tr>
<tr>
<td>3.3.38</td>
<td>DROP SYNONYM</td>
<td>169</td>
</tr>
<tr>
<td>3.3.39</td>
<td>DROP ROLE</td>
<td>170</td>
</tr>
<tr>
<td>3.3.40</td>
<td>DROP SEQUENCE</td>
<td>172</td>
</tr>
<tr>
<td>3.3.41</td>
<td>DROP TABLE</td>
<td>173</td>
</tr>
<tr>
<td>3.3.42</td>
<td>DROP TABLESPACE</td>
<td>175</td>
</tr>
<tr>
<td>3.3.43</td>
<td>DROP TRIGGER</td>
<td>176</td>
</tr>
<tr>
<td>3.3.44</td>
<td>DROP TYPE</td>
<td>177</td>
</tr>
<tr>
<td>3.3.45</td>
<td>DROP USER</td>
<td>178</td>
</tr>
<tr>
<td>3.3.46</td>
<td>DROP VIEW</td>
<td>180</td>
</tr>
<tr>
<td>3.3.47</td>
<td>EXEC</td>
<td>181</td>
</tr>
<tr>
<td>3.3.48</td>
<td>GRANT</td>
<td>182</td>
</tr>
<tr>
<td>3.3.49</td>
<td>GRANT on Database Objects</td>
<td>184</td>
</tr>
</tbody>
</table>
3.5.0  GRANT on Roles ................................................................. 186
3.5.1  GRANT on System Privileges ............................................ 189
3.5.2  INSERT ........................................................................ 191
3.5.3  LOCK ........................................................................ 195
3.5.4  REVOKE ..................................................................... 198
3.5.5  ROLLBACK ................................................................. 202
3.5.6  ROLLBACK TO SAVEPOINT .......................................... 203
3.5.7  SAVEPOINT ................................................................. 204
3.5.8  SELECT .................................................................... 206
3.5.8.1 FROM Clause ............................................................. 207
3.5.8.2 WHERE Clause ........................................................... 209
3.5.8.3 GROUP BY Clause ....................................................... 210
3.5.8.4 HAVING Clause ......................................................... 210
3.5.8.5 SELECT List ............................................................... 210
3.5.8.6 UNION Clause ........................................................... 211
3.5.8.7 INTERSECT Clause ..................................................... 211
3.5.8.8 MINUS Clause .......................................................... 212
3.5.8.9 CONNECT BY Clause .................................................. 212
3.5.8.10 ORDER BY Clause ..................................................... 213
3.5.8.11 DISTINCT Clause ..................................................... 214
3.5.8.12 FOR UPDATE Clause ................................................. 214
3.5.9  SET CONSTRAINTS ....................................................... 217
3.5.10 SET ROLE ................................................................. 218
3.5.11 SET TRANSACTION ..................................................... 220
3.5.12 TRUNCATE ................................................................. 221
3.5.13 UPDATE ................................................................. 222
3.4  Optimizer Hints ............................................................... 225
3.4.1  Default Optimization Modes ........................................... 227
3.4.2  Access Method Hints ..................................................... 229
3.4.3  Specifying a Join Order .................................................. 233
3.4.4  Joining Relations Hints ................................................... 234
3.4.5  Global Hints ............................................................... 237
3.4.6  Using the APPEND Optimizer Hint ................................... 240
3.4.7  Conflicting Hints ........................................................ 241
3.5  Functions and Operators .................................................... 242
3.5.1  Logical Operators ........................................................ 242
3.5.2  Comparison Operators .................................................... 243
3.5.3  Mathematical Functions and Operators ............................. 245
3.5.4  String Functions and Operators ....................................... 247
3.5.5  Pattern Matching Using the LIKE Operator ....................... 250
3.5.6  Data Type Formatting Functions ..................................... 251
3.5.7  Date/Time Functions and Operators .................................. 256
3.5.7.1 ADD_MONTHS ............................................................ 257
3.5.7.2 EXTRACT ................................................................. 258
3.5.7.3 MONTHS_BETWEEN ................................................... 259
3.5.7.4 NEXT_DAY ............................................................. 260
3.5.7.5 NEW_TIME ........................................................... 260
3.5.7.6 ROUND ............................................................... 261
3.5.7.7 TRUNC ................................................................. 266
3.5.7.8 CURRENT_DATE/TIME .............................................. 269
3.5.8  Sequence Manipulation Functions ..................................... 271
3.5.9  Conditional Expressions .................................................. 272
3.5.9.1 CASE ................................................................. 272
3.5.9.2 COALESCE ............................................................ 273
3.5.9.3 NULLIF ............................................................... 274
3.5.9.4 NVL ................................................................. 274
### 4.11 Collection Methods

- **4.11.1 COUNT** ................................................................. 405
- **4.11.2 DELETE** .................................................................. 405
- **4.11.3 EXISTS** .................................................................. 407
- **4.11.4 EXTEND** .................................................................. 407
- **4.11.5 FIRST** ..................................................................... 409
- **4.11.6 LAST** ..................................................................... 410
- **4.11.7 LIMIT** ..................................................................... 410
- **4.11.8 NEXT** ..................................................................... 411
- **4.11.9 PRIOR** ..................................................................... 411
- **4.11.10 TRIM** ..................................................................... 412

### 4.12 Working with Collections

- **4.12.1 TABLE()** ................................................................. 414
- **4.12.2 Using the MULTiset UNION OPERator** ..................... 414
- **4.12.3 Using the FORALL Statement** ................................. 416
- **4.12.4 Using the BULK COLLECT Clause** ............................ 419
  - **4.12.4.1 SELECT BULK COLLECT** ...................................... 419
  - **4.12.4.2 FETCH BULK COLLECT** ...................................... 421
  - **4.12.4.3 EXECUTE IMMEDIATE BULK COLLECT** ............... 422
  - **4.12.4.4 RETURNING BULK COLLECT** ............................ 423

### 4.13 Errors and Messages

- **4.13** Errors and Messages .................................................. 425

### 5 Triggers

- **5.1 Overview** ..................................................................... 426
- **5.2 Types of Triggers** ........................................................ 426
- **5.3 Creating Triggers** .......................................................... 427
- **5.4 Trigger Variables** .......................................................... 429
- **5.5 Transactions and Exceptions** ......................................... 430
- **5.6 Trigger Examples** .......................................................... 431
  - **5.6.1 Before Statement-Level Trigger** .............................. 431
  - **5.6.2 After Statement-Level Trigger** ............................... 431
  - **5.6.3 Before Row-Level Trigger** ...................................... 432
  - **5.6.4 After Row-Level Trigger** ........................................ 433

### 6 Packages

- **6.1 Package Components** .................................................. 436
  - **6.1.1 Package Specification Syntax** .................................. 437
  - **6.1.2 Package Body Syntax** ............................................ 438
- **6.2 Creating Packages** ....................................................... 444
  - **6.2.1 Creating the Package Specification** ......................... 441
  - **6.2.2 Creating the Package Body** .................................... 441
- **6.3 Referencing a Package** ................................................ 443
- **6.4 Using Packages With User Defined Types** .................... 444
- **6.5 Dropping a Package** .................................................... 447

### 7 Built-In Packages

- **7.1 DBMS_ALERT** ............................................................. 448
  - **7.1.1 REGISTER** ........................................................... 449
  - **7.1.2 REMOVE** ............................................................. 450
  - **7.1.3 REMOVEALL** ........................................................ 450
  - **7.1.4 SIGNAL** ............................................................... 450
  - **7.1.5 WAITANY** ............................................................ 451
  - **7.1.6 WAITONE** ............................................................. 452
  - **7.1.7 Comprehensive Example** ....................................... 453
- **7.2 DBMS_JOB** ............................................................... 456
  - **7.2.1 BROKEN** ............................................................. 457
  - **7.2.2 CHANGE** ............................................................. 458
  - **7.2.3 INTERVAL** ............................................................. 459
  - **7.2.4 NEXT_DATE** ......................................................... 459
<table>
<thead>
<tr>
<th>Section</th>
<th>Function Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.8.3</td>
<td>BIND VARIABLE RAW</td>
<td>517</td>
</tr>
<tr>
<td>7.8.4</td>
<td>CLOSE_CURSOR</td>
<td>518</td>
</tr>
<tr>
<td>7.8.5</td>
<td>COLUMN_VALUE</td>
<td>518</td>
</tr>
<tr>
<td>7.8.6</td>
<td>COLUMN_VALUE_CHAR</td>
<td>520</td>
</tr>
<tr>
<td>7.8.7</td>
<td>COLUMN_VALUE_RAW</td>
<td>520</td>
</tr>
<tr>
<td>7.8.8</td>
<td>DEFINE_COLUMN</td>
<td>521</td>
</tr>
<tr>
<td>7.8.9</td>
<td>DEFINE_COLUMN_CHAR</td>
<td>523</td>
</tr>
<tr>
<td>7.8.10</td>
<td>DEFINE_COLUMN_RAW</td>
<td>523</td>
</tr>
<tr>
<td>7.8.11</td>
<td>DESCRIBE_COLUMNS</td>
<td>524</td>
</tr>
<tr>
<td>7.8.12</td>
<td>EXECUTE</td>
<td>525</td>
</tr>
<tr>
<td>7.8.13</td>
<td>EXECUTE_AND_FETCH</td>
<td>525</td>
</tr>
<tr>
<td>7.8.14</td>
<td>FETCH_ROWS</td>
<td>527</td>
</tr>
<tr>
<td>7.8.15</td>
<td>IS_OPEN</td>
<td>529</td>
</tr>
<tr>
<td>7.8.16</td>
<td>LAST_ROW_COUNT</td>
<td>529</td>
</tr>
<tr>
<td>7.8.17</td>
<td>OPEN_CURSOR</td>
<td>530</td>
</tr>
<tr>
<td>7.8.18</td>
<td>PARSE</td>
<td>531</td>
</tr>
<tr>
<td>7.9</td>
<td>DBMS_UTILITIES</td>
<td>533</td>
</tr>
<tr>
<td>7.9.1</td>
<td>LNAME_ARRAY</td>
<td>534</td>
</tr>
<tr>
<td>7.9.2</td>
<td>UNCL_ARRAY</td>
<td>534</td>
</tr>
<tr>
<td>7.9.3</td>
<td>ANALYZE_DATABASE, ANALYZE_SCHEMA and ANALYZE_PART_OBJECT</td>
<td>534</td>
</tr>
<tr>
<td>7.9.4</td>
<td>CANONICALIZE</td>
<td>536</td>
</tr>
<tr>
<td>7.9.5</td>
<td>COMMA_TO_TABLE</td>
<td>538</td>
</tr>
<tr>
<td>7.9.6</td>
<td>DB_VERSION</td>
<td>539</td>
</tr>
<tr>
<td>7.9.7</td>
<td>EXEC_DDL_STATEMENT</td>
<td>540</td>
</tr>
<tr>
<td>7.9.8</td>
<td>FORMAT_CALL_STACK</td>
<td>540</td>
</tr>
<tr>
<td>7.9.9</td>
<td>GET_CPU_TIME</td>
<td>541</td>
</tr>
<tr>
<td>7.9.10</td>
<td>GET_DEPENDENCY</td>
<td>541</td>
</tr>
<tr>
<td>7.9.11</td>
<td>GET_HASH_VALUE</td>
<td>542</td>
</tr>
<tr>
<td>7.9.12</td>
<td>GET_PARAMETER_VALUE</td>
<td>543</td>
</tr>
<tr>
<td>7.9.13</td>
<td>GET_TIME</td>
<td>544</td>
</tr>
<tr>
<td>7.9.14</td>
<td>NAME_TOKENIZE</td>
<td>545</td>
</tr>
<tr>
<td>7.9.15</td>
<td>TABLE_TO_COMMA</td>
<td>547</td>
</tr>
<tr>
<td>7.10</td>
<td>UTL_FILE</td>
<td>549</td>
</tr>
<tr>
<td>7.10.1</td>
<td>FCLOSE</td>
<td>550</td>
</tr>
<tr>
<td>7.10.2</td>
<td>FCLOSE_ALL</td>
<td>550</td>
</tr>
<tr>
<td>7.10.3</td>
<td>FCOPY</td>
<td>550</td>
</tr>
<tr>
<td>7.10.4</td>
<td>FFLUSH</td>
<td>552</td>
</tr>
<tr>
<td>7.10.5</td>
<td>FOPEN</td>
<td>553</td>
</tr>
<tr>
<td>7.10.6</td>
<td>FREMOVE</td>
<td>554</td>
</tr>
<tr>
<td>7.10.7</td>
<td>FRENNAME</td>
<td>554</td>
</tr>
<tr>
<td>7.10.8</td>
<td>GET_LINE</td>
<td>556</td>
</tr>
<tr>
<td>7.10.9</td>
<td>IS_OPEN</td>
<td>557</td>
</tr>
<tr>
<td>7.10.10</td>
<td>NEW_LINE</td>
<td>558</td>
</tr>
<tr>
<td>7.10.11</td>
<td>PUT</td>
<td>559</td>
</tr>
<tr>
<td>7.10.12</td>
<td>PUT_LINE</td>
<td>560</td>
</tr>
<tr>
<td>7.10.13</td>
<td>PUTF</td>
<td>562</td>
</tr>
<tr>
<td>7.11</td>
<td>UTL_MAIL</td>
<td>564</td>
</tr>
<tr>
<td>7.11.1</td>
<td>SEND</td>
<td>564</td>
</tr>
<tr>
<td>7.11.2</td>
<td>SEND_ATTACH_RAW</td>
<td>565</td>
</tr>
<tr>
<td>7.11.3</td>
<td>SEND_ATTACH_VARCHAR2</td>
<td>567</td>
</tr>
<tr>
<td>7.12</td>
<td>UTL_SMTP</td>
<td>569</td>
</tr>
<tr>
<td>7.12.1</td>
<td>CONNECTION</td>
<td>570</td>
</tr>
<tr>
<td>7.12.2</td>
<td>REPLY/REPLIES</td>
<td>570</td>
</tr>
<tr>
<td>7.12.3</td>
<td>CLOSE_DATA</td>
<td>570</td>
</tr>
<tr>
<td>7.12.4</td>
<td>COMMAND</td>
<td>571</td>
</tr>
<tr>
<td>7.12.5</td>
<td>COMMAND_REPLIES</td>
<td>571</td>
</tr>
</tbody>
</table>
Database Compatibility for Oracle® Developer’s Guide

7.12.6   DATA..................................................................................572
7.12.7   EHLO..................................................................................572
7.12.8   HELO..................................................................................573
7.12.9   HELP..................................................................................573
7.12.10  MAIL....................................................................................574
7.12.11  NOOP..................................................................................574
7.12.12  OPEN_CONNECTION.........................................................575
7.12.13  OPEN_DATA........................................................................575
7.12.14  QUIT....................................................................................576
7.12.15  RCPT....................................................................................576
7.12.16  RSET....................................................................................577
7.12.17  VRFY....................................................................................577
7.12.18  WRITE_DATA....................................................................577
7.12.19  Comprehensive Example..................................................578

8   Object Types and Objects............................................................580
8.1   Basic Object Concepts.............................................................580
8.1.1   Attributes..........................................................................581
8.1.2   Methods..............................................................................581
8.1.3   Overloading Methods.........................................................581
8.2   Object Type Components......................................................582
8.2.1   Object Type Specification Syntax.......................................582
8.2.2   Object Type Body Syntax..................................................585
8.3   Creating Object Types.............................................................588
8.4   Creating Object Instances......................................................590
8.5   Referencing an Object.............................................................591
8.6   Dropping an Object Type.........................................................593

9   Open Client Library.................................................................594
9.1   Comparison with Oracle Call Interface....................................594
9.2   Ref Cursor Support..................................................................595
9.3   OCL Reference.......................................................................598
9.3.1   Connect, Authorize and Initialize Functions......................598
9.3.2   Handle and Descriptor Functions.......................................598
9.3.3   Bind, Define and Describe Functions...................................598
9.3.4   Statement Functions............................................................599
9.3.5   Transaction Functions..........................................................599
9.3.6   Date and Datetime Functions..............................................599
9.3.7   Interval Functions.................................................................600
9.3.8   Number Functions...............................................................600
9.3.9   String Functions.................................................................602
9.3.10  Cartridge Services and File I/O Interface Functions............602
9.3.11  Miscellaneous Functions.....................................................602
9.3.12  Supported Data Types..........................................................603

10  Oracle Catalog Views...............................................................604
10.1   ALL_ALL_TABLES.................................................................604
10.2   ALLCONS_COLUMNS............................................................604
10.3   ALL_CONSTRAINTS...............................................................604
10.4   ALL_DB_LINKS.....................................................................605
10.5   ALL_IND_COLUMNS...............................................................605
10.6   ALL_INDEXES.......................................................................606
10.7   ALL_OBJECTS......................................................................607
10.8   ALL_Policies.......................................................................607
10.9   ALL_SEQUENCES.................................................................608
10.10  ALL_SOURCE.......................................................................608
10.11  ALL_SYNONYMS..................................................................609
10.12  ALL_TAB_COLUMNS.............................................................609
10.13  ALL_TABLES........................................................................609
Utilities

11.1 EDB*Plus

11.1.1 Starting EDB*Plus

11.1.2 Command Summary

11.1.2.1 ACCEPT

11.1.2.2 APPEND

11.1.2.3 CHANGE

11.1.2.4 CLEAR

11.1.2.5 COLUMN

11.1.2.6 CONNECT

11.1.2.7 DEFINE

10.58

10.57

10.56

10.55

10.54

10.53

10.52

10.51

10.50

10.49

10.48

10.47

10.46

10.45

10.44

10.43

10.42

10.41

10.40

10.39

10.38

10.37

10.36

10.35

10.34

10.33

10.32

10.31

10.30

10.29

10.28

10.27

10.26

10.25

10.24

10.23

10.22

10.21

10.20

10.19

10.18

10.17

10.16

10.15

10.14

ALL_TRIGGERS ................................................................. 610
ALL_TYPES ........................................................................ 610
ALL_USERS ............................................................... 611
ALL_VIEW_COLUMNS ...................................................... 611
ALL_VIEWS .................................................................. 611
DBA_ALL_TABLES .................................................... 612
DBA_CONS_COLUMNS .................................................. 612
DBA_CONSTRAINTS ..................................................... 612
DBA_DB_LINKS .......................................................... 613
DBA_IND_COLUMNS .................................................... 613
DBA_INDEXES .......................................................... 614
DBA_JOBS ................................................................ 614
DBA_OBJECTS .......................................................... 615
DBA_POLICIES .......................................................... 615
DBA_ROLE_PRIVS ...................................................... 616
DBA_ROLES ................................................................ 617
DBA_SEQUENCES ....................................................... 617
DBA_SOURCE ............................................................. 617
DBA_SYNONYMS ........................................................ 618
DBA_TABLES ............................................................. 618
DBA_TRIGGERS ........................................................... 618
DBA_TYPES ................................................................ 619
DBA_USERS .............................................................. 619
DBA_VIEWS ................................................................ 620
USER_ALL_TABLES .................................................... 620
USER_CONS_COLUMNS ................................................ 620
USER_CONSTRAINTS ..................................................... 621
USER_DB_LINKS .......................................................... 621
USER_IND_COLUMNS .................................................... 622
USER_INDEXES .......................................................... 622
USER_JOBS ................................................................ 623
USER_OBJECTS .......................................................... 623
USER_POLICIES .......................................................... 624
USER_SEQUENCES ....................................................... 625
USER_SOURCE ............................................................. 625
USER_SYNONYMS ........................................................ 625
USER_TAB_COLUMNS .................................................. 626
USER_TABLES ............................................................. 626
USER_TRIGGERS .......................................................... 626
USER_TYPES ............................................................. 627
USER_USERS .............................................................. 627
USER_VIEW_COLUMNS ............................................. 628
USER_VIEWS ............................................................. 628
V$VERSION ................................................................. 628
PRODUCT_COMPONENT_VERSION .................................. 629
11.9.2  edb$session_waits ........................................................................................................714
11.9.3  edb$session_wait_history ..........................................................................................715
12  Table Partitioning ........................................................................................................716
12.1  CREATE TABLE...PARTITION BY ..................................................................................719
12.2  ALTER TABLE...ADD PARTITION .................................................................................724
12.3  ALTER TABLE...ADD SUBPARTITION .........................................................................728
12.4  ALTER TABLE...DROP PARTITION ..............................................................................731
12.5  ALTER TABLE...DROP SUBPARTITION .........................................................................732
12.6  ALTER TABLE...SPLIT PARTITION ..............................................................................733
12.7  ALTER TABLE...SPLIT SUBPARTITION ........................................................................737
12.8  ALTER TABLE...TRUNCATE PARTITION .....................................................................740
12.9  ALTER TABLE...TRUNCATE SUBPARTITION .................................................................742
12.10 ALTER TABLE...EXCHANGE PARTITION ....................................................................744
12.11 ALTER TABLE...MOVE PARTITION ............................................................................746
12.12 ALTER TABLE...RENAME PARTITION ........................................................................748
12.13 Table Partitioning Views ..........................................................................................750
12.13.1 ALL_PART_TABLES ..................................................................................................750
12.13.2 ALL_TAB_PARTITIONS .............................................................................................752
12.13.3 ALL_TAB_SUBPARTITIONS .......................................................................................754
12.13.4 ALL_PART_KEY_COLUMNS .......................................................................................756
12.13.5 ALL_SUBPART_KEY_COLUMNS ..............................................................................756
13  ECPGPlus.....................................................................................................................757
13.1  C-preprocessor Directives ..........................................................................................758
13.2  Supported C Data Types .............................................................................................760
13.3  Type Codes ..................................................................................................................761
13.4  The SQLDA Structure ................................................................................................762
13.5  ECPGPlus Statements ................................................................................................766
13.5.1  ALLOCATE DESCRIPTOR ..........................................................................................766
13.5.2  CALL .........................................................................................................................767
13.5.3  CLOSE .......................................................................................................................768
13.5.4  COMMIT .....................................................................................................................768
13.5.5  CONNECT ..................................................................................................................769
13.5.6  DEALLOCATE DESCRIPTOR ......................................................................................771
13.5.7  DECLARE CURSOR ....................................................................................................772
13.5.8  DECLARE DATABASE ................................................................................................773
13.5.9  DECLARE STATEMENT ...............................................................................................773
13.5.10 DELETE .....................................................................................................................774
13.5.11 DESCRIBE ................................................................................................................775
13.5.12 DESCRIBE DESCRIPTOR ........................................................................................776
13.5.13 DISCONNECT ............................................................................................................778
13.5.14 EXECUTE ..................................................................................................................778
13.5.15 EXECUTE DESCRIPTOR ............................................................................................779
13.5.16 EXECUTE...END EXEC .............................................................................................780
13.5.17 EXECUTE IMMEDIATE ..............................................................................................781
13.5.18 FETCH ......................................................................................................................781
13.5.19 FETCH DESCRIPTOR ...............................................................................................782
13.5.20 GET DESCRIPTOR ....................................................................................................783
13.5.21 INSERT .....................................................................................................................784
13.5.22 OPEN .........................................................................................................................785
13.5.23 OPEN DESCRIPTOR ................................................................................................786
13.5.24 PREPARE ..................................................................................................................787
13.5.25 ROLLBACK ...............................................................................................................788
13.5.26 SAVEPOINT ..............................................................................................................789
13.5.27 SELECT .....................................................................................................................790
13.5.28 SET CONNECTION ..................................................................................................791
13.5.29 SET DESCRIPTOR .....................................................................................................792
13.5.30  UPDATE..................................................................................795
13.5.31  WHENEVER..............................................................................796
14  System Catalog Tables ..............................................................................798
  14.1.1  edb_dir..................................................................................798
  14.1.2  pg_synonym ..............................................................................798
  14.1.3  edb_variable............................................................................799
  14.1.4  product_component_version..........................................................799
15  Acknowledgements ......................................................................................800
1 Introduction

Database compatibility for Oracle® means that an application runs in an Oracle environment as well as in a Postgres Plus Advanced Server environment with minimal or no changes to the application code.

Postgres Plus Advanced Server contains a rich set of features that enables development of database applications for PostgreSQL or Oracle. This guide focuses solely on the features that are compatible with Oracle. To learn about all of the features of Postgres Plus Advanced Server, consult the Postgres Plus documentation set.

Developing a compatible application in Postgres Plus Advanced Server requires special attention to which features are used in the construction of the application. For example, developing a compatible application means choosing:

- compatible data types to define the application’s database tables
- SQL statements that are compatible with Oracle SQL
- compatible system and built-in functions for use in SQL statements and procedural logic
- Stored Procedure Language (SPL) to create database server-side application logic for stored procedures, functions, triggers, and packages
- System catalog views that are compatible with Oracle’s data dictionary

Postgres Plus Advanced Server provides these features.

In addition, for applications written using the Oracle Call Interface (OCI), EnterpriseDB’s Open Client Library (OCL) provides interoperability with these applications.

The remainder of this guide explains each of these areas in more detail.
1.1 What's New

The following database compatibility for Oracle® features have been added to Postgres Plus Advanced Server 9.1 to create Postgres Plus Advanced Server 9.2:

- Advanced Server supports an Oracle-style CASCADE CONSTRAINTS clause for the DROP TABLE command. For more information, see Section 3.3.41.
- Advanced Server now supports the APPEND optimizer hint. For more information, see Section 3.4.6.
- Advanced Server now supports user-defined PL/SQL subtypes. For more information, see Section 4.1.6.
- Advanced Server supports DEFAULT and NOT NULL declarations for members of a RECORD TYPE. For more information, see Section 4.3.4.
- Advanced Server now supports the TABLE() collection operator. For more information, see Section 4.12.1.
- Advanced Server now includes support for the max_error_count configuration parameter. For more information, see Section 4.2.6.
- Advanced Server now supports methods in object types. For more information, see Section 8.
- Advanced Server's EDB*Loader utility now supports multi-byte and single-byte characters as a delimiter. For more information about EDB*Loader, see Section 11.2.
1.2 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 which 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, etc. For example, `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”.


1.3 Oracle Compatible Configuration Parameters

Postgres Plus Advanced Server supports the development and execution of PostgreSQL and Oracle applications. There are a number of system behaviors that can be altered to act in a more PostgreSQL or in a more Oracle compliant manner. These are controlled by configuration parameters that can be found in the postgresql.conf file in the database cluster data directory. Changing the parameters in the postgresql.conf file changes the behavior over all databases in the cluster. More fine-grained adjustment of these parameters can be done by database, by user or group, or by session. These parameters are the following:

- **edb_redwood_date** – Controls whether or not a time component is stored in DATE columns. For Oracle compatible behavior, set `edb_redwood_date` to TRUE.
- **edb_redwood_strings** – Equates NULL to an empty string for purposes of string concatenation operations. For Oracle compatible behavior, set `edb_redwood_strings` to TRUE.
- **edb_stmt_level_tx** – Isolates automatic rollback of an aborted SQL command to statement level rollback only – the entire, current transaction is not automatically rolled back back as is the case for default PostgreSQL behavior. For Oracle compatible behavior, set `edb_stmt_level_tx` to TRUE; however, use only when absolutely necessary. See Section 1.3.3.
- **oracle_home** – Point Postgres Plus Advanced Server to the correct Oracle installation directory. See Section 1.3.4.

1.3.1 edb_redwood_date

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. This is consistent with Oracle’s DATE data type.

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.
1.3.2 edb_redwood_strings

In Oracle, when a string is concatenated with a null variable or null column, the result is the original string; however, in PostgreSQL concatenation of a string with a null variable or null column gives a null result. If the edb_redwood_strings parameter is set to TRUE, the aforementioned concatenation operation results in the original string as done by Oracle. If edb_redwood_strings is set to FALSE, the native PostgreSQL behavior is maintained.

The following example illustrates the difference.

The sample application introduced in the next section 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. This result is consistent with the results produced by Oracle for the same query.

```
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
```
1.3.3 edb_stmt_level_tx

In Oracle, when a runtime error occurs in a SQL command, all the updates on the database caused by that single command are rolled back. This is called statement level transaction isolation. 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 PostgreSQL, 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, emulating the Oracle behavior. 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 SQL, 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_txn 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;
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)
```

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.

### 1.3.4 oracle_home

Before creating a link to an Oracle server, you must direct Advanced Server to the correct Oracle home directory. You can either set the `LD_LIBRARY_PATH` environment variable (or `PATH` on Windows) to the `lib` directory of the Oracle client installation directory or 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 `LD_LIBRARY_PATH` (or `PATH` on Windows) environment variable.

The `LD_LIBRARY_PATH` (or `PATH` on Windows) environment variable must be set properly each time you start Advanced Server. 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 directory that contains `libclntsh.so` (on Linux) or `oci.dll` (on Windows) for `lib_directory`.

After setting the `oracle_home` configuration parameter, you must restart the server for the changes to take effect. Restart the server by executing the following command:

```
/etc/init.d/ppas-9.2 restart
```
1.4 About the Examples Used in this Guide

The examples shown in this guide are illustrated using the PSQL program. The prompt that normally appears when using PSQL is omitted in these examples to provide extra clarity for the point being demonstrated.

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

Also note the following points:

- During installation of Postgres Plus Advanced Server the selection for Oracle compatible configuration and defaults must be chosen in order to reproduce the same results as the examples shown in this guide. A default Oracle compatible configuration can be verified by issuing the following commands in PSQL and obtaining the same results as shown below.

```
SHOW edb_redwood_date;
edb_redwood_date
--------------
on
SHOW datestyle;
  DateStyle
----------
Redwood, DMY
SHOW edb_redwood_strings;
edb_redwood_strings
----------------------
on
```

- The examples use the sample tables, `dept`, `emp`, and `jobhist`, created and loaded when Postgres Plus Advanced Server is installed. The `emp` table is installed with triggers that must be disabled in order to reproduce the same results as shown in this guide. Log on to Postgres Plus Advanced Server as the `enterprisedb` superuser and disable the triggers by issuing the following command.

```
ALTER TABLE emp DISABLE TRIGGER USER;
```

The triggers on the `emp` table can later be re-activated with the following command.

```
ALTER TABLE emp ENABLE TRIGGER USER;
```
2 SQL Tutorial

This section is an introduction to the SQL language for those new to relational database management systems. Basic operations such as creating, populating, querying, and updating tables are discussed along with examples.

More advanced concepts such as view, foreign keys, and transactions are discussed as well.

2.1 Getting Started

Postgres Plus Advanced Server is a relational database management system (RDBMS). That means it is a system for managing data stored in relations. A relation is essentially a mathematical term for a table. The notion of storing data in tables is so commonplace today that it might seem inherently obvious, but there are a number of other ways of organizing databases. Files and directories on Unix-like operating systems form an example of a hierarchical database. A more modern development is the object-oriented database.

Each table is a named collection of rows. Each row of a given table has the same set of named columns, and each column is of a specific data type. Whereas columns have a fixed order in each row, it is important to remember that SQL does not guarantee the order of the rows within the table in any way (although they can be explicitly sorted for display).

Tables are grouped into databases, and a collection of databases managed by a single Postgres Plus Advanced Server instance constitutes a database cluster.
2.1.1 Sample Database

Throughout this documentation we will be working with a sample database to help explain some basic to advanced level database concepts.

2.1.1.1 Sample Database Installation

When Postgres Plus Advanced Server is installed a sample database named, edb, is automatically created. This sample database contains the tables and programs used throughout this document.

The tables and programs in the sample database can be re-created at any time by executing the script, edb-sample.sql, located in the samples subdirectory of the Postgres Plus Advanced Server home directory.

This script does the following:

- 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;
```

Altering the search path can be done using commands in PSQL.

2.1.1.2 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 the database keeps track of the location of the 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 an entity relationship diagram of the sample database tables.

![Figure 1 Sample Database Tables](image)

The following is the `edb-sample.sql` script.

```sql
-- Script that creates the 'sample' tables, views, procedures, 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  NUMBER(2) NOT NULL CONSTRAINT dept_pk PRIMARY KEY,
  dname   VARCHAR2(14) CONSTRAINT dept_dname_uq UNIQUE,
  loc     VARCHAR2(13)
);
```

Copyright © 2007 - 2015 EnterpriseDB Corporation. All rights reserved.
-- Create the 'emp' table
CREATE TABLE emp (  
  empno       NUMBER(4) NOT NULL CONSTRAINT emp_pk PRIMARY KEY,  
  ename       VARCHAR2(10),  
  job         VARCHAR2(9),  
  mgr         NUMBER(4),  
  hiredate    DATE,  
  sal         NUMBER(7,2) CONSTRAINT emp_sal_ck CHECK (sal > 0),  
  comm        NUMBER(7,2),  
  deptno      NUMBER(2) CONSTRAINT emp_ref_dept_fk  
        REFERENCES dept(deptno)
);

-- Create the 'jobhist' table
CREATE TABLE jobhist (  
  empno       NUMBER(4) NOT NULL,  
  startdate   DATE NOT NULL,  
  enddate     DATE,  
  job         VARCHAR2(9),  
  sal         NUMBER(7,2),  
  comm        NUMBER(7,2),  
  deptno      NUMBER(2),  
  chgdesc     VARCHAR2(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,30);
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 (7788,'19-APR-87',12-APR-88','CLERK',1000,NULL,20,'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-89',NULL,'ANALYST',3000,NULL,20,'Promoted to Analyst');
INSERT INTO jobhist VALUES (7789,'17-NOV-81',NULL,'PRESIDENT',5000,NULL,10,'New Hire');
INSERT INTO jobhist VALUES (7839,'08-SEP-81',NULL,'SALESMAN',1500,0,30,'New Hire');
INSERT INTO jobhist VALUES (7844,'17-DEC-80',NULL,'CLERK',800,NULL,20,'New Hire');
```

```
-- Load the 'jobhist' table
--
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,30,'New Hire');
INSERT INTO jobhist VALUES (7566,'12-APR-88',13-APR-88','CLERK',1000,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 (7698,'01-APR-87',3000,NULL,20,'New Hire');
INSERT INTO jobhist VALUES (7782,'09-JUN-81',2450,NULL,10,'New Hire');
INSERT INTO jobhist VALUES (7788,'23-JAN-82',1300,NULL,10,'New Hire');
```

```
-- Populate statistics table and view (pg_statistic/pg_stats)
--
ANALYZE dept;
ANALYZE emp;
ANALYZE jobhist;
```
-- Procedure that lists all employees' numbers and names
-- from the 'emp' table using a cursor.
PROCEDURE list_emp IS
  v_empno NUMBER(4);
  v_ename VARCHAR2(10);
  CURSOR emp_cur IS
    SELECT empno, ename FROM emp ORDER BY empno;
BEGIN
  OPEN emp_cur;
  DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
  DBMS_OUTPUT.PUT_LINE('-----    -------');
  LOOP
    FETCH emp_cur INTO v_empno, v_ename;
    EXIT WHEN emp_cur%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE(v_empno || '     ' || v_ename);
  END LOOP;
  CLOSE emp_cur;
END;
/
-- Procedure that selects an employee row given the employee
-- number and displays certain columns.
PROCEDURE select_emp (p_empno IN NUMBER) IS
  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 VARCHAR2(10);
BEGIN
  SELECT ename, hiredate, sal, NVL(comm, 0), dname
  INTO v_ename, v_hiredate, v_sal, v_comm, v_dname
  FROM emp e, dept d
  WHERE empno = p_empno
    AND e.deptno = d.deptno;
  v_disp_date := TO_CHAR(v_hiredate, 'MM/DD/YYYY');
  DBMS_OUTPUT.PUT_LINE('Number : ' || p_empno);
  DBMS_OUTPUT.PUT_LINE('Name : ' || v_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);
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    DBMS_OUTPUT.PUT_LINE('Employee ' || p_empno || ' not found');
  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);
END;
/
-- Procedure that queries the 'emp' table based on
-- department number and employee number or name. Returns
-- employee number and name as IN OUT parameters and job,
-- hire date, and salary as OUT parameters.
--
CREATE OR REPLACE PROCEDURE emp_query (p_deptno IN NUMBER,
p_empno IN OUT NUMBER,
p_ename IN OUT VARCHAR2,
p_job OUT VARCHAR2,
p_hiredate OUT DATE,
p_sal OUT NUMBER)
IS
BEGIN
SELECT empno, ename, job, hiredate, sal
INTO p_empno, p_ename, p_job, p_hiredate, p_sal
FROM emp
WHERE deptno = p_deptno
AND (empno = p_empno
OR ename = UPPER(p_ename));
END;
/
--
-- Procedure to call 'emp_query_caller' with IN and IN OUT
-- parameters. Displays the results received from IN OUT and
-- OUT parameters.
--
CREATE OR REPLACE PROCEDURE emp_query_caller
IS
v_deptno NUMBER(2);
v_empno NUMBER(4);
v_ename VARCHAR2(10);
v_job VARCHAR2(9);
v_hiredate DATE;
v_sal NUMBER;
BEGIN
v_deptno := 30;
v_empno  := 0;
v_ename  := 'Martin';
emp_query(v_deptno, v_empno, v_ename, v_job, v_hiredate, v_sal);
DBMS_OUTPUT.PUT_LINE('Department : ' || v_deptno);
DBMS_OUTPUT.PUT_LINE('Employee No: ' || v_empno);
DBMS_OUTPUT.PUT_LINE('Name       : ' || v_ename);
DBMS_OUTPUT.PUT_LINE('Job        : ' || v_job);
DBMS_OUTPUT.PUT_LINE('Hire Date  : ' || v_hiredate);
DBMS_OUTPUT.PUT_LINE('Salary     : ' || v_sal);
EXCEPTION
WHEN TOO_MANY_ROWS THEN
DBMS_OUTPUT.PUT_LINE('More than one employee was selected');
WHEN NO_DATA_FOUND THEN
DBMS_OUTPUT.PUT_LINE('No employees were selected');
END;
/
--
-- Function to compute yearly compensation based on semimonthly
-- salary.
--
CREATE OR REPLACE FUNCTION emp_comp (p_sal NUMBER,
p_comm NUMBER)
RETURN NUMBER
IS
BEGIN
RETURN (p_sal + NVL(p_comm, 0)) * 24;
END;
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 RETURN NUMBER IS
  v_cnt           INTEGER := 1;
  v_new_empno     NUMBER;
BEGIN
  WHILE v_cnt > 0 LOOP
    SELECT next_empno.nextval INTO v_new_empno FROM dual;
    SELECT COUNT(*) INTO v_cnt FROM emp WHERE empno = v_new_empno;
  END LOOP;
  RETURN v_new_empno;
END;
```

EDB-SPL function that adds a new clerk to table 'emp'. This function uses package 'emp_admin'.

```
CREATE OR REPLACE FUNCTION hire_clerk (p_ename VARCHAR2, p_deptno NUMBER) RETURN NUMBER IS
  v_empno         NUMBER(4);
  v_ename         VARCHAR2(10);
  v_job           VARCHAR2(9);
  v_mgr NUMBER(4);
  v_hiredate      DATE;
  v_sal NUMBER(7,2);
  v_comm NUMBER(7,2);
  v_deptno NUMBER(2);
BEGIN
  v_empno := new_empno;
  INSERT INTO emp VALUES (v_empno, p_ename, 'CLERK', 7782, TRUNC(SYSDATE), 950.00, NULL, p_deptno);
  SELECT empno, ename, job, mgr, hiredate, sal, comm, deptno INTO v_empno, v_ename, v_job, v_mgr, v_hiredate, v_sal, v_comm, v_deptno FROM emp WHERE empno = v_empno;
  DBMS_OUTPUT.PUT_LINE('Department : ' || v_deptno);
  DBMS_OUTPUT.PUT_LINE('Employee No: ' || v_empno);
  DBMS_OUTPUT.PUT_LINE('Name       : ' || v_ename);
  DBMS_OUTPUT.PUT_LINE('Job        : ' || v_job);
  DBMS_OUTPUT.PUT_LINE('Manager    : ' || v_mgr);
  DBMS_OUTPUT.PUT_LINE('Hire Date  : ' || v_hiredate);
  DBMS_OUTPUT.PUT_LINE('Salary     : ' || v_sal);
  DBMS_OUTPUT.PUT_LINE('Commission : ' || v_comm);
  RETURN v_empno;
EXCEPTION
  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);
    RETURN -1;
END;
```

PostgreSQL PL/pgSQL 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 $$  
DECLARE  
    v_empno         NUMERIC(4);  
    v_ename         VARCHAR(10);  
    v_job           VARCHAR(9);  
    v_mgr           NUMERIC(4);  
    v_hiredate      DATE;  
    v_sal           NUMERIC(7,2);  
    v_comm          NUMERIC(7,2);  
    v_deptno        NUMERIC(2);  
BEGIN  
    v_empno := new_empno();  
    INSERT INTO emp VALUES (v_empno, p_ename, 'SALESMAN', 7698,  
                            CURRENT_DATE, p_sal, p_comm, 30);  
    SELECT INTO  
        v_empno, v_ename, v_job, v_mgr, v_hiredate, v_sal, v_comm, v_deptno  
    empno, ename, job, mgr, hiredate, sal, comm, deptno  
    FROM emp WHERE empno = v_empno;  
    RAISE INFO 'Department : %', v_deptno;  
    RAISE INFO 'Employee No: %', v_empno;  
    RAISE INFO 'Name       : %', v_ename;  
    RAISE INFO 'Job        : %', v_job;  
    RAISE INFO 'Manager    : %', v_mgr;  
    RAISE INFO 'Hire Date  : %', v_hiredate;  
    RAISE INFO 'Salary     : %', v_sal;  
    RAISE INFO 'Commission : %', v_comm;  
    RETURN v_empno;  
END;  
$$ 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 TRIGGER user_audit_trig
  AFTER INSERT OR UPDATE OR DELETE ON emp
DECLARE
    v_action        VARCHAR2(24);
BEGIN
    IF INSERTING THEN
      v_action := ' added employee(s) on ';
    ELSIF UPDATING THEN
      v_action := ' updated employee(s) on ';
    ELSIF DELETING THEN
      v_action := ' deleted employee(s) on ';
    END IF;
    DBMS_OUTPUT.PUT_LINE('User ' || USER || v_action ||
      TO_CHAR(SYSDATE,'YYYY-MM-DD'));
END;
/
--
-- 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 TRIGGER emp_sal_trig
  BEFORE DELETE OR INSERT OR UPDATE ON emp
FOR EACH ROW
DECLARE
    sal_diff       NUMBER;
BEGIN
    IF INSERTING THEN
      DBMS_OUTPUT.PUT_LINE('Inserting employee ' || :NEW.empno);
      DBMS_OUTPUT.PUT_LINE('..New salary: ' || :NEW.sal);
    END IF;
    IF UPDATING THEN
      sal_diff := :NEW.sal - :OLD.sal;
      DBMS_OUTPUT.PUT_LINE('..Old salary: ' || :OLD.sal);
      DBMS_OUTPUT.PUT_LINE('..New salary: ' || :NEW.sal);
      DBMS_OUTPUT.PUT_LINE('..Raise     : ' || sal_diff);
    END IF;
    IF DELETING THEN
      DBMS_OUTPUT.PUT_LINE('Deleting employee ' || :OLD.empno);
      DBMS_OUTPUT.PUT_LINE('..Old salary: ' || :OLD.sal);
    END IF;
END;
/
--
-- Package specification for the 'emp_admin' package.
--
CREATE OR REPLACE PACKAGE emp_admin
IS
  FUNCTION get_dept_name (p_deptno NUMBER) RETURN VARCHAR2;
  FUNCTION update_emp_sal (p_empno NUMBER, p_raise NUMBER) RETURN VARCHAR2;
END emp_admin;
/
--
PROCEDURE hire_emp {
    p_empno         NUMBER,
    p_ename         VARCHAR2,
    p_job           VARCHAR2,
    p_sal           NUMBER,
    p_hiredate      DATE,
    p_comm          NUMBER,
    p_mgr           NUMBER,
    p_deptno        NUMBER
};
PROCEDURE fire_emp {
    p_empno         NUMBER
};
END emp_admin;
/
-- Package body for the 'emp_admin' package.
--
CREATE OR REPLACE PACKAGE BODY emp_admin
IS
    FUNCTION get_dept_name (p_deptno IN NUMBER) RETURN VARCHAR2 IS
        v_dname VARCHAR2(14);
        BEGIN
            SELECT dname INTO v_dname FROM dept WHERE deptno = p_deptno;
            RETURN v_dname;
        EXCEPTION
            WHEN NO_DATA_FOUND THEN
                DBMS_OUTPUT.PUT_LINE('Invalid department number ' || p_deptno);
                RETURN '';
    END;

    FUNCTION update_emp_sal (p_empno IN NUMBER, p_raise IN NUMBER) RETURN NUMBER IS
        v_sal NUMBER := 0;
        BEGIN
            SELECT sal INTO v_sal FROM emp WHERE empno = p_empno;
            v_sal := v_sal + p_raise;
            UPDATE emp SET sal = v_sal WHERE empno = p_empno;
            RETURN v_sal;
        EXCEPTION
            WHEN NO_DATA_FOUND THEN
                DBMS_OUTPUT.PUT_LINE('Employee ' || p_empno || ' not found');
                RETURN -1;
            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);
RETURN -1;
END;

-- Procedure that inserts a new employee record into the 'emp' table.
PROCEDURE hire_emp (p_empno NUMBER,
p_ename VARCHAR2,
p_job VARCHAR2,
p_sal NUMBER,
p_hiredate DATE,
p_comm NUMBER,
p_mgr NUMBER,
p_deptno NUMBER)
AS
BEGIN
    INSERT INTO emp(empno, ename, job, sal, hiredate, comm, mgr, deptno)
    VALUES(p_empno, p_ename, p_job, p_sal,
            p_hiredate, p_comm, p_mgr, p_deptno);
END;

-- Procedure that deletes an employee record from the 'emp' table based
-- on the employee number.
PROCEDURE fire_emp (p_empno NUMBER)
AS
BEGIN
    DELETE FROM emp WHERE empno = p_empno;
END;
/COMMIT;
2.1.2 Creating a New Table

A new table is created by specifying the table name, along with all column names and their types. The following is a simplified version of the `emp` sample table with just the minimal information needed to define a table.

```sql
CREATE TABLE emp (  
    empno           NUMBER(4),  
    ename           VARCHAR2(10),  
    job             VARCHAR2(9),  
    mgr             NUMBER(4),  
    hiredate        DATE,  
    sal             NUMBER(7,2),  
    comm            NUMBER(7,2),  
    deptno          NUMBER(2)  
) ;
```

You can enter this into PSQL with line breaks. PSQL will recognize that the command is not terminated until the semicolon.

White space (i.e., spaces, tabs, and newlines) may be used freely in SQL commands. That means you can type the command aligned differently than the above, or even all on one line. Two dashes ("--") introduce comments. Whatever follows them is ignored up to the end of the line. SQL is case insensitive about key words and identifiers, except when identifiers are double-quoted to preserve the case (not done above).

`VARCHAR2(10)` specifies a data type that can store arbitrary character strings up to 10 characters in length. `NUMBER(7,2)` is a fixed point number with precision 7 and scale 2. `NUMBER(4)` is an integer number with precision 4 and scale 0.

Postgres Plus Advanced Server supports the usual SQL data types `INTEGER`, `SMALLINT`, `NUMBER`, `REAL`, `DOUBLE PRECISION`, `CHAR`, `VARCHAR2`, `DATE`, and `TIMESTAMP` as well as various synonyms for these types.

If you don’t need a table any longer or want to recreate it differently you can remove it using the following command:

```sql
DROP TABLE tablename;
```
2.1.3 Populating a Table With Rows

The `INSERT` statement is used to populate a table with rows:

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

Note that all data types use rather obvious input formats. Constants that are not simple numeric values usually must be surrounded by single quotes ('), as in the example. The `DATE` type is actually quite flexible in what it accepts, but for this tutorial we will stick to the unambiguous format shown here.

The syntax used so far requires you to remember the order of the columns. An alternative syntax allows you to list the columns explicitly:

```
INSERT INTO emp(empno,ename,job,mgr,hiredate,sal,comm,deptno)
VALUES (7499,'ALLEN','SALESMAN',7698,'20-FEB-81',1600,300,30);
```

You can list the columns in a different order if you wish or even omit some columns, e.g., if the commission is unknown:

```
INSERT INTO emp(empno,ename,job,mgr,hiredate,sal,deptno)
VALUES (7369,'SMITH','CLERK',7902,'17-DEC-80',800,20);
```

Many developers consider explicitly listing the columns better style than relying on the order implicitly.
2.1.4 Querying a Table

To retrieve data from a table, the table is *queried*. An SQL SELECT statement is used to do this. The statement is divided into a select list (the part that lists the columns to be returned), a table list (the part that lists the tables from which to retrieve the data), and an optional qualification (the part that specifies any restrictions). The following query lists all columns of all employees in the table in no particular order.

```
SELECT * FROM emp;
```

Here, "*" in the select list means all columns. The following is the output from this query.

<table>
<thead>
<tr>
<th>empno</th>
<th>ename</th>
<th>job</th>
<th>mgr</th>
<th>hiredate</th>
<th>sal</th>
<th>comm</th>
<th>deptno</th>
</tr>
</thead>
<tbody>
<tr>
<td>7369</td>
<td>SMITH</td>
<td>CLERK</td>
<td>7902</td>
<td>17-DEC-80 00:00:00</td>
<td>800.00</td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>7499</td>
<td>ALLEN</td>
<td>SALESMAN</td>
<td>7698</td>
<td>20-FEB-81 00:00:00</td>
<td>1600.00</td>
<td>300.00</td>
<td>30</td>
</tr>
<tr>
<td>7521</td>
<td>WARD</td>
<td>SALESMAN</td>
<td>7698</td>
<td>22-FEB-81 00:00:00</td>
<td>1250.00</td>
<td>500.00</td>
<td>30</td>
</tr>
<tr>
<td>7566</td>
<td>JONES</td>
<td>MANAGER</td>
<td>7839</td>
<td>02-APR-81 00:00:00</td>
<td>2975.00</td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>7654</td>
<td>MARTIN</td>
<td>SALESMAN</td>
<td>7698</td>
<td>28-SEP-81 00:00:00</td>
<td>1250.00</td>
<td>1400.00</td>
<td>30</td>
</tr>
<tr>
<td>7698</td>
<td>BLAKE</td>
<td>MANAGER</td>
<td>7839</td>
<td>01-MAY-81 00:00:00</td>
<td>2850.00</td>
<td></td>
<td>30</td>
</tr>
<tr>
<td>7782</td>
<td>CLARK</td>
<td>MANAGER</td>
<td>7839</td>
<td>09-JUN-81 00:00:00</td>
<td>2450.00</td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>7788</td>
<td>SCOTT</td>
<td>ANALYST</td>
<td>7566</td>
<td>19-APR-87 00:00:00</td>
<td>3000.00</td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>7839</td>
<td>KING</td>
<td>PRESIDENT</td>
<td></td>
<td>17-NOV-81 00:00:00</td>
<td>5000.00</td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>7844</td>
<td>TURNER</td>
<td>SALESMAN</td>
<td>7698</td>
<td>08-SEP-81 00:00:00</td>
<td>1500.00</td>
<td>0.00</td>
<td>30</td>
</tr>
<tr>
<td>7876</td>
<td>ADAMS</td>
<td>CLERK</td>
<td>7782</td>
<td>23-MAY-87 00:00:00</td>
<td>1100.00</td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>7900</td>
<td>JAMES</td>
<td>CLERK</td>
<td>7698</td>
<td>03-DEC-81 00:00:00</td>
<td>950.00</td>
<td></td>
<td>30</td>
</tr>
<tr>
<td>7902</td>
<td>FORO</td>
<td>ANALYST</td>
<td>7566</td>
<td>03-DEC-81 00:00:00</td>
<td>3000.00</td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>7934</td>
<td>MILLER</td>
<td>CLERK</td>
<td>7782</td>
<td>23-JAN-82 00:00:00</td>
<td>1300.00</td>
<td></td>
<td>10</td>
</tr>
</tbody>
</table>

(14 rows)

You may specify any arbitrary expression in the select list. For example, you can do:

```
SELECT ename, sal, sal * 24 AS yearly_salary, deptno FROM emp;
```

```
<table>
<thead>
<tr>
<th>ename</th>
<th>sal</th>
<th>yearly_salary</th>
<th>deptno</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMITH</td>
<td>800.00</td>
<td>19200.00</td>
<td>20</td>
</tr>
<tr>
<td>ALLEN</td>
<td>1600.00</td>
<td>38400.00</td>
<td>30</td>
</tr>
<tr>
<td>WARD</td>
<td>1250.00</td>
<td>30000.00</td>
<td>30</td>
</tr>
<tr>
<td>JONES</td>
<td>2975.00</td>
<td>71400.00</td>
<td>20</td>
</tr>
<tr>
<td>MARTIN</td>
<td>1250.00</td>
<td>30000.00</td>
<td>30</td>
</tr>
<tr>
<td>BLAKE</td>
<td>2850.00</td>
<td>68400.00</td>
<td>30</td>
</tr>
<tr>
<td>CLARK</td>
<td>2450.00</td>
<td>58800.00</td>
<td>10</td>
</tr>
<tr>
<td>SCOTT</td>
<td>3000.00</td>
<td>72000.00</td>
<td>20</td>
</tr>
<tr>
<td>KING</td>
<td>5000.00</td>
<td>120000.00</td>
<td>10</td>
</tr>
<tr>
<td>TURNER</td>
<td>1500.00</td>
<td>36000.00</td>
<td>30</td>
</tr>
<tr>
<td>ADAMS</td>
<td>1100.00</td>
<td>26400.00</td>
<td>20</td>
</tr>
<tr>
<td>JAMES</td>
<td>950.00</td>
<td>22800.00</td>
<td>30</td>
</tr>
<tr>
<td>FORO</td>
<td>3000.00</td>
<td>72000.00</td>
<td>20</td>
</tr>
<tr>
<td>MILLER</td>
<td>1300.00</td>
<td>31200.00</td>
<td>10</td>
</tr>
</tbody>
</table>
```

(14 rows)

Notice how the `AS` clause is used to re-label the output column. (The `AS` clause is optional.)
A query can be qualified by adding a WHERE clause that specifies which rows are wanted. The WHERE clause contains a Boolean (truth value) expression, and only rows for which the Boolean expression is true are returned. The usual Boolean operators (AND, OR, and NOT) are allowed in the qualification. For example, the following retrieves the employees in department 20 with salaries over $1000.00:

```
SELECT ename, sal, deptno FROM emp WHERE deptno = 20 AND sal > 1000;
```

<table>
<thead>
<tr>
<th>ename</th>
<th>sal</th>
<th>deptno</th>
</tr>
</thead>
<tbody>
<tr>
<td>JONES</td>
<td>2975.00</td>
<td>20</td>
</tr>
<tr>
<td>SCOTT</td>
<td>3000.00</td>
<td>20</td>
</tr>
<tr>
<td>ADAMS</td>
<td>1100.00</td>
<td>20</td>
</tr>
<tr>
<td>FORD</td>
<td>3000.00</td>
<td>20</td>
</tr>
</tbody>
</table>

(4 rows)

You can request that the results of a query be returned in sorted order:

```
SELECT ename, sal, deptno FROM emp ORDER BY ename;
```

<table>
<thead>
<tr>
<th>ename</th>
<th>sal</th>
<th>deptno</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADAMS</td>
<td>1100.00</td>
<td>20</td>
</tr>
<tr>
<td>ALLEN</td>
<td>1600.00</td>
<td>30</td>
</tr>
<tr>
<td>BLAKE</td>
<td>2850.00</td>
<td>30</td>
</tr>
<tr>
<td>CLARK</td>
<td>2450.00</td>
<td>10</td>
</tr>
<tr>
<td>FORD</td>
<td>3000.00</td>
<td>20</td>
</tr>
<tr>
<td>JAMES</td>
<td>950.00</td>
<td>30</td>
</tr>
<tr>
<td>JONES</td>
<td>2975.00</td>
<td>20</td>
</tr>
<tr>
<td>KING</td>
<td>5000.00</td>
<td>10</td>
</tr>
<tr>
<td>MARTIN</td>
<td>1250.00</td>
<td>30</td>
</tr>
<tr>
<td>MILLER</td>
<td>1300.00</td>
<td>10</td>
</tr>
<tr>
<td>SCOTT</td>
<td>3000.00</td>
<td>20</td>
</tr>
<tr>
<td>SMITH</td>
<td>800.00</td>
<td>20</td>
</tr>
<tr>
<td>TURNER</td>
<td>1500.00</td>
<td>30</td>
</tr>
<tr>
<td>WARD</td>
<td>1250.00</td>
<td>30</td>
</tr>
</tbody>
</table>

(14 rows)

You can request that duplicate rows be removed from the result of a query:

```
SELECT DISTINCT job FROM emp;
```

```
job
----
ANALYST
CLERK
MANAGER
PRESIDENT
SALESMAN
```

(5 rows)

The following section shows how to obtain rows from more than one table in a single query.
2.1.5 Joins Between Tables

Thus far, our queries have only accessed one table at a time. Queries can access multiple tables at once, or access the same table in such a way that multiple rows of the table are being processed at the same time. A query that accesses multiple rows of the same or different tables at one time is called a join query. For example, say you wish to list all the employee records together with the name and location of the associated department. To do that, we need to compare the deptno column of each row of the emp table with the deptno column of all rows in the dept table, and select the pairs of rows where these values match. This would be accomplished by the following query:

```
SELECT emp.ename, emp.sal, dept.deptno, dept.dname, dept.loc FROM emp, dept
WHERE emp.deptno = dept.deptno;
```

<table>
<thead>
<tr>
<th>ename</th>
<th>sal</th>
<th>deptno</th>
<th>dname</th>
<th>loc</th>
</tr>
</thead>
<tbody>
<tr>
<td>MILLER</td>
<td>1300.00</td>
<td>10</td>
<td>ACCOUNTING</td>
<td>NEW YORK</td>
</tr>
<tr>
<td>CLARK</td>
<td>2450.00</td>
<td>10</td>
<td>ACCOUNTING</td>
<td>NEW YORK</td>
</tr>
<tr>
<td>KING</td>
<td>5000.00</td>
<td>10</td>
<td>ACCOUNTING</td>
<td>NEW YORK</td>
</tr>
<tr>
<td>SCOTT</td>
<td>3000.00</td>
<td>20</td>
<td>RESEARCH</td>
<td>DALLAS</td>
</tr>
<tr>
<td>JONES</td>
<td>2975.00</td>
<td>20</td>
<td>RESEARCH</td>
<td>DALLAS</td>
</tr>
<tr>
<td>SMITH</td>
<td>800.00</td>
<td>20</td>
<td>RESEARCH</td>
<td>DALLAS</td>
</tr>
<tr>
<td>ADAMS</td>
<td>1100.00</td>
<td>20</td>
<td>RESEARCH</td>
<td>DALLAS</td>
</tr>
<tr>
<td>FORD</td>
<td>3000.00</td>
<td>20</td>
<td>RESEARCH</td>
<td>DALLAS</td>
</tr>
<tr>
<td>WARD</td>
<td>1250.00</td>
<td>30</td>
<td>SALES</td>
<td>CHICAGO</td>
</tr>
<tr>
<td>TURNER</td>
<td>1500.00</td>
<td>30</td>
<td>SALES</td>
<td>CHICAGO</td>
</tr>
<tr>
<td>ALLEN</td>
<td>1600.00</td>
<td>30</td>
<td>SALES</td>
<td>CHICAGO</td>
</tr>
<tr>
<td>BLAKE</td>
<td>2850.00</td>
<td>30</td>
<td>SALES</td>
<td>CHICAGO</td>
</tr>
<tr>
<td>MARTIN</td>
<td>1250.00</td>
<td>30</td>
<td>SALES</td>
<td>CHICAGO</td>
</tr>
<tr>
<td>JAMES</td>
<td>950.00</td>
<td>30</td>
<td>SALES</td>
<td>CHICAGO</td>
</tr>
</tbody>
</table>

(14 rows)

Observe two things about the result set:

- There is no result row for department 40. This is because there is no matching entry in the emp table for department 40, so the join ignores the unmatched rows in the dept table. Shortly we will see how this can be fixed.
- It is more desirable to list the output columns qualified by table name rather than using * or leaving out the qualification as follows:

```
SELECT ename, sal, dept.deptno, dname, loc FROM emp, dept
WHERE emp.deptno = dept.deptno;
```

Since all the columns had different names (except for deptno which therefore must be qualified), the parser automatically found out which table they belong to, but it is good style to fully qualify column names in join queries:

Join queries of the kind seen thus far can also be written in this alternative form:
SELECT emp.ename, emp.sal, dept.deptno, dept.dname, dept.loc FROM emp INNER JOIN dept ON emp.deptno = dept.deptno;

This syntax is not as commonly used as the one above, but we show it here to help you understand the following topics.

You will notice that in all the above results for joins no employees were returned that belonged to department 40 and as a consequence, the record for department 40 never appears. Now we will figure out how we can get the department 40 record in the results despite the fact that there are no matching employees. What we want the query to do is to scan the dept table and for each row to find the matching emp row. If no matching row is found we want some “empty” values to be substituted for the emp table’s columns. This kind of query is called an outer join. (The joins we have seen so far are inner joins.) The command looks like this:

```
SELECT emp.ename, emp.sal, dept.deptno, dept.dname, dept.loc FROM dept LEFT OUTER JOIN emp ON emp.deptno = dept.deptno;
```

<table>
<thead>
<tr>
<th>ename</th>
<th>sal</th>
<th>deptno</th>
<th>dname</th>
<th>loc</th>
</tr>
</thead>
<tbody>
<tr>
<td>MILLER</td>
<td>1300.00</td>
<td>10</td>
<td>ACCOUNTING</td>
<td>NEW YORK</td>
</tr>
<tr>
<td>CLARK</td>
<td>2450.00</td>
<td>10</td>
<td>ACCOUNTING</td>
<td>NEW YORK</td>
</tr>
<tr>
<td>KING</td>
<td>5000.00</td>
<td>10</td>
<td>ACCOUNTING</td>
<td>NEW YORK</td>
</tr>
<tr>
<td>SCOTT</td>
<td>3000.00</td>
<td>20</td>
<td>RESEARCH</td>
<td>DALLAS</td>
</tr>
<tr>
<td>JONES</td>
<td>2975.00</td>
<td>20</td>
<td>RESEARCH</td>
<td>DALLAS</td>
</tr>
<tr>
<td>SMITH</td>
<td>800.00</td>
<td>20</td>
<td>RESEARCH</td>
<td>DALLAS</td>
</tr>
<tr>
<td>ADAMS</td>
<td>1100.00</td>
<td>20</td>
<td>RESEARCH</td>
<td>DALLAS</td>
</tr>
<tr>
<td>FORD</td>
<td>3000.00</td>
<td>20</td>
<td>RESEARCH</td>
<td>DALLAS</td>
</tr>
<tr>
<td>WARD</td>
<td>1250.00</td>
<td>30</td>
<td>SALES</td>
<td>CHICAGO</td>
</tr>
<tr>
<td>TURNER</td>
<td>1500.00</td>
<td>30</td>
<td>SALES</td>
<td>CHICAGO</td>
</tr>
<tr>
<td>ALLEN</td>
<td>1600.00</td>
<td>30</td>
<td>SALES</td>
<td>CHICAGO</td>
</tr>
<tr>
<td>BLAKE</td>
<td>2850.00</td>
<td>30</td>
<td>SALES</td>
<td>CHICAGO</td>
</tr>
<tr>
<td>MARTIN</td>
<td>1250.00</td>
<td>30</td>
<td>SALES</td>
<td>CHICAGO</td>
</tr>
<tr>
<td>JAMES</td>
<td>950.00</td>
<td>30</td>
<td>SALES</td>
<td>CHICAGO</td>
</tr>
<tr>
<td></td>
<td></td>
<td>40</td>
<td>OPERATIONS</td>
<td>BOSTON</td>
</tr>
</tbody>
</table>

This query is called a left outer join because the table mentioned on the left of the join operator will have each of its rows in the output at least once, whereas the table on the right will only have those rows output that match some row of the left table. When a left-table row is selected for which there is no right-table match, empty (NULL) values are substituted for the right-table columns.

An alternative syntax for an outer join is to use the outer join operator, “(+),” in the join condition within the WHERE clause. The outer join operator is placed after the column name of the table for which null values should be substituted for unmatched rows. So for all the rows in the dept table that have no matching rows in the emp table, Postgres Plus Advanced Server returns null for any select list expressions containing columns of emp. Hence the above example could be rewritten as:

```
SELECT emp.ename, emp.sal, dept.deptno, dept.dname, dept.loc FROM dept, emp
WHERE emp.deptno(+) = dept.deptno;
```
We can also join a table against itself. This is called a self join. As an example, suppose we wish to find the name of each employee along with the name of that employee’s manager. So we need to compare the `mgr` column of each `emp` row to the `empno` column of all other `emp` rows.

```
SELECT e1.ename || ' works for ' || e2.ename AS "Employees and their Managers" FROM emp e1, emp e2 WHERE e1.mgr = e2.empno;
```

**Employees and their Managers**

<table>
<thead>
<tr>
<th>FORD</th>
<th>works for JONES</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCOTT</td>
<td>works for JONES</td>
</tr>
<tr>
<td>WARD</td>
<td>works for BLAKE</td>
</tr>
<tr>
<td>TURNER</td>
<td>works for BLAKE</td>
</tr>
<tr>
<td>MARTIN</td>
<td>works for BLAKE</td>
</tr>
<tr>
<td>ALLEN</td>
<td>works for BLAKE</td>
</tr>
<tr>
<td>MILLER</td>
<td>works for CLARK</td>
</tr>
<tr>
<td>ADAMS</td>
<td>works for SCOTT</td>
</tr>
<tr>
<td>CLARK</td>
<td>works for KING</td>
</tr>
<tr>
<td>BLAKE</td>
<td>works for KING</td>
</tr>
<tr>
<td>JONES</td>
<td>works for KING</td>
</tr>
<tr>
<td>SMITH</td>
<td>works for FORD</td>
</tr>
</tbody>
</table>

(13 rows)

Here, the `emp` table has been re-labeled as `e1` to represent the employee row in the select list and in the join condition, and also as `e2` to represent the matching employee row acting as manager in the select list and in the join condition. These kinds of aliases can be used in other queries to save some typing, for example:

```
SELECT e.ename, e.mgr, d.deptno, d.dname, d.loc FROM emp e, dept d WHERE e.deptno = d.deptno;
```

<table>
<thead>
<tr>
<th>ename</th>
<th>mgr</th>
<th>deptno</th>
<th>dname</th>
<th>loc</th>
</tr>
</thead>
<tbody>
<tr>
<td>MILLER</td>
<td>7782</td>
<td>10</td>
<td>ACCOUNTING</td>
<td>NEW YORK</td>
</tr>
<tr>
<td>CLARK</td>
<td>7839</td>
<td>10</td>
<td>ACCOUNTING</td>
<td>NEW YORK</td>
</tr>
<tr>
<td>KING</td>
<td>7839</td>
<td>10</td>
<td>CLACCOUNTING</td>
<td>NEW YORK</td>
</tr>
<tr>
<td>SCOTT</td>
<td>7566</td>
<td>20</td>
<td>RESEARCH</td>
<td>DALLAS</td>
</tr>
</tbody>
</table>

(15 rows)
This style of abbreviating will be encountered quite frequently.
2.1.6 Aggregate Functions

Like most other relational database products, Postgres Plus Advanced Server supports aggregate functions. An aggregate function computes a single result from multiple input rows. For example, there are aggregates to compute the COUNT, SUM, AVG (average), MAX (maximum), and MIN (minimum) over a set of rows.

As an example, the highest and lowest salaries can be found with the following query:

```
SELECT MAX(sal) highest_salary, MIN(sal) lowest_salary FROM emp;
```

<table>
<thead>
<tr>
<th>highest_salary</th>
<th>lowest_salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>5000.00</td>
<td>800.00</td>
</tr>
</tbody>
</table>

If we wanted to find the employee with the largest salary, we may be tempted to try:

```
SELECT ename FROM emp WHERE sal = MAX(sal);
```

```
ERROR: aggregates not allowed in WHERE clause
```

This does not work because the aggregate function, MAX, cannot be used in the WHERE clause. This restriction exists because the WHERE clause determines the rows that will go into the aggregation stage so it has to be evaluated before aggregate functions are computed. However, the query can be restated to accomplish the intended result by using a subquery:

```
SELECT ename FROM emp WHERE sal = (SELECT MAX(sal) FROM emp);
```

<table>
<thead>
<tr>
<th>ename</th>
</tr>
</thead>
<tbody>
<tr>
<td>KING</td>
</tr>
</tbody>
</table>

The subquery is an independent computation that obtains its own result separately from the outer query.

Aggregates are also very useful in combination with the GROUP BY clause. For example, the following query gets the highest salary in each department.

```
SELECT deptno, MAX(sal) FROM emp GROUP BY deptno;
```

<table>
<thead>
<tr>
<th>deptno</th>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>5000.00</td>
</tr>
<tr>
<td>20</td>
<td>3000.00</td>
</tr>
<tr>
<td>30</td>
<td>2850.00</td>
</tr>
</tbody>
</table>

(3 rows)
This query produces one output row per department. Each aggregate result is computed over the rows matching that department. These grouped rows can be filtered using the HAVING clause.

```
SELECT deptno, MAX(sal) FROM emp GROUP BY deptno HAVING AVG(sal) > 2000;
```

<table>
<thead>
<tr>
<th>deptno</th>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>5000.00</td>
</tr>
<tr>
<td>20</td>
<td>3000.00</td>
</tr>
</tbody>
</table>

(2 rows)

This query gives the same results for only those departments that have an average salary greater than 2000.

Finally, the following query takes into account only the highest paid employees who are analysts in each department.

```
SELECT deptno, MAX(sal) FROM emp WHERE job = 'ANALYST' GROUP BY deptno HAVING AVG(sal) > 2000;
```

<table>
<thead>
<tr>
<th>deptno</th>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>3000.00</td>
</tr>
</tbody>
</table>

(1 row)

There is a subtle distinction between the WHERE and HAVING clauses. The WHERE clause filters out rows before grouping occurs and aggregate functions are applied. The HAVING clause applies filters on the results after rows have been grouped and aggregate functions have been computed for each group.

So in the previous example, only employees who are analysts are considered. From this subset, the employees are grouped by department and only those groups where the average salary of analysts in the group is greater than 2000 are in the final result. This is true of only the group for department 20 and the maximum analyst salary in department 20 is 3000.00.
2.1.7 Updates

The column values of existing rows can be changed using the `UPDATE` command. For example, the following sequence of commands shows the before and after results of giving everyone who is a manager a 10% raise:

```sql
SELECT ename, sal FROM emp WHERE job = 'MANAGER';

ename | sal
-------|--------
JONES  | 2975.00
BLAKE | 2850.00
CLARK | 2450.00
(3 rows)

UPDATE emp SET sal = sal * 1.1 WHERE job = 'MANAGER';

SELECT ename, sal FROM emp WHERE job = 'MANAGER';

ename | sal
-------|--------
JONES  | 3272.50
BLAKE | 3135.00
CLARK | 2695.00
(3 rows)
```
2.1.8 Deletions

Rows can be removed from a table using the DELETE command. For example, the following sequence of commands shows the before and after results of deleting all employees in department 20.

```
SELECT ename, deptno FROM emp;

ename   | deptno
--------|--------
SMITH   | 20
ALLEN   | 30
WARD    | 30
JONES   | 20
MARTIN  | 30
BLAKE   | 30
CLARK   | 10
SCOTT   | 20
KING    | 10
TURNER  | 30
ADAMS   | 20
JAMES   | 30
FORD    | 20
MILLER  | 10
(14 rows)

DELETE FROM emp WHERE deptno = 20;

SELECT ename, deptno FROM emp;

ename   | deptno
--------|--------
ALLEN   | 30
WARD    | 30
MARTIN  | 30
BLAKE   | 30
CLARK   | 10
KING    | 10
TURNER  | 30
JAMES   | 30
MILLER  | 10
(9 rows)
```

Be extremely careful of giving a DELETE command without a WHERE clause such as the following:

```
DELETE FROM tablename;
```

This statement will remove all rows from the given table, leaving it completely empty. The system will not request confirmation before doing this.
2.2 Advanced Concepts

The previous section discussed the basics of using SQL to store and access your data in Postgres Plus Advanced Server. This section discusses more advanced SQL features that may simplify management and prevent loss or corruption of your data.

2.2.1 Views

Consider the following SELECT command.

```sql
SELECT ename, sal, sal * 24 AS yearly_salary, deptno FROM emp;
```

<table>
<thead>
<tr>
<th>ename</th>
<th>sal</th>
<th>yearly_salary</th>
<th>deptno</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMITH</td>
<td>800.00</td>
<td>19200.00</td>
<td>20</td>
</tr>
<tr>
<td>ALLEN</td>
<td>1600.00</td>
<td>38400.00</td>
<td>30</td>
</tr>
<tr>
<td>WARD</td>
<td>1250.00</td>
<td>30000.00</td>
<td>30</td>
</tr>
<tr>
<td>JONES</td>
<td>2975.00</td>
<td>71400.00</td>
<td>20</td>
</tr>
<tr>
<td>MARTIN</td>
<td>1250.00</td>
<td>30000.00</td>
<td>30</td>
</tr>
<tr>
<td>BLAKE</td>
<td>2850.00</td>
<td>68400.00</td>
<td>30</td>
</tr>
<tr>
<td>CLARK</td>
<td>2450.00</td>
<td>58800.00</td>
<td>10</td>
</tr>
<tr>
<td>SCOTT</td>
<td>3000.00</td>
<td>72000.00</td>
<td>20</td>
</tr>
<tr>
<td>KING</td>
<td>5000.00</td>
<td>120000.00</td>
<td>10</td>
</tr>
<tr>
<td>TURNER</td>
<td>1500.00</td>
<td>36000.00</td>
<td>30</td>
</tr>
<tr>
<td>ADAMS</td>
<td>1100.00</td>
<td>26400.00</td>
<td>20</td>
</tr>
<tr>
<td>JAMES</td>
<td>950.00</td>
<td>22800.00</td>
<td>30</td>
</tr>
<tr>
<td>FORD</td>
<td>3000.00</td>
<td>72000.00</td>
<td>20</td>
</tr>
<tr>
<td>MILLER</td>
<td>1300.00</td>
<td>31200.00</td>
<td>10</td>
</tr>
</tbody>
</table>

(14 rows)

If this is a query that is used repeatedly, a shorthand method of reusing this query without re-typing the entire SELECT command each time is to create a view as shown below.

```sql
CREATE VIEW employee_pay AS SELECT ename, sal, sal * 24 AS yearly_salary, deptno FROM emp;
```

The view name, `employee_pay`, can now be used like an ordinary table name to perform the query.

```sql
SELECT * FROM employee_pay;
```

<table>
<thead>
<tr>
<th>ename</th>
<th>sal</th>
<th>yearly_salary</th>
<th>deptno</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMITH</td>
<td>800.00</td>
<td>19200.00</td>
<td>20</td>
</tr>
<tr>
<td>ALLEN</td>
<td>1600.00</td>
<td>38400.00</td>
<td>30</td>
</tr>
<tr>
<td>WARD</td>
<td>1250.00</td>
<td>30000.00</td>
<td>30</td>
</tr>
<tr>
<td>JONES</td>
<td>2975.00</td>
<td>71400.00</td>
<td>20</td>
</tr>
<tr>
<td>MARTIN</td>
<td>1250.00</td>
<td>30000.00</td>
<td>30</td>
</tr>
<tr>
<td>BLAKE</td>
<td>2850.00</td>
<td>68400.00</td>
<td>30</td>
</tr>
<tr>
<td>CLARK</td>
<td>2450.00</td>
<td>58800.00</td>
<td>10</td>
</tr>
<tr>
<td>SCOTT</td>
<td>3000.00</td>
<td>72000.00</td>
<td>20</td>
</tr>
<tr>
<td>KING</td>
<td>5000.00</td>
<td>120000.00</td>
<td>10</td>
</tr>
<tr>
<td>TURNER</td>
<td>1500.00</td>
<td>36000.00</td>
<td>30</td>
</tr>
<tr>
<td>ADAMS</td>
<td>1100.00</td>
<td>26400.00</td>
<td>20</td>
</tr>
<tr>
<td>JAMES</td>
<td>950.00</td>
<td>22800.00</td>
<td>30</td>
</tr>
<tr>
<td>FORD</td>
<td>3000.00</td>
<td>72000.00</td>
<td>20</td>
</tr>
</tbody>
</table>
Making liberal use of views is a key aspect of good SQL database design. Views provide a consistent interface that encapsulate details of the structure of your tables which may change as your application evolves.

Views can be used in almost any place a real table can be used. Building views upon other views is not uncommon.

### 2.2.2 Foreign Keys

Suppose you want to make sure all employees belong to a valid department. This is called maintaining the *referential integrity* of your data. In simplistic database systems this would be implemented (if at all) by first looking at the `dept` table to check if a matching record exists, and then inserting or rejecting the new employee record. This approach has a number of problems and is very inconvenient. Postgres Plus Advanced Server can make it easier for you.

A modified version of the `emp` table presented in Section 2.1.2 is shown in this section with the addition of a foreign key constraint. The modified `emp` table looks like the following:

```sql
CREATE TABLE emp (
    empno           NUMBER(4) NOT NULL CONSTRAINT emp_pk PRIMARY KEY,
    ename           VARCHAR2(10),
    job             VARCHAR2(9),
    mgr             NUMBER(4),
    hiredate        DATE,
    sal             NUMBER(7,2),
    comm            NUMBER(7,2),
    deptno          NUMBER(2) CONSTRAINT emp_ref_dept_fk REFERENCES dept(deptno) );
```

If an attempt is made to issue the following `INSERT` command in the sample `emp` table, the foreign key constraint, `emp_ref_dept_fk`, ensures that department 50 exists in the `dept` table. Since it does not, the command is rejected.

```sql
INSERT INTO emp VALUES (8000,'JONES','CLERK',7902,'17-AUG-07',1200,NULL,50);
ERROR:  insert or update on table "emp" violates foreign key constraint "emp_ref_dept_fk"
DETAIL:  Key (deptno)=(50) is not present in table "dept".
```

The behavior of foreign keys can be finely tuned to your application. Making correct use of foreign keys will definitely improve the quality of your database applications, so you are strongly encouraged to learn more about them.
2.2.3 The ROWNUM Pseudo-Column

**ROWNUM** is a pseudo-column that is assigned an incremental, unique integer value for each row based on the order the rows were retrieved from a query. Therefore, the first row retrieved will have ROWNUM of 1; the second row will have ROWNUM of 2 and so on.

This feature can be used to limit the number of rows retrieved by a query. This is demonstrated in the following example:

```
SELECT empno, ename, job FROM emp WHERE ROWNUM < 5;
```

<table>
<thead>
<tr>
<th>empno</th>
<th>ename</th>
<th>job</th>
</tr>
</thead>
<tbody>
<tr>
<td>7369</td>
<td>SMITH</td>
<td>CLERK</td>
</tr>
<tr>
<td>7499</td>
<td>ALLEN</td>
<td>SALESMAN</td>
</tr>
<tr>
<td>7521</td>
<td>WARD</td>
<td>SALESMAN</td>
</tr>
<tr>
<td>7566</td>
<td>JONES</td>
<td>MANAGER</td>
</tr>
</tbody>
</table>

(4 rows)

The ROWNUM value is assigned to each row before any sorting of the result set takes place. Thus, the result set is returned in the order given by the ORDER BY clause, but the ROWNUM values may not necessarily be in ascending order as shown in the following example:

```
SELECT ROWNUM, empno, ename, job FROM emp WHERE ROWNUM < 5 ORDER BY ename;
```

<table>
<thead>
<tr>
<th>rownum</th>
<th>empno</th>
<th>ename</th>
<th>job</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>7499</td>
<td>ALLEN</td>
<td>SALESMAN</td>
</tr>
<tr>
<td>4</td>
<td>7566</td>
<td>JONES</td>
<td>MANAGER</td>
</tr>
<tr>
<td>1</td>
<td>7369</td>
<td>SMITH</td>
<td>CLERK</td>
</tr>
<tr>
<td>3</td>
<td>7521</td>
<td>WARD</td>
<td>SALESMAN</td>
</tr>
</tbody>
</table>

(4 rows)

The following example shows how a sequence number can be added to every row in the jobhist table. First a new column named, seqno, is added to the table and then seqno is set to ROWNUM in the UPDATE command.

```
ALTER TABLE jobhist ADD seqno NUMBER(3);
UPDATE jobhist SET seqno = ROWNUM;
```

The following SELECT command shows the new seqno values.

```
SELECT seqno, empno, TO_CHAR(startdate,'DD-MON-YY') AS start, job FROM jobhist;
```

<table>
<thead>
<tr>
<th>seqno</th>
<th>empno</th>
<th>start</th>
<th>job</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7369</td>
<td>17-DEC-80</td>
<td>CLERK</td>
</tr>
<tr>
<td>2</td>
<td>7499</td>
<td>20-FEB-81</td>
<td>SALESMAN</td>
</tr>
<tr>
<td>3</td>
<td>7521</td>
<td>22-FEB-81</td>
<td>SALESMAN</td>
</tr>
<tr>
<td>4</td>
<td>7566</td>
<td>02-APR-81</td>
<td>MANAGER</td>
</tr>
<tr>
<td>5</td>
<td>7654</td>
<td>28-SEP-81</td>
<td>SALESMAN</td>
</tr>
<tr>
<td>6</td>
<td>7698</td>
<td>01-MAY-81</td>
<td>MANAGER</td>
</tr>
</tbody>
</table>
2.2.4 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.

Postgres Plus Advanced Server supports synonyms for:

- tables
- views
- sequences
- procedures
- functions
- 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 Oracle-compatible 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 |         |     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 Oracle-compatible `DROP PUBLIC SYNONYM` command drops a synonym that resides in the `public` schema:

```sql
DROP PUBLIC SYNONYM syn_name;
```

The following example drops the synonym, `personnel`:

```sql
DROP SYNONYM personnel;
```
2.2.5 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.

```sql
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.

2.2.5.1 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 (=)

Copyright © 2007 - 2015 EnterpriseDB Corporation. All rights reserved.
operator. In addition, one of these two expressions must be preceded by the keyword, PRIOR.

For any given row, to determine its children:

1. Evaluate parent_expr on the given row
2. Evaluate child_expr on any other row resulting from the evaluation of table_expression
3. If parent_expr = child_expr, then this row is a child node of the given parent row
4. 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.

2.2.5.2 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.

2.2.5.3 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.
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 is a direct report of 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.

```sql
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.

<table>
<thead>
<tr>
<th>ename</th>
<th>empno</th>
<th>mgr</th>
</tr>
</thead>
<tbody>
<tr>
<td>KING</td>
<td>7839</td>
<td></td>
</tr>
<tr>
<td>JONES</td>
<td>7566</td>
<td>7839</td>
</tr>
<tr>
<td>SCOTT</td>
<td>7788</td>
<td>7566</td>
</tr>
<tr>
<td>ADAMS</td>
<td>7876</td>
<td>7788</td>
</tr>
<tr>
<td>FORD</td>
<td>7902</td>
<td>7566</td>
</tr>
<tr>
<td>SMITH</td>
<td>7369</td>
<td>7902</td>
</tr>
<tr>
<td>BLAKE</td>
<td>7698</td>
<td>7839</td>
</tr>
</tbody>
</table>
2.2.5.4 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.

```sql
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
```

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.
2.2.5.5 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.

<table>
<thead>
<tr>
<th>level</th>
<th>employee</th>
<th>empno</th>
<th>mgr</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>KING</td>
<td>7839</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>BLAKE</td>
<td>7698</td>
<td>7839</td>
</tr>
<tr>
<td>3</td>
<td>ALLEN</td>
<td>7499</td>
<td>7698</td>
</tr>
<tr>
<td>3</td>
<td>JAMES</td>
<td>7900</td>
<td>7698</td>
</tr>
<tr>
<td>3</td>
<td>MARTIN</td>
<td>7654</td>
<td>7698</td>
</tr>
<tr>
<td>3</td>
<td>TURNER</td>
<td>7844</td>
<td>7698</td>
</tr>
<tr>
<td>3</td>
<td>WARD</td>
<td>7521</td>
<td>7698</td>
</tr>
<tr>
<td>2</td>
<td>CLARK</td>
<td>7782</td>
<td>7839</td>
</tr>
<tr>
<td>3</td>
<td>MILLER</td>
<td>7934</td>
<td>7782</td>
</tr>
<tr>
<td>2</td>
<td>JONES</td>
<td>7566</td>
<td>7839</td>
</tr>
<tr>
<td>3</td>
<td>FORD</td>
<td>7902</td>
<td>7566</td>
</tr>
<tr>
<td>4</td>
<td>SMITH</td>
<td>7369</td>
<td>7902</td>
</tr>
<tr>
<td>3</td>
<td>SCOTT</td>
<td>7788</td>
<td>7566</td>
</tr>
<tr>
<td>4</td>
<td>ADAMS</td>
<td>7876</td>
<td>7788</td>
</tr>
</tbody>
</table>

(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.

<table>
<thead>
<tr>
<th>level</th>
<th>employee</th>
<th>empno</th>
<th>mgr</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>BLAKE</td>
<td>7698</td>
<td>7839</td>
</tr>
<tr>
<td>1</td>
<td>CLARK</td>
<td>7782</td>
<td>7839</td>
</tr>
<tr>
<td></td>
<td>MILLER</td>
<td>7934</td>
<td>7782</td>
</tr>
<tr>
<td>---</td>
<td>--------</td>
<td>-------</td>
<td>------</td>
</tr>
<tr>
<td>1</td>
<td>JONES</td>
<td>7566</td>
<td>7839</td>
</tr>
<tr>
<td>3</td>
<td>SMITH</td>
<td>7369</td>
<td>7902</td>
</tr>
<tr>
<td>3</td>
<td>ADAMS</td>
<td>7876</td>
<td>7788</td>
</tr>
</tbody>
</table>

(6 rows)
3 The SQL Language

The following sections describe the Oracle-compatible subset of the Postgres Plus Advanced Server SQL language. The SQL syntax, commands, data types, and functions that follow work in both Postgres Plus Advanced Server and in Oracle.

Other aspects of the Postgres Plus Advanced Server SQL language that are not Oracle compatible can be found in the Postgres Plus documentation set. The Postgres Plus documentation set includes syntax and commands for extended functionality not included in this guide.

This section is organized into the following sections:

- General discussion of Postgres Plus Advanced Server SQL syntax and language elements
- Data types
- Summary of SQL commands
- Built-in functions

3.1 SQL Syntax

This section describes the general syntax of SQL. It forms the foundation for understanding the following chapters that include detail about how the SQL commands are applied to define and modify data.

3.1.1 Lexical Structure

SQL input consists of a sequence of commands. A command is composed of a sequence of tokens, terminated by a semicolon (;). The end of the input stream also terminates a command. Which tokens are valid depends on the syntax of the particular command.

A token can be a key word, an identifier, a quoted identifier, a literal (or constant), or a special character symbol. Tokens are normally separated by whitespace (space, tab, newline), but need not be if there is no ambiguity (which is generally only the case if a special character is adjacent to some other token type).

Additionally, comments can occur in SQL input. They are not tokens - they are effectively equivalent to whitespace.

For example, the following is (syntactically) valid SQL input:

```
SELECT * FROM MY_TABLE;
UPDATE MY_TABLE SET A = 5;
INSERT INTO MY_TABLE VALUES (3, 'hi there');
```
This is a sequence of three commands, one per line (although this is not required; more than one command can be on a line, and commands can usually be split across lines).

The SQL syntax is not very consistent regarding what tokens identify commands and which are operands or parameters. The first few tokens are generally the command name, so in the above example we would usually speak of a SELECT, an UPDATE, and an INSERT command. But for instance the UPDATE command always requires a SET token to appear in a certain position, and this particular variation of INSERT also requires a VALUES token in order to be complete. The precise syntax rules for each command are described in Section 3.3.

3.1.2 Identifiers and Key Words

Tokens such as SELECT, UPDATE, or VALUES in the example above are examples of key words, that is, words that have a fixed meaning in the SQL language. The tokens MY_TABLE and A are examples of identifiers. They identify names of tables, columns, or other database objects, depending on the command they are used in. Therefore they are sometimes simply called, “names”. Key words and identifiers have the same lexical structure, meaning that one cannot know whether a token is an identifier or a key word without knowing the language.

SQL identifiers and key words must begin with a letter (a-z or A-Z). Subsequent characters in an identifier or key word can be letters, underscores, digits (0-9), dollar signs ($), or number signs (#).

Identifier and key word names are case insensitive. Therefore

```
UPDATE MY_TABLE SET A = 5;
```

can equivalently be written as:

```
UPDaTE my_Table SeT a = 5;
```

A convention often used is to write key words in upper case and names in lower case, e.g.,

```
UPDATE my_table SET a = 5;
```

There is a second kind of identifier: the delimited identifier or quoted identifier. It is formed by enclosing an arbitrary sequence of characters in double-quotes ("'). A delimited identifier is always an identifier, never a key word. So "select" could be used to refer to a column or table named "select", whereas an unquoted select would be taken as a key word and would therefore provoke a parse error when used where a table or column name is expected. The example can be written with quoted identifiers like this:

```
UPDATE "my_table" SET "a" = 5;
```
Quoted identifiers can contain any character, except the character with code zero. (To include a double quote, write two double quotes.) This allows constructing table or column names that would otherwise not be possible, such as ones containing spaces or ampersands. The length limitation still applies.

Quoting an identifier also makes it case-sensitive, whereas unquoted names are always folded to lower case. For example, the identifiers FOO, foo, and "foo" are considered the same by Postgres Plus Advanced Server, but "Foo" and "FOO" are different from these three and each other. (The folding of unquoted names to lower case in Postgres Plus Advanced Server is an area of Oracle-incompatibility. In Oracle unquoted names are folded to upper case. Thus, foo is equivalent to "FOO" not "foo" in Oracle. If you want to write portable applications you are advised to always quote a particular name or never quote it.)
3.1.3 Constants

The kinds of implicitly-typed constants in Postgres Plus Advanced Server are *strings* and *numbers*. Constants can also be specified with explicit types, which can enable more accurate representation and more efficient handling by the system. These alternatives are discussed in the following subsections.

3.1.3.1 String Constants

A *string constant* in SQL is an arbitrary sequence of characters bounded by single quotes ('), for example 'This is a string'. To include a single-quote character within a string constant, write two adjacent single quotes, e.g. 'Dianne''s horse'. Note that this is not the same as a double-quote character (").

3.1.3.2 Numeric Constants

Numeric constants are accepted in these general forms:

\[
\begin{align*}
\text{digits} \\
\text{digits}[\text{digits}\text{[e[+]\text{-}\text{digits}]}] \\
\text{[digits].digits[e[+]\text{-}\text{digits}]} \\
\text{digitse[+]\text{-}\text{digits}}
\end{align*}
\]

where *digits* is one or more decimal digits (0 through 9). At least one digit must be before or after the decimal point, if one is used. At least one digit must follow the exponent marker (e), if one is present. There may not be any spaces or other characters embedded in the constant. Note that any leading plus or minus sign is not actually considered part of the constant; it is an operator applied to the constant.

These are some examples of valid numeric constants:

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>42</td>
<td>3.5</td>
<td>4.</td>
<td>.001</td>
<td>5e2</td>
</tr>
<tr>
<td>1.925e-3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A numeric constant that contains neither a decimal point nor an exponent is initially presumed to be type **INTEGER** if its value fits in type **INTEGER** (32 bits); otherwise it is presumed to be type **BIGINT** if its value fits in type **BIGINT** (64 bits); otherwise it is taken to be type **NUMBER**. Constants that contain decimal points and/or exponents are always initially presumed to be type **NUMBER**.

The initially assigned data type of a numeric constant is just a starting point for the type resolution algorithms. In most cases the constant will be automatically coerced to the
most appropriate type depending on context. When necessary, you can force a numeric value to be interpreted as a specific data type by casting it as described in the following section.

3.1.3.3 Constants of Other Types

A constant of an arbitrary type can be entered using the following notation:

```
CAST('string' AS type)
```

The string constant’s text is passed to the input conversion routine for the type called `type`. The result is a constant of the indicated type. The explicit type cast may be omitted if there is no ambiguity as to the type the constant must be (for example, when it is assigned directly to a table column), in which case it is automatically coerced.

`CAST` can also be used to specify runtime type conversions of arbitrary expressions.

3.1.4 Comments

A comment is an arbitrary sequence of characters beginning with double dashes and extending to the end of the line, e.g.:

```
-- This is a standard SQL comment
```

Alternatively, C-style block comments can be used:

```
/* multiline comment
 * block
 */
```

where the comment begins with `/*` and extends to the matching occurrence of `*/`. 
3.2 Data Types

The following table shows the built-in general-purpose data types.

Table 3-3-1 Data Types

<table>
<thead>
<tr>
<th>Name</th>
<th>Alias</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLOB</td>
<td>LONG RAW, RAW(n)</td>
<td>Binary data</td>
</tr>
<tr>
<td>BOOLEAN</td>
<td></td>
<td>Logical Boolean (true/false)</td>
</tr>
<tr>
<td>CHAR [ (n) ]</td>
<td>CHARACTER [ (n) ]</td>
<td>Fixed-length character string of n characters</td>
</tr>
<tr>
<td>CLOB</td>
<td>LONG, LONG VARCHAR</td>
<td>Long character string</td>
</tr>
<tr>
<td>DATE</td>
<td>TIMESTAMP(0)</td>
<td>Date and time to the second</td>
</tr>
<tr>
<td>DOUBLE PRECISION</td>
<td>FLOAT, FLOAT(25) - FLOAT(53)</td>
<td>Double precision floating-point number</td>
</tr>
<tr>
<td>INTEGER</td>
<td>INT, BINARY_INTEGER, PLS_INTEGER</td>
<td>Signed four-byte integer</td>
</tr>
<tr>
<td>NUMBER</td>
<td>DEC, DECIMAL, NUMERIC</td>
<td>Exact numeric with optional decimal places</td>
</tr>
<tr>
<td>NUMBER(p [, s])</td>
<td>DEC(p [, s]), DECIMAL(p [, s]), NUMERIC(p [, s])</td>
<td>Exact numeric of maximum precision, p, and optional scale, s</td>
</tr>
<tr>
<td>REAL</td>
<td>FLOAT(1) - FLOAT(24)</td>
<td>Single precision floating-point number</td>
</tr>
<tr>
<td>TIMESTAMP [ (p) ]</td>
<td></td>
<td>Date and time with optional, fractional second precision, p</td>
</tr>
<tr>
<td>TIMESTAMP [ (p) ] WITH TIME ZONE</td>
<td></td>
<td>Date and time with optional, fractional second precision, p, and with time zone</td>
</tr>
<tr>
<td>VARCHAR2(n)</td>
<td>CHAR VARYING(n), CHARACTER VARYING(n), VARCHAR(n)</td>
<td>Variable-length character string with a maximum length of n characters</td>
</tr>
<tr>
<td>XMLTYPE</td>
<td></td>
<td>XML data</td>
</tr>
</tbody>
</table>

The following sections describe each data type in more detail.
3.2.1 Numeric Types

Numeric types consist of four-byte integers, four-byte and eight-byte floating-point numbers, and fixed-precision decimals. The following table lists the available types.

Table 3-3-2 Numeric Types

<table>
<thead>
<tr>
<th>Name</th>
<th>Storage Size</th>
<th>Description</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>BINARY_INTEGER</td>
<td>4 bytes</td>
<td>Signed integer, Alias for INTEGER</td>
<td>-2,147,483,648 to +2,147,483,647</td>
</tr>
<tr>
<td>DOUBLE PRECISION</td>
<td>8 bytes</td>
<td>Variable-precision, inexact</td>
<td>15 decimal digits precision</td>
</tr>
<tr>
<td>INTEGER</td>
<td>4 bytes</td>
<td>Usual choice for integer</td>
<td>-2,147,483,648 to +2,147,483,647</td>
</tr>
<tr>
<td>NUMBER</td>
<td>Variable</td>
<td>User-specified precision, exact</td>
<td>Up to 1000 digits of precision</td>
</tr>
<tr>
<td>NUMBER(p [, s ] )</td>
<td>Variable</td>
<td>Exact numeric of maximum precision, p, and optional scale, s</td>
<td>Up to 1000 digits of precision</td>
</tr>
<tr>
<td>PLS_INTEGER</td>
<td>4 bytes</td>
<td>Signed integer, Alias for INTEGER</td>
<td>-2,147,483,648 to +2,147,483,647</td>
</tr>
<tr>
<td>REAL</td>
<td>4 bytes</td>
<td>Variable-precision, inexact</td>
<td>6 decimal digits precision</td>
</tr>
<tr>
<td>ROWID</td>
<td>8 bytes</td>
<td>Signed 8 bit integer.</td>
<td>-9223372036854775808 to 9223372036854775807</td>
</tr>
</tbody>
</table>

The following sections describe the types in detail.

3.2.1.1 Integer Types

The type, INTEGER, stores whole numbers (without fractional components) between the values of -2,147,483,648 and +2,147,483,647. Attempts to store values outside of the allowed range will result in an error.

Columns of the ROWID type holds fixed-length binary data that describes the physical address of a record. ROWID is an unsigned, four-byte INTEGER that stores whole numbers (without fractional components) between the values of 0 and 4,294,967,295. Attempts to store values outside of the allowed range will result in an error.

3.2.1.2 Arbitrary Precision Numbers

The type, NUMBER, can store practically an unlimited number of digits of precision and perform calculations exactly. It is especially recommended for storing monetary amounts and other quantities where exactness is required. However, the NUMBER type is very slow compared to the floating-point types described in the next section.
In what follows we use these terms: The scale of a NUMBER is the count of decimal digits in the fractional part, to the right of the decimal point. The precision of a NUMBER is the total count of significant digits in the whole number, that is, the number of digits to both sides of the decimal point. So the number 23.5141 has a precision of 6 and a scale of 4. Integers can be considered to have a scale of zero.

Both the precision and the scale of the NUMBER type can be configured. To declare a column of type NUMBER use the syntax

\[
\text{NUMBER(}\ precision, \ scale)\]

The precision must be positive, the scale zero or positive. Alternatively,

\[
\text{NUMBER(}\ precision)\]

selects a scale of 0. Specifying NUMBER without any precision or scale creates a column in which numeric values of any precision and scale can be stored, up to the implementation limit on precision. A column of this kind will not coerce input values to any particular scale, whereas NUMBER columns with a declared scale will coerce input values to that scale. (The SQL standard requires a default scale of 0, i.e., coercion to integer precision. For maximum portability, it is best to specify the precision and scale explicitly.)

If the precision or scale of a value is greater than the declared precision or scale of a column, the system will attempt to round the value. If the value cannot be rounded so as to satisfy the declared limits, an error is raised.

### 3.2.1.3 Floating-Point Types

The data types REAL and DOUBLE PRECISION are inexact, variable-precision numeric types. In practice, these types are usually implementations of IEEE Standard 754 for Binary Floating-Point Arithmetic (single and double precision, respectively), to the extent that the underlying processor, operating system, and compiler support it.

Inexact means that some values cannot be converted exactly to the internal format and are stored as approximations, so that storing and printing back out a value may show slight discrepancies. Managing these errors and how they propagate through calculations is the subject of an entire branch of mathematics and computer science and will not be discussed further here, except for the following points:

If you require exact storage and calculations (such as for monetary amounts), use the NUMBER type instead.

If you want to do complicated calculations with these types for anything important, especially if you rely on certain behavior in boundary cases (infinity, underflow), you should evaluate the implementation carefully.
Comparing two floating-point values for equality may or may not work as expected.

On most platforms, the REAL type has a range of at least 1E-37 to 1E+37 with a precision of at least 6 decimal digits. The DOUBLE PRECISION type typically has a range of around 1E-307 to 1E+308 with a precision of at least 15 digits. Values that are too large or too small will cause an error. Rounding may take place if the precision of an input number is too high. Numbers too close to zero that are not representable as distinct from zero will cause an underflow error.

Postgres Plus Advanced Server also supports the SQL standard notations FLOAT and FLOAT(p) for specifying inexact numeric types. Here, p specifies the minimum acceptable precision in binary digits. Postgres Plus Advanced Server accepts FLOAT(1) to FLOAT(24) as selecting the REAL type, while FLOAT(25) to FLOAT(53) as selecting DOUBLE PRECISION. Values of p outside the allowed range draw an error. FLOAT with no precision specified is taken to mean DOUBLE PRECISION.
3.2.2 Character Types

The following table lists the general-purpose character types available in Postgres Plus Advanced Server.

Table 3-3-3 Character Types

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR[ (n) ]</td>
<td>Fixed-length character string, blank-padded to the size specified by n</td>
</tr>
<tr>
<td>CLOB</td>
<td>Large variable-length up to 1 GB</td>
</tr>
<tr>
<td>LONG</td>
<td>Variable unlimited length.</td>
</tr>
<tr>
<td>NVARCHAR (n)</td>
<td>Variable-length national character string, with limit.</td>
</tr>
<tr>
<td>VARCHAR2 (n)</td>
<td>Variable-length national character string, with limit.</td>
</tr>
<tr>
<td>STRING</td>
<td>Alias for VARCHAR2.</td>
</tr>
<tr>
<td>VARCHAR (n)</td>
<td>Variable-length character string, with limit (considered deprecated, but supported for compatibility)</td>
</tr>
<tr>
<td>VARCHAR2 (n)</td>
<td>Variable-length character string, with limit</td>
</tr>
</tbody>
</table>

Where \( n \) is a positive integer; these types can store strings up to \( n \) characters in length. An attempt to assign a value that exceeds the length of \( n \) will result in an error, unless the excess characters are all spaces, in which case the string will be truncated to the maximum length.

**CHAR**

If you do not specify a value for \( n \), \( n \) will default to 1. If the string to be assigned is shorter than \( n \), values of type CHAR will be space-padded to the specified width \( (n) \), and will be stored and displayed that way.

Padding spaces are treated as semantically insignificant. That is, trailing spaces are disregarded when comparing two values of type CHAR, and they will be removed when converting a CHAR value to one of the other string types.

If you explicitly cast an over-length value to a CHAR \( (n) \) type, the value will be truncated to \( n \) characters without raising an error (as specified by the SQL standard).

**VARCHAR, VARCHAR2, NVARCHAR and NVARCHAR2**

If the string to be assigned is shorter than \( n \), values of type VARCHAR, VARCHAR2, NVARCHAR and NVARCHAR2 will store the shorter string without padding.

Note that trailing spaces are semantically significant in VARCHAR values.
If you explicitly cast a value to a VARCHAR type, an over-length value will be truncated to \( n \) characters without raising an error (as specified by the SQL standard).

**CLOB**

You can store a large character string in a CLOB type. CLOB is semantically equivalent to VARCHAR2 except no length limit is specified. Generally, you should use a CLOB type if the maximum string length is not known.

The longest possible character string that can be stored in a CLOB type is about 1 GB.

The storage requirement for data of these types is the actual string plus 1 byte if the string is less than 127 bytes, or 4 bytes if the string is 127 bytes or greater. In the case of CHAR, the padding also requires storage. Long strings are compressed by the system automatically, so the physical requirement on disk may be less. Long values are stored in background tables so they do not interfere with rapid access to the shorter column values.

The database character set determines the character set used to store textual values.
### 3.2.3 Binary Data

The following data types allows storage of binary strings.

#### Table 3-3-4 Binary Large Object

<table>
<thead>
<tr>
<th>Name</th>
<th>Storage Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BINARY</td>
<td>The length of the binary string.</td>
<td>Fixed-length binary string, with a length between 1 and 8300.</td>
</tr>
<tr>
<td>BLOB</td>
<td>The actual binary string plus 1 byte if the binary string is less than 127 bytes, or 4 bytes if the binary string is 127 bytes or greater.</td>
<td>Variable-length binary string</td>
</tr>
<tr>
<td>VARBINARY</td>
<td>The length of the binary string</td>
<td>Variable-length binary string, with a length between 1 and 8300.</td>
</tr>
</tbody>
</table>

A binary string is a sequence of octets (or bytes). Binary strings are distinguished from characters strings by two characteristics: First, binary strings specifically allow storing octets of value zero and other "non-printable" octets (defined as octets outside the range 32 to 126). Second, operations on binary strings process the actual bytes, whereas the encoding and processing of character strings depends on locale settings.
3.2.4 Date/Time Types

The following discussion of the date/time types assumes that the configuration parameter, `edb_redwood_date`, has been set to `TRUE` whenever a table is created or altered.

Postgres Plus Advanced Server supports the date/time types shown in the following table.

<table>
<thead>
<tr>
<th>Name</th>
<th>Storage Size</th>
<th>Description</th>
<th>Low Value</th>
<th>High Value</th>
<th>Resolution</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATE</td>
<td>8 bytes</td>
<td>Date and time</td>
<td>4713 BC</td>
<td>5874897 AD</td>
<td>1 second</td>
</tr>
<tr>
<td>INTERVAL DAY TO SECOND</td>
<td>12 bytes</td>
<td>Period of time</td>
<td>-1780000000 years</td>
<td>178000000 years</td>
<td>1 microsecond / 14 digits</td>
</tr>
<tr>
<td>INTERVAL YEAR TO MONTH</td>
<td>12 bytes</td>
<td>Period of time</td>
<td>-1780000000 years</td>
<td>178000000 years</td>
<td>1 microsecond / 14 digits</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>8 bytes</td>
<td>Date and time</td>
<td>4713 BC</td>
<td>5874897 AD</td>
<td>1 microsecond</td>
</tr>
<tr>
<td>TIMESTAMP WITH TIME ZONE</td>
<td>8 bytes</td>
<td>Date and time with time zone</td>
<td>4713 BC</td>
<td>5874897 AD</td>
<td>1 microsecond</td>
</tr>
</tbody>
</table>

When `DATE` appears as the data type of a column in the data definition language (DDL) commands, `CREATE TABLE` or `ALTER TABLE`, it is translated to `TIMESTAMP(0)` at the time the table definition is stored in the database. Thus, a time component will also be stored in the column along with the date.

When `DATE` appears as a data type of a variable in an SPL declaration section, or the data type of a formal parameter in an SPL procedure or an SPL function, or the return type of an SPL function, it is always translated to `TIMESTAMP(0)` and thus can handle a time component if present.

`TIMESTAMP` accepts an optional precision value `p` which specifies the number of fractional digits retained in the seconds field. The allowed range of `p` is from 0 to 6 with the default being 6.

When `TIMESTAMP` values are stored as double precision floating-point numbers (currently the default), the effective limit of precision may be less than 6. `TIMESTAMP` values are stored as seconds before or after midnight 2000-01-01. Microsecond precision is achieved for dates within a few years of 2000-01-01, but the precision degrades for dates further away. When `TIMESTAMP` values are stored as eight-byte integers (a compile-time option), microsecond precision is available over the full range of values. However eight-byte integer timestamps have a more limited range of dates than shown above: from 4713 BC up to 294276 AD.

`TIMESTAMP (p) WITH TIME ZONE` is similar to `TIMESTAMP (p)`, but includes the time zone as well.
3.2.4.1 INTERVAL Types

INTERVAL values specify a period of time. Values of INTERVAL type are composed of fields that describe the value of the data. The following table lists the fields allowed in an INTERVAL type:

<table>
<thead>
<tr>
<th>Field Name</th>
<th>INTERVAL Values Allowed</th>
</tr>
</thead>
<tbody>
<tr>
<td>YEAR</td>
<td>Integer value (positive or negative)</td>
</tr>
<tr>
<td>MONTH</td>
<td>0 through 11</td>
</tr>
<tr>
<td>DAY</td>
<td>Integer value (positive or negative)</td>
</tr>
<tr>
<td>HOUR</td>
<td>0 through 23</td>
</tr>
<tr>
<td>MINUTE</td>
<td>0 through 59</td>
</tr>
<tr>
<td>SECOND</td>
<td>0 through 59.9((p)) where 9((p)) is the precision of fractional seconds</td>
</tr>
</tbody>
</table>

The fields must be presented in descending order – from YEARS to MONTHS, and from DAYS to HOURS, MINUTES and then SECONDS.

Advanced Server supports two Oracle-compatible INTERVAL types.

The first variation supported by Advanced Server is INTERVAL DAY TO SECOND \([ (p) ]\). INTERVAL DAY TO SECOND \([ (p) ]\) stores a time interval in days, hours, minutes and seconds.

\(p\) specifies the precision of the second field.

Advanced Server interprets the value:

\[
\text{INTERVAL } '1 \ 2:34:5.678' \ \text{DAY TO SECOND}(3)
\]

as 1 day, 2 hours, 34 minutes, 5 seconds and 678 thousandths of a second.

Advanced Server interprets the value:

\[
\text{INTERVAL } '1 \ 23' \ \text{DAY TO HOUR}
\]

as 1 day and 23 hours.

Advanced Server interprets the value:

\[
\text{INTERVAL } '2:34' \ \text{HOUR TO MINUTE}
\]

as 2 hours and 34 minutes.

Advanced Server interprets the value:

\[
\text{INTERVAL } '2:34:56.129' \ \text{HOUR TO SECOND}(2)
\]
as 2 hours, 34 minutes, 56 seconds and 13 thousandths of a second. Note that the fractional second is rounded up to 13 because of the specified precision.

The second Oracle-compatible variation supported by Advanced Server is \texttt{INTERVAL YEAR TO MONTH}. This variation stores a time interval in years and months.

Advanced Server interprets the value:

\begin{verbatim}
INTERVAL '12-3' YEAR TO MONTH
\end{verbatim}

as 12 years and 3 months.

Advanced Server interprets the value:

\begin{verbatim}
INTERVAL '456' YEAR(2)
\end{verbatim}

as 12 years and 3 months.

Advanced Server interprets the value:

\begin{verbatim}
INTERVAL '300' MONTH
\end{verbatim}

as 25 years.

\subsection*{3.2.4.2 Date/Time Input}

Date and time input is accepted in ISO 8601 SQL-compatible format, the Oracle default \texttt{dd-MON-yy} format, as well as a number of other formats provided that there is no ambiguity as to which component is the year, month, and day. However, use of the \texttt{TO_DATE} function is strongly recommended to avoid ambiguities. See Section 3.5.7.

Any date or time literal input needs to be enclosed in single quotes, like text strings. The following SQL standard syntax is also accepted:

\begin{verbatim}
type 'value'
\end{verbatim}

\texttt{type} is either \texttt{DATE} or \texttt{TIMESTAMP}.

\texttt{value} is a date/time text string.

\subsubsection*{3.2.4.2.1 Dates}

The following table shows some possible input formats for dates, all of which equate to January 8, 1999.
Table 3-3-6 Date Input

<table>
<thead>
<tr>
<th>Example</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>January 8, 1999</td>
<td></td>
</tr>
<tr>
<td>1999-01-08</td>
<td></td>
</tr>
<tr>
<td>1999-Jan-08</td>
<td></td>
</tr>
<tr>
<td>Jan-08-1999</td>
<td></td>
</tr>
<tr>
<td>08-Jan-1999</td>
<td></td>
</tr>
<tr>
<td>08-Jan-99</td>
<td></td>
</tr>
<tr>
<td>Jan-08-99</td>
<td></td>
</tr>
<tr>
<td>19990108</td>
<td></td>
</tr>
<tr>
<td>990108</td>
<td></td>
</tr>
</tbody>
</table>

The date values can be assigned to a `DATE` or `TIMESTAMP` column or variable. The hour, minute, and seconds fields will be set to zero if the date value is not appended with a time value.

3.2.4.2.2 Times

Some examples of the time component of a date or time stamp are shown in the following table.

Table 3-3-7 Time Input

<table>
<thead>
<tr>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>04:05:06.789</td>
<td>ISO 8601</td>
</tr>
<tr>
<td>04:05:06</td>
<td>ISO 8601</td>
</tr>
<tr>
<td>04:05</td>
<td>ISO 8601</td>
</tr>
<tr>
<td>040506</td>
<td>ISO 8601</td>
</tr>
<tr>
<td>04:05 AM</td>
<td>Same as 04:05; AM does not affect value</td>
</tr>
<tr>
<td>04:05 PM</td>
<td>Same as 16:05; input hour must be &lt;= 12</td>
</tr>
</tbody>
</table>

3.2.4.2.3 Time Stamps

Valid input for time stamps consists of a concatenation of a date and a time. The date portion of the time stamp can be formatted according to any of the examples shown in Table 3-3-6 Date Input. The time portion of the time stamp can be formatted according to any of examples shown in Table 3-3-7 Time Input.

The following is an example of a time stamp which follows the Oracle default format.

```
08-JAN-99 04:05:06
```

The following is an example of a time stamp which follows the ISO 8601 standard.

```
1999-01-08 04:05:06
```
3.2.4.3 Date/Time Output

The default output format of the date/time types will be either the Oracle compatible style (dd-MON-yy) referred to as the Redwood date style, or the ISO 8601 format (yyyy-mm-dd) depending upon the application interface to the database. Applications that use JDBC such as SQL Interactive always present the date in ISO 8601 form. Other applications such as PSQL present the date in Redwood form.

The following table shows examples of the output formats for the two styles, Redwood and ISO 8601.

Table 3-3-8 Date/Time Output Styles

<table>
<thead>
<tr>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Redwood style</td>
<td>31-DEC-05 07:37:16</td>
</tr>
<tr>
<td>ISO 8601/SQL standard</td>
<td>1997-12-17 07:37:16</td>
</tr>
</tbody>
</table>

3.2.4.4 Internals

Postgres Plus Advanced Server uses Julian dates for all date/time calculations. Julian dates correctly predict or calculate any date after 4713 BC based on the assumption that the length of the year is 365.2425 days.
3.2.5 Boolean Type

Postgres Plus Advanced Server provides the standard SQL type BOOLEAN. BOOLEAN can have one of only two states: TRUE or FALSE. A third state, UNKNOWN, is represented by the SQL NULL value.

Table 3-3-9 Boolean Type

<table>
<thead>
<tr>
<th>Name</th>
<th>Storage Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOLEAN</td>
<td>1 byte</td>
<td>Logical Boolean (true/false)</td>
</tr>
</tbody>
</table>

The valid literal value for representing the true state is TRUE. The valid literal for representing the false state is FALSE.
3.2.6 XML Type

The XMLTYPE data type is used to store XML data. Its advantage over storing XML data in a character field is that it checks the input values for well-formedness, and there are support functions to perform type-safe operations on it.

The XML type can store well-formed “documents”, as defined by the XML standard, as well as “content” fragments, which are defined by the production XMLDecl? content in the XML standard. Roughly, this means that content fragments can have more than one top-level element or character node.

Note: Oracle does not support the storage of content fragments in XMLTYPE columns.

The following example shows the creation and insertion of a row into a table with an XMLTYPE column.

```sql
CREATE TABLE books (
    content XMLTYPE
);

INSERT INTO books VALUES (XMLPARSE (DOCUMENT '<?xml version="1.0"?><book><title>Manual</title><chapter>...</chapter></book>');

SELECT * FROM books;

<table>
<thead>
<tr>
<th>content</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;book&gt;&lt;title&gt;Manual&lt;/title&gt;&lt;chapter&gt;...&lt;/chapter&gt;&lt;/book&gt;</td>
</tr>
</tbody>
</table>
```

(1 row)
3.3 SQL Commands

This section provides a summary of the Oracle compatible SQL commands supported by Postgres Plus Advanced Server. The SQL commands in this section will work on both an Oracle database and a Postgres Plus Advanced Server database.

Note the following points:

- Postgres Plus Advanced Server supports other commands that are not listed here. These commands may have no Oracle equivalent or they may provide the similar or same functionality as an Oracle SQL command, but with different syntax.
- The SQL commands in this section do not necessarily represent the full syntax, options, and functionality available in the command. Syntax, options, and functionality that are not Oracle compatible have been omitted from the command description and syntax.
- The Postgres Plus documentation set contains aspects of the command that may not be Oracle compatible.
3.3.1 ALTER INDEX

Name

ALTER INDEX -- change the definition of an index

Synopsis

ALTER INDEX name RENAME TO new_name

Description

ALTER INDEX changes the definition of an existing index. RENAME changes the name of the index. There is no effect on the stored data.

Parameters

name

The name (possibly schema-qualified) of an existing index to alter.

ew_name

New name for the index.

Examples

To rename an existing index:

ALTER INDEX name_idx RENAME TO empname_idx;

See Also

CREATE INDEX, DROP INDEX
### 3.3.2 ALTER PROCEDURE

**Name**

`ALTER PROCEDURE`

**Synopsis**

`ALTER PROCEDURE procedure_name options [RESTRICT]`

**Description**

Use the `ALTER PROCEDURE` statement to specify that a procedure is a `SECURITY INVOKER` or `SECURITY DEFINER`.

**Parameters**

- `procedure_name`
  
  `procedure_name` specifies the (possibly schema-qualified) name of a stored procedure.

- `options` may be:
  
  `[EXTERNAL] SECURITY DEFINER`

  Specify `SECURITY DEFINER` to instruct the server to execute the procedure with the privileges of the user that created the procedure. The `EXTERNAL` keyword is accepted for compatibility, but ignored.

  `[EXTERNAL] SECURITY INVOKER`

  Specify `SECURITY INVOKER` to instruct the server to execute the procedure with the privileges of the user that is invoking the procedure. The `EXTERNAL` keyword is accepted for compatibility, but ignored.

The `RESTRICT` keyword is accepted for compatibility, but ignored.

**Examples**

The following command specifies that the `update_balance` procedure should execute with the privileges of the user invoking the procedure:

```
ALTER PROCEDURE update_balance SECURITY INVOKER;
```
### 3.3.3 ALTER ROLE

**Name**

ALTER ROLE -- change a database role

**Synopsis**

ALTER ROLE role_name IDENTIFIED BY password

**Description**

ALTER ROLE changes the password of a role. Only superusers or users with the CREATEROLE attribute can use this command. If the role to be altered has the SUPERUSER attribute, then only a superuser can give this command. Note that unless the role has the LOGIN attribute, the password serves no real purpose.

**Parameters**

*role_name*

The name of the role whose password is to be altered.

*password*

The role’s new password.

**Notes**

You can use GRANT and REVOKE to change a role’s memberships.

**Examples**

To change a role’s password:

```sql
ALTER ROLE admins IDENTIFIED BY xyRP35z;
```

**Using ALTER ROLE to Manage Database Link and DBMS_RLS Privileges**

Advanced Server 9.2 includes extra syntax (not offered by Oracle) for the ALTER ROLE command. This syntax can be useful when assigning privileges related to creating and dropping Oracle-compatible database links, and fine-grained access control (using DBMS_RLS).
CREATE DATABASE LINK

A user who holds the CREATE DATABASE LINK privilege may create a private database link. The following ALTER ROLE command grants privileges to an Advanced Server role that allow the specified role to create a private database link:

```
ALTER ROLE role_name
   WITH [CREATEDBLINK | CREATE DATABASE LINK]
```

This command is the functional equivalent of:

```
GRANT CREATE DATABASE LINK to role_name
```

Use the following command to revoke the privilege:

```
ALTER ROLE role_name
   WITH [NOCREATEDBLINK | NO CREATE DATABASE LINK]
```

Please note: the CREATEDBLINK and NOCREATEDBLINK keywords should be considered deprecated syntax; we recommend using the CREATE DATABASE LINK and NO CREATE DATABASE LINK syntax options.

CREATE PUBLIC DATABASE LINK

A user who holds the CREATE PUBLIC DATABASE LINK privilege may create a public database link. The following ALTER ROLE command grants privileges to an Advanced Server role that allow the specified role to create a public database link:

```
ALTER ROLE role_name
   WITH [CREATEPUBLICDBLINK | CREATE PUBLIC DATABASE LINK]
```

This command is the functional equivalent of:

```
GRANT CREATE PUBLIC DATABASE LINK to role_name
```

Use the following command to revoke the privilege:

```
ALTER ROLE role_name
   WITH [NOCREATEPUBLICDBLINK | NO CREATE PUBLIC DATABASE LINK]
```

Please note: the CREATEPUBLICDBLINK and NOCREATEPUBLICDBLINK keywords should be considered deprecated syntax; we recommend using the CREATE PUBLIC DATABASE LINK and NO CREATE PUBLIC DATABASE LINK syntax options.
**DROP PUBLIC DATABASE LINK**

A user who holds the DROP PUBLIC DATABASE LINK privilege may drop a public database link. The following ALTER ROLE command grants privileges to an Advanced Server role that allow the specified role to drop a public database link:

```
ALTER ROLE role_name
    WITH [DROPPUBLICDBLINK | DROP PUBLIC DATABASE LINK]
```

This command is the functional equivalent of:

```
GRANT DROP PUBLIC DATABASE LINK to role_name
```

Use the following command to revoke the privilege:

```
ALTER ROLE role_name
    WITH [NODROPPUBLICDBLINK | NO DROP PUBLIC DATABASE LINK]
```

Please note: the DROPPUBLICDBLINK and NODROPPUBLICDBLINK keywords should be considered deprecated syntax; we recommend using the DROP PUBLIC DATABASE LINK and NO DROP PUBLIC DATABASE LINK syntax options.

**EXEMPT ACCESS POLICY**

A user who holds the EXEMPT ACCESS POLICY privilege is exempt from fine-grained access control (DBMS_RLS) policies. A user who holds these privileges will be able to view or modify any row in a table constrained by a DBMS_RLS policy. The following ALTER ROLE command grants privileges to an Advanced Server role that exempt the specified role from any defined DBMS_RLS policies:

```
ALTER ROLE role_name
    WITH [POLICYEXEMPT | EXEMPT ACCESS POLICY]
```

This command is the functional equivalent of:

```
GRANT EXEMPT ACCESS POLICY TO role_name
```

Use the following command to revoke the privilege:

```
ALTER ROLE role_name
    WITH [NOPOLICYEXEMPT | NO EXEMPT ACCESS POLICY]
```

Please note: the POLICYEXEMPT and NOPOLICYEXEMPT keywords should be considered deprecated syntax; we recommend using the EXEMPT ACCESS POLICY and NO EXEMPT ACCESS POLICY syntax options.
See Also

CREATE ROLE, DROP ROLE, GRANT, REVOKE, SET ROLE
### 3.3.4 ALTER SEQUENCE

#### Name

ALTER SEQUENCE -- change the definition of a sequence generator

#### Synopsis

```
ALTER SEQUENCE name [ INCREMENT BY increment ]
    [ MINVALUE minvalue ] [ MAXVALUE maxvalue ]
    [ CACHE cache | NOCACHE ] [ CYCLE ]
```

#### Description

ALTER SEQUENCE changes the parameters of an existing sequence generator. Any parameter not specifically set in the ALTER SEQUENCE command retains its prior setting.

#### Parameters

**name**

The name (optionally schema-qualified) of a sequence to be altered.

**increment**

The clause `INCREMENT BY increment` is optional. A positive value will make an ascending sequence, a negative one a descending sequence. If unspecified, the old increment value will be maintained.

**minvalue**

The optional clause `MINVALUE minvalue` determines the minimum value a sequence can generate. If not specified, the current minimum value will be maintained. Note that the keywords, `NO MINVALUE`, may be used to set this behavior back to the defaults of 1 and $-2^{63}-1$ for ascending and descending sequences, respectively, however, this term is not Oracle compatible.

**maxvalue**

The optional clause `MAXVALUE maxvalue` determines the maximum value for the sequence. If not specified, the current maximum value will be maintained. Note that the keywords, `NO MAXVALUE`, may be used to set this behavior back to the defaults of $2^{63}-1$ and -1 for ascending and descending sequences, respectively, however, this term is not Oracle compatible.
The optional clause `CACHE cache` specifies how many sequence numbers are to be preallocated and stored in memory for faster access. The minimum value is 1 (only one value can be generated at a time, i.e., `NOCACHE`). If unspecified, the old cache value will be maintained.

**CYCLE**

The `CYCLE` option allows the sequence to wrap around when the `maxvalue` or `minvalue` has been reached by an ascending or descending sequence respectively. If the limit is reached, the next number generated will be the `minvalue` or `maxvalue`, respectively. If not specified, the old cycle behavior will be maintained. Note that the key words, `NO CYCLE`, may be used to alter the sequence so that it does not recycle, however, this term is not Oracle compatible.

**Notes**

To avoid blocking of concurrent transactions that obtain numbers from the same sequence, `ALTER SEQUENCE` is never rolled back; the changes take effect immediately and are not reversible.

`ALTER SEQUENCE` will not immediately affect `NEXTVAL` results in backends, other than the current one, that have pre-allocated (cached) sequence values. They will use up all cached values prior to noticing the changed sequence parameters. The current backend will be affected immediately.

**Examples**

Change the increment and cache value of sequence, `serial`.

```
ALTER SEQUENCE serial INCREMENT BY 2 CACHE 5;
```

**See Also**

`CREATE SEQUENCE`, `DROP SEQUENCE`
3.3.5 ALTER SESSION

Name

ALTER SESSION -- change a runtime parameter

Synopsis

ALTER SESSION SET name = value

Description

The ALTER SESSION command changes runtime configuration parameters. ALTER SESSION only affects the value used by the current session. Some of these parameters are provided solely for Oracle syntax compatibility and have no effect whatsoever on the runtime behavior of Postgres Plus Advanced Server. Others will alter a corresponding Postgres Plus Advanced Server database server runtime configuration parameter.

Parameters

name

Name of a settable runtime parameter. Available parameters are listed below.

value

New value of parameter.

Configuration Parameters

The following configuration parameters can be modified using the ALTER SESSION command:

NLS_DATE_FORMAT (string)

Sets the display format for date and time values as well as the rules for interpreting ambiguous date input values. Has the same effect as setting the Postgres Plus Advanced Server datestyle runtime configuration parameter.

NLS_LANGUAGE (string)

Sets the language in which messages are displayed. Has the same effect as setting the Postgres Plus Advanced Server lc_messages runtime configuration parameter.
NLS_LENGTH_SEMANTICS (string)

Valid values are BYTE and CHAR. The default is BYTE. This parameter is provided for syntax compatibility only and has no effect in Postgres Plus Advanced Server.

OPTIMIZER_MODE (string)

Sets the default optimization mode for queries. Valid values are ALL_ROWS, CHOOSE, FIRST_ROWS, FIRST_ROWS_10, FIRST_ROWS_100, and FIRST_ROWS_1000. The default is CHOOSE. This parameter is implemented in Postgres Plus Advanced Server. See Section 3.4 for more information about optimizer hints.

QUERY_REWRITE_ENABLED (string)

Valid values are TRUE, FALSE, and FORCE. The default is FALSE. This parameter is provided for syntax compatibility only and has no effect in Postgres Plus Advanced Server.

QUERY_REWRITE_INTEGRITY (string)

Valid values are ENFORCED, TRUSTED, and STALE_TOLERATED. The default is ENFORCED. This parameter is provided for syntax compatibility only and has no effect in Postgres Plus Advanced Server.

Examples

Set the language to U.S. English in UTF-8 encoding. Note that in this example, the value, en_US.UTF-8, is in the format that must be specified for Postgres Plus Advanced Server. This form is not Oracle compatible.

```
ALTER SESSION SET NLS_LANGUAGE = 'en_US.UTF-8';
```

Set the date display format.

```
ALTER SESSION SET NLS_DATE_FORMAT = 'dd/mm/yyyy';
```
3.3.6 ALTER TABLE

Name

ALTER TABLE -- change the definition of a table

Synopsis

ALTER TABLE name
    action [, ...]
ALTER TABLE name
    RENAME COLUMN column TO new_column
ALTER TABLE name
    RENAME TO new_name

where action is one of:

    ADD column type [ column_constraint [ ... ] ]
    DROP COLUMN column
    ADD table_constraint
    DROP CONSTRAINT constraint_name [ CASCADE ]

Description

ALTER TABLE changes the definition of an existing table. There are several subforms:

ADD column type

    This form adds a new column to the table using the same syntax as CREATE TABLE.

DROP COLUMN

    This form drops a column from a table. Indexes and table constraints involving the column will be automatically dropped as well.

ADD table_constraint

    This form adds a new constraint to a table using the same syntax as CREATE TABLE.
DROP CONSTRAINT

This form drops constraints on a table. Currently, constraints on tables are not required to have unique names, so there may be more than one constraint matching the specified name. All matching constraints will be dropped.

RENAME

The RENAME forms change the name of a table (or an index, sequence, or view) or the name of an individual column in a table. There is no effect on the stored data.

You must own the table to use ALTER TABLE.

Parameters

name

The name (possibly schema-qualified) of an existing table to alter.

column

Name of a new or existing column.

new_column

New name for an existing column.

new_name

New name for the table.

type

Data type of the new column.

table_constraint

New table constraint for the table.

constraint_name

Name of an existing constraint to drop.

CASCADE

Automatically drop objects that depend on the dropped constraint.
Notes

When a column is added with **ADD COLUMN**, all existing rows in the table are initialized with the column’s default value (null if no **DEFAULT** clause is specified).

Adding a column with a non-null default will require the entire table to be rewritten. This may take a significant amount of time for a large table; and it will temporarily require double the disk space.

Adding a **CHECK** or **NOT NULL** constraint requires scanning the table to verify that existing rows meet the constraint.

The **DROP COLUMN** form does not physically remove the column, but simply makes it invisible to SQL operations. Subsequent insert and update operations in the table will store a null value for the column. Thus, dropping a column is quick but it will not immediately reduce the on-disk size of your table, as the space occupied by the dropped column is not reclaimed. The space will be reclaimed over time as existing rows are updated.

Changing any part of a system catalog table is not permitted. Refer to **CREATE TABLE** for a further description of valid parameters.

Examples

To add a column of type **VARCHAR2** to a table:

```
ALTER TABLE emp ADD address VARCHAR2(30);
```

To drop a column from a table:

```
ALTER TABLE emp DROP COLUMN address;
```

To rename an existing column:

```
ALTER TABLE emp RENAME COLUMN address TO city;
```

To rename an existing table:

```
ALTER TABLE emp RENAME TO employee;
```

To add a check constraint to a table:

```
ALTER TABLE emp ADD CONSTRAINT sal_chk CHECK (sal > 500);
```

To remove a check constraint from a table:

```
ALTER TABLE emp DROP CONSTRAINT sal_chk;
```
See Also

CREATE TABLE, DROP TABLE
3.3.7 ALTER TABLESPACE

Name

ALTER TABLESPACE -- change the definition of a tablespace

Synopsis

ALTER TABLESPACE name RENAME TO newname

Description

ALTER TABLESPACE changes the definition of a tablespace.

Parameters

name

The name of an existing tablespace.

newname

The new name of the tablespace. The new name cannot begin with pg_, as such names are reserved for system tablespaces.

Examples

Rename tablespace empspace to employee_space:

ALTER TABLESPACE empspace RENAME TO employee_space;

See Also

DROP TABLESPACE
3.3.8 ALTER USER

Name

ALTER USER -- change a database user account

Synopsis

ALTER USER name IDENTIFIED BY password

Description

ALTER USER is used to change the password of a Postgres Plus Advanced Server user account. A database superuser or a user with the CREATEROLE privilege can use this command. Ordinary users can also use this command to change their own password.

Parameters

name

The name of the user whose attributes are to be altered.

password

The new password to be used for this account.

Examples

Change a user password:

ALTER USER john IDENTIFIED BY xyz;

See Also

CREATE USER, DROP USER
3.3.9 CALL

Name

CALL

Synopsis

CALL procedure_name ["[argument_list]"]

Description

Use the CALL statement to invoke a procedure. To use the CALL statement, you must have EXECUTE privileges on the procedure that the CALL statement is invoking.

Parameters

procedure_name

procedure_name is the (optionally schema-qualified) procedure name.

argument_list

argument_list specifies a comma-separated list of arguments required by the procedure. Note that each member of argument_list corresponds to a formal argument expected by the procedure. Each formal argument may be an IN parameter, an OUT parameter, or an INOUT parameter.

Examples

The CALL statement may take one of several forms, depending on the arguments required by the procedure:

CALL update_balance;
CALL update_balance();
CALL update_balance(1,2,3);
3.3.10  COMMENT

Name

COMMENT -- define or change the comment of an object

Synopsis

COMMENT ON
{
    TABLE table_name |
    COLUMN table_name.column_name
} IS 'text'

Description

COMMENT stores a comment about a database object. To modify a comment, issue a new
COMMENT command for the same object. Only one comment string is stored for each
object. To remove a comment, specify the empty string (two consecutive single quotes
with no intervening space) for text. Comments are automatically dropped when the
object is dropped.

Parameters

table_name

The name of the table to be commented. The table name may be schema-
qualified.

table_name.column_name

The name of a column within table_name to be commented. The table name
may be schema-qualified.

text

The new comment.

Notes

There is presently no security mechanism for comments: any user connected to a
database can see all the comments for objects in that database (although only superusers
can change comments for objects that they don’t own). Do not put security-critical
information in comments.
Examples

Attach a comment to the table emp:

```sql
COMMENT ON TABLE emp IS 'Current employee information';
```

Attach a comment to the empno column of the emp table:

```sql
COMMENT ON COLUMN emp.empno IS 'Employee identification number';
```

Remove these comments:

```sql
COMMENT ON TABLE emp IS '';
COMMENT ON COLUMN emp.empno IS '';
```
3.3.11  COMMIT

Name

COMMIT -- commit the current transaction

Synopsis

COMMIT [ WORK ]

Description

COMMIT commits the current transaction. All changes made by the transaction become visible to others and are guaranteed to be durable if a crash occurs.

Parameters

WORK

  Optional key word - has no effect.

Notes

Use ROLLBACK to abort a transaction. Issuing COMMIT when not inside a transaction does no harm.

Examples

To commit the current transaction and make all changes permanent:

  COMMIT;

See Also

ROLLBACK, ROLLBACK TO SAVEPOINT
3.3.12 CREATE DATABASE

Name

CREATE DATABASE -- create a new database

Synopsis

CREATE DATABASE name

Description

CREATE DATABASE creates a new database.

To create a database, you must be a superuser or have the special CREATEDB privilege. Normally, the creator becomes the owner of the new database. Non-superusers with CREATEDB privilege can only create databases owned by them.

The new database will be created by cloning the standard system database template1.

Parameters

name

The name of the database to be created.

Notes

CREATE DATABASE cannot be executed inside a transaction block.

Errors along the line of “could not initialize database directory” are most likely related to insufficient permissions on the data directory, a full disk, or other file system problems.

Examples

To create a new database:

CREATE DATABASE employees;
3.3.13 CREATE [PUBLIC] DATABASE LINK

Name

CREATE [PUBLIC] DATABASE LINK -- create a new database link.

Synopsis

CREATE [ PUBLIC ] DATABASE LINK name
    CONNECT TO { CURRENT_USER |
        username IDENTIFIED BY 'password' }
    USING { libpq 'libpq_connection_string' | [ oci ] 'oracle_connection_string' }

Description

CREATE DATABASE LINK creates a new database link. A database link is an object that allows a reference to a table or view in a remote database within a DELETE, INSERT, SELECT or UPDATE command. A database link is referenced by appending @dblink to the table or view name referenced in the SQL command where dblink is the name of the database link.

Database links can be public or private. A public database link is one that can be used by any user. A private database link can be used only by the database link’s owner. Specification of the PUBLIC option creates a public database link. If omitted, a private database link is created.

When the CREATE DATABASE LINK command is given, the database link name and the given connection attributes are stored in the Postgres Plus Advanced Server system table named pg_catalog.edb_dblink. When using a given database link, the database containing the edb_dblink entry defining this database link is called the local database. The server and database whose connection attributes are defined within the edb_dblink entry is called the remote database.

A SQL command containing a reference to a database link must be issued while connected to the local database. When the SQL command is executed, the appropriate authentication and connection is made to the remote database to access the table or view to which the @dblink reference is appended.
Parameters

PUBLIC

Create a public database link that can be used by any user. If omitted, then the database link is private and can only be used by the database link’s owner.

name

The name of the database link.

username

The username to be used for connecting to the remote database.

CURRENT_USER

Include CURRENT_USER to specify that Advanced Server should use the user mapping associated with the role that is using the link when establishing a connection to the remote server.

password

The password for username.

libpq

Specifies a libpq connection to a remote Postgres Plus Advanced Server database.

libpq_connection_string

Specify the connection information for a libpq connection.

oci

Specifies a connection to a remote Oracle database. This is Advanced Server’s default behavior.

oracle_connection_string

Specify the connection information for an oci connection.
Notes

To create a non-public database link you must have the `CREATE DATABASE LINK` privilege. To create a public database link you must have the `CREATE PUBLIC DATABASE LINK` privilege.

If you are executing a SQL command that references a database link to a remote Oracle database, Advanced Server needs a way to know where the correct Oracle installation resides on disk. You can either set the `LD_LIBRARY_PATH` environment variable (or `PATH` on Windows) to the `lib` directory of the Oracle client installation directory or 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 `LD_LIBRARY_PATH` (or `PATH` on Windows) environment variable.

The `LD_LIBRARY_PATH` (or `PATH` on Windows) environment variable must be set properly each time you start Advanced Server. To set the `oracle_home` configuration parameter in the `postgresql.conf` file, edit the file, adding the following line:

```bash
oracle_home = 'lib_directory'
```

Substitute the name of the directory that contains `libclntsh.so` (on Linux) or `oci.dll` (on Windows) for `lib_directory`.

After setting the `oracle_home` configuration parameter, you must restart the server for the changes to take effect. Restart the server by executing the following command:

```
/etc/init.d/ppas-9.2 restart
```

Examples

**Creating an oci-dblink Database Link**

The following example demonstrates using the `CREATE DATABASE LINK` command to create a database link (named `chicago`) that connects an instance of Advanced Server to an Oracle server via an oci-dblink connection. The connection information tells Advanced Server to log in to Oracle as user `admin`, whose password is `mypassword`. Including the `oci` option tells Advanced Server that this is an oci-dblink connection; the connection string, `''//127.0.0.1/acctg''` specifies the server address and name of the database.

```sql
CREATE DATABASE LINK chicago
    CONNECT TO admin IDENTIFIED BY 'mypassword'
    USING oci '''//127.0.0.1/acctg''';
```

Note: You can specify a hostname in the connection string (in place of an IP address).
Creating a libpq Database Link

The following example demonstrates using the `CREATE DATABASE LINK` command to create a database link (named `boston`) that connects an instance of Advanced Server to a Postgres Server via a libpq connection. The connection information tells Advanced Server to log in to Postgres as user `admin`, whose password is `mypassword`. Including the `libpq` option tells Advanced Server that this is a libpq connection; the connection string, `'host=127.0.0.1 dbname=sales'` specifies the server address and name of the database.

```
CREATE DATABASE LINK boston
CONNECT TO admin IDENTIFIED BY 'mypassword'
USING libpq 'host=127.0.0.1 dbname=sales';
```

Note: You can specify a hostname in the connection string (in place of an IP address).

Using a Database Link

The following examples demonstrate using a database link with Advanced Server to connect to an Oracle database. The examples assume that a copy of the Postgres Plus Advanced Server sample application’s `emp` table has been created in an Oracle database and a second Postgres Plus Advanced Server database cluster with the sample application is accepting connections at port 5443.

Create a public database link named, `oralink`, to an Oracle database named, `xe`, located at 127.0.0.1 on port 1521. Connect to the Oracle database with username, `edb`, and password, `password`.

```
CREATE PUBLIC DATABASE LINK oralink
CONNECT TO edb IDENTIFIED BY 'password'
USING '/127.0.0.1:1521/xe';
```

Issue a `SELECT` command on the `emp` table in the Oracle database using database link, `oralink`.

```
SELECT * FROM emp@oralink;
```

<table>
<thead>
<tr>
<th>empno</th>
<th>ename</th>
<th>job</th>
<th>mgr</th>
<th>hiredate</th>
<th>sal</th>
<th>comm</th>
<th>deptno</th>
</tr>
</thead>
<tbody>
<tr>
<td>7369</td>
<td>SMITH</td>
<td>CLERK</td>
<td>7902</td>
<td>17-DEC-80 00:00:00</td>
<td>800</td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>7499</td>
<td>ALLEN</td>
<td>SALESMAN</td>
<td>7698</td>
<td>20-FEB-81 00:00:00</td>
<td>1600</td>
<td>300</td>
<td>30</td>
</tr>
<tr>
<td>7521</td>
<td>WARD</td>
<td>SALESMAN</td>
<td>7698</td>
<td>22-FEB-81 00:00:00</td>
<td>1250</td>
<td>500</td>
<td>30</td>
</tr>
<tr>
<td>7566</td>
<td>JONES</td>
<td>MANAGER</td>
<td>7839</td>
<td>02-APR-81 00:00:00</td>
<td>2975</td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>7654</td>
<td>MARTIN</td>
<td>SALESMAN</td>
<td>7698</td>
<td>28-SEP-81 00:00:00</td>
<td>1250</td>
<td>1400</td>
<td>30</td>
</tr>
<tr>
<td>7698</td>
<td>BLAKE</td>
<td>MANAGER</td>
<td>7839</td>
<td>01-MAY-81 00:00:00</td>
<td>2850</td>
<td></td>
<td>30</td>
</tr>
<tr>
<td>7782</td>
<td>CLARK</td>
<td>MANAGER</td>
<td>7839</td>
<td>09-JUN-81 00:00:00</td>
<td>2450</td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>7788</td>
<td>SCOTT</td>
<td>ANALYST</td>
<td>7566</td>
<td>19-APR-87 00:00:00</td>
<td>3000</td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>7839</td>
<td>KING</td>
<td>PRESIDENT</td>
<td>7698</td>
<td>17-NOV-81 00:00:00</td>
<td>5000</td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>7844</td>
<td>TURNER</td>
<td>SALESMAN</td>
<td>7698</td>
<td>08-SEP-81 00:00:00</td>
<td>1500</td>
<td>0</td>
<td>30</td>
</tr>
<tr>
<td>7876</td>
<td>ADAMS</td>
<td>CLERK</td>
<td>7782</td>
<td>23-MAY-87 00:00:00</td>
<td>1100</td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>7900</td>
<td>JAMES</td>
<td>CLERK</td>
<td>7698</td>
<td>03-DEC-81 00:00:00</td>
<td>950</td>
<td></td>
<td>30</td>
</tr>
<tr>
<td>7902</td>
<td>FORD</td>
<td>ANALYST</td>
<td>7566</td>
<td>03-DEC-81 00:00:00</td>
<td>3000</td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>7934</td>
<td>MILLER</td>
<td>CLERK</td>
<td>7782</td>
<td>23-JAN-82 00:00:00</td>
<td>1300</td>
<td></td>
<td>10</td>
</tr>
</tbody>
</table>
```
(14 rows)
Create a private database link named, edblink, to a Postgres Plus Advanced Server database named, edb, located on localhost, running on port 5443. Connect to the Postgres Plus Advanced Server database with username, enterprisedb, and password, password.

```
CREATE DATABASE LINK edblink CONNECT TO enterprisedb IDENTIFIED BY 'password'
USING libpq 'host=localhost port=5443 dbname=edb';
```

Display attributes of database links, oralink and edblink, from the local edb_dblink system table:

```
SELECT lnkname, lnkuser, lnkconnstr FROM pg_catalog.edb_dblink;
```

<table>
<thead>
<tr>
<th>lnkname</th>
<th>lnkuser</th>
<th>lnkconnstr</th>
</tr>
</thead>
<tbody>
<tr>
<td>oralink</td>
<td>edb</td>
<td>//127.0.0.1:1521/xe</td>
</tr>
<tr>
<td>edblink</td>
<td>enterprisedb</td>
<td>host=localhost port=5443 dbname=edb</td>
</tr>
</tbody>
</table>

(2 rows)

Perform a join of the emp table from the Oracle database with the dept table from the Postgres Plus Advanced Server database:

```
SELECT d.deptno, d.dname, e.empno, e.ename, e.job, e.sal, e.comm FROM
emp@oralink e, dept@edblink d WHERE e.deptno = d.deptno ORDER BY 1, 3;
```

<table>
<thead>
<tr>
<th>deptno</th>
<th>dname</th>
<th>empno</th>
<th>ename</th>
<th>job</th>
<th>sal</th>
<th>comm</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>ACCOUNTING</td>
<td>7782</td>
<td>CLARK</td>
<td>MANAGER</td>
<td>2450</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>ACCOUNTING</td>
<td>7839</td>
<td>KING</td>
<td>PRESIDENT</td>
<td>5000</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>ACCOUNTING</td>
<td>7934</td>
<td>MILLER</td>
<td>CLERK</td>
<td>1300</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>RESEARCH</td>
<td>7369</td>
<td>SMITH</td>
<td>CLERK</td>
<td>800</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>RESEARCH</td>
<td>7566</td>
<td>JONES</td>
<td>MANAGER</td>
<td>2975</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>RESEARCH</td>
<td>7788</td>
<td>SCOTT</td>
<td>ANALYST</td>
<td>3000</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>RESEARCH</td>
<td>7876</td>
<td>ADAMS</td>
<td>CLERK</td>
<td>1100</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>RESEARCH</td>
<td>7902</td>
<td>FORD</td>
<td>ANALYST</td>
<td>3000</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>SALES</td>
<td>7499</td>
<td>ALLEN</td>
<td>SALESMAN</td>
<td>1600</td>
<td>300</td>
</tr>
<tr>
<td>30</td>
<td>SALES</td>
<td>7521</td>
<td>WARD</td>
<td>SALESMAN</td>
<td>1250</td>
<td>500</td>
</tr>
<tr>
<td>30</td>
<td>SALES</td>
<td>7654</td>
<td>MARTIN</td>
<td>SALESMAN</td>
<td>1250</td>
<td>1400</td>
</tr>
<tr>
<td>30</td>
<td>SALES</td>
<td>7698</td>
<td>BLAKE</td>
<td>MANAGER</td>
<td>2850</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>SALES</td>
<td>7844</td>
<td>TURNER</td>
<td>SALESMAN</td>
<td>1500</td>
<td>0</td>
</tr>
<tr>
<td>30</td>
<td>SALES</td>
<td>7900</td>
<td>JAMES</td>
<td>CLERK</td>
<td>950</td>
<td></td>
</tr>
</tbody>
</table>

(14 rows)

See Also

DROP DATABASE LINK
3.3.14 CREATE DIRECTORY

Name

CREATE DIRECTORY -- create an alias for a file system directory path

Synopsis

CREATE DIRECTORY name AS 'pathname'

Description

The CREATE DIRECTORY command creates an alias for a file system directory pathname. You must be a database superuser to use this command.

When the alias is specified as the appropriate parameter to the programs of the UTL_FILE package, the operating system files are created in, or accessed from the directory corresponding to the given alias. See Section 7.4 for information about the UTL_FILE package.

Parameters

name

The directory alias name.

pathname

The fully-qualified directory path represented by the alias name. The CREATE DIRECTORY command does not create the operating system directory. The physical directory must be created independently using the appropriate operating system commands.

Notes

The operating system user id, enterprisedb, must have the appropriate read and/or write privileges on the directory if the UTL_FILE package is to be used to create and/or read files using the directory.

The directory alias is stored in the pg_catalog.edb_dir system catalog table. Note that edb_dir is not an Oracle compatible table.
Use the `DROP DIRECTORY` command to delete the directory alias. When a directory alias is deleted, the corresponding physical file system directory is not affected. The file system directory must be deleted using the appropriate operating system commands.

In a Linux system, the directory name separator is a forward slash (`/`).

In a Windows system, the directory name separator can be specified as a forward slash (`/`) or two consecutive backslashes (`\`).

**Examples**

Create an alias named `empdir` for directory `/tmp/empdir` on Linux:

```
CREATE DIRECTORY empdir AS '/tmp/empdir';
```

Create an alias named `empdir` for directory `C:\TEMP\EMPDIR` on Windows:

```
CREATE DIRECTORY empdir AS 'C:/TEMP/EMPDIR';
```

View all of the directory aliases:

```
SELECT * FROM pg_catalog.edb_dir;
```

<table>
<thead>
<tr>
<th>dirname</th>
<th>dirpath</th>
</tr>
</thead>
<tbody>
<tr>
<td>empdir</td>
<td>C:/TEMP/EMPDIR</td>
</tr>
</tbody>
</table>

(1 row)

**See Also**

`DROP DIRECTORY`
3.3.15 CREATE FUNCTION

Name

CREATE FUNCTION -- define a new function

Synopsis

CREATE [ OR REPLACE ] FUNCTION name
   [ (argname [ IN | IN OUT | OUT ] argtype [ DEFAULT value ]
      [, ...]) ]
RETURN rettype [DETERMINISTIC]
[ AUTHID { DEFINER | CURRENT_USER } ]
{ IS | AS }
   [ declaration; ] [, ...]
BEGIN
   statement; [...] [ EXCEPTION
   { WHEN exception [ OR exception ] [...] THEN
      statement; [, ...] } [, ...]
   ]
END [ name ]

Description

CREATE FUNCTION defines a new function. CREATE OR REPLACE FUNCTION will
either create a new function, or replace an existing definition.

If a schema name is included, then the function is created in the specified schema.
Otherwise it is created in the current schema. The name of the new function must not
match any existing function with the same argument types in the same schema. However,
functions of different input argument types may share a name (this is called overloading).
(Overloading of functions is a Postgres Plus Advanced Server feature - overloading of
stored functions is not Oracle compatible.)

To update the definition of an existing function, use CREATE OR REPLACE FUNCTION.
It is not possible to change the name or argument types of a function this way (if you
tried, you would actually be creating a new, distinct function). Also, CREATE OR
REPLACE FUNCTION will not let you change the return type of an existing function. To
do that, you must drop and recreate the function.

The user that creates the function becomes the owner of the function.

See Section 4.2.4 for more information about functions.
Parameters

**name**

The name (optionally schema-qualified) of the function to create.

**argname**

The name of an argument. The argument is referenced by this name within the function body.

**IN | IN OUT | OUT**

The argument mode. **IN** declares the argument for input only. This is the default. **IN OUT** allows the argument to receive a value as well as return a value. **OUT** specifies the argument is for output only.

**argtype**

The data type(s) of the function’s arguments. The argument types may be a base data type, a copy of the type of an existing column using `%TYPE`, or a user-defined type such as a nested table or an object type. A length must not be specified for any base type – for example, specify `VARCHAR2`, not `VARCHAR2(10)`.

The type of a column is referenced by writing `tablename.columnname%TYPE`; using this can sometimes help make a function independent from changes to the definition of a table.

**DEFAULT value**

Supplies a default value for an input argument if one is not supplied in the function call. **DEFAULT** may not be specified for arguments with modes **IN OUT** or **OUT**.

**rettype**

The return data type, which may be any of the types listed for **argtype**. As for **argtype**, a length must not be specified for **rettype**.

**DETERMINISTIC**

Include **DETERMINISTIC** to specify that the function will always return the same result when given the same argument values.
DEFINER | CURRENT_USER

Specifies whether the privileges of the function owner (DEFINER) or the privileges of the current user executing the function (CURRENT_USER) are to be used to determine whether or not access is allowed to database objects referenced in the function. DEFINER is the default.

declaration

A variable, type, or REF CURSOR declaration.

statement

An SPL program statement. Note that a DECLARE - BEGIN - END block is considered an SPL statement unto itself. Thus, the function body may contain nested blocks.

exception

An exception condition name such as NO_DATA_FOUND, OTHERS, etc.

Notes

Postgres Plus Advanced Server allows function overloading; that is, the same name can be used for several different functions so long as they have distinct input (IN, IN OUT) argument data types.

Examples

The function emp_comp takes two numbers as input and returns a computed value. The SELECT command illustrates use of the function.

```
CREATE OR REPLACE FUNCTION emp_comp (p_sal NUMBER, p_comm NUMBER) RETURN NUMBER IS BEGIN RETURN (p_sal + NVL(p_comm, 0)) * 24; END;
SELECT ename "Name", sal "Salary", comm "Commission", emp_comp(sal, comm) "Total Compensation" FROM emp;
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Salary</th>
<th>Commission</th>
<th>Total Compensation</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMITH</td>
<td>800.00</td>
<td></td>
<td>19200.00</td>
</tr>
<tr>
<td>ALLEN</td>
<td>1600.00</td>
<td>300.00</td>
<td>45600.00</td>
</tr>
<tr>
<td>WARD</td>
<td>1250.00</td>
<td>500.00</td>
<td>42000.00</td>
</tr>
<tr>
<td>JONES</td>
<td>2975.00</td>
<td></td>
<td>71400.00</td>
</tr>
<tr>
<td>MARTIN</td>
<td>1250.00</td>
<td>1400.00</td>
<td>63600.00</td>
</tr>
</tbody>
</table>
Function `sal_range` returns a count of the number of employees whose salary falls in the specified range. The following anonymous block calls the function a number of times using the arguments’ default values for the first two calls.

```sql
CREATE OR REPLACE FUNCTION sal_range (  
p_sal_min       NUMBER DEFAULT 0,
p_sal_max       NUMBER DEFAULT 10000
) RETURN INTEGER  
IS  
v_count         INTEGER;  
BEGIN  
    SELECT COUNT(*) INTO v_count FROM emp
    WHERE sal BETWEEN p_sal_min AND p_sal_max;
    RETURN v_count;
END;
BEGIN  
    DBMS_OUTPUT.PUT_LINE('Number of employees with a salary: ' || sal_range);
    DBMS_OUTPUT.PUT_LINE('Number of employees with a salary of at least $2000.00: ' || sal_range(2000.00));
    DBMS_OUTPUT.PUT_LINE('Number of employees with a salary between $2000.00 and $3000.00: ' || sal_range(2000.00, 3000.00));
END;
```

Number of employees with a salary: 14
Number of employees with a salary of at least $2000.00: 6
Number of employees with a salary between $2000.00 and $3000.00: 5

**Pragmas**

```sql
PRAGMA RESTRICT_REFERENCE
```

Advanced Server accepts but ignores syntax referencing `PRAGMA RESTRICT_REFERENCE`.

**See Also**

```sql
DROP FUNCTION
```
3.3.16 CREATE INDEX

Name

CREATE INDEX -- define a new index

Synopsis

CREATE [ UNIQUE ] INDEX name ON table
   ( { column | ( expression ) } )
   [ TABLESPACE tablespace ]

Description

CREATE INDEX constructs an index, name, on the specified table. Indexes are primarily used to enhance database performance (though inappropriate use will result in slower performance). Postgres Plus Advanced Server provides the B-tree index method. The B-tree index method is an implementation of Lehman-Yao high-concurrency B-trees.

The key field(s) for the index are specified as column names, or alternatively as expressions written in parentheses. Multiple fields can be specified to create multicolumn indexes.

An index field can be an expression computed from the values of one or more columns of the table row. This feature can be used to obtain fast access to data based on some transformation of the basic data. For example, an index computed on \texttt{UPPER(col)} would allow the clause \texttt{WHERE UPPER(col) = 'JIM'} to use an index.

Indexes are not used for \texttt{IS NULL} clauses by default.

All functions and operators used in an index definition must be "immutable", that is, their results must depend only on their arguments and never on any outside influence (such as the contents of another table or the current time). This restriction ensures that the behavior of the index is well-defined. To use a user-defined function in an index expression remember to mark the function immutable when you create it.

Parameters

UNIQUE

Causes the system to check for duplicate values in the table when the index is created (if data already exist) and each time data is added. Attempts to insert or update data which would result in duplicate entries will generate an error.
**name**

The name of the index to be created. No schema name can be included here; the index is always created in the same schema as its parent table.

**table**

The name (possibly schema-qualified) of the table to be indexed.

**column**

The name of a column in the table.

**expression**

An expression based on one or more columns of the table. The expression usually must be written with surrounding parentheses, as shown in the syntax. However, the parentheses may be omitted if the expression has the form of a function call.

**tablespace**

The tablespace in which to create the index. If not specified, `default_tablespace` is used, or the database’s default tablespace if `default_tablespace` is an empty string.

### Notes

Up to 32 fields may be specified in a multicolumn index.

### Examples

To create a B-tree index on the column, `ename`, in the table, `emp`:

```sql
CREATE INDEX name_idx ON emp (ename);
```

To create the same index as above, but have it reside in the `index_tblspc` tablespace:

```sql
CREATE INDEX name_idx ON emp (ename) TABLESPACE index_tblspc;
```

### See Also

`DROP INDEX`, `ALTER INDEX`
### 3.3.17 CREATE PACKAGE

**Name**

CREATE PACKAGE -- define a new package specification

#### Synopsis

```sql
CREATE [ OR REPLACE ] PACKAGE name 
[ AUTHID { DEFINER | CURRENT_USER } ] 
{ IS | AS } 
[ declaration; ] [, ...] 
[ { PROCEDURE proc_name 
  [ (argname [ IN | IN OUT | OUT ] argtype [ DEFAULT value ] 
  [, ...]) ]; 
| 
FUNCTION func_name 
  [ (argname [ IN | IN OUT | OUT ] argtype [ DEFAULT value ] 
  [, ...]) ] 
  RETURN rettype; 
} [, ...] 
END [ name ]
```

#### Description

CREATE PACKAGE defines a new package specification. CREATE OR REPLACE PACKAGE will either create a new package specification, or replace an existing specification.

If a schema name is included, then the package is created in the specified schema. Otherwise it is created in the current schema. The name of the new package must not match any existing package in the same schema unless the intent is to update the definition of an existing package, in which case use CREATE OR REPLACE PACKAGE.

The user that creates the procedure becomes the owner of the package.

See Section 6 for more information about packages.

#### Parameters

**name**

The name (optionally schema-qualified) of the package to create.
DEFINER | CURRENT_USER

Specifies whether the privileges of the package owner (DEFINER) or the privileges of the current user executing a program in the package (CURRENT_USER) are to be used to determine whether or not access is allowed to database objects referenced in the package. DEFINER is the default.

declaration

A public variable, type, cursor, or REF CURSOR declaration.

proc_name

The name of a public procedure.

argname

The name of an argument.

IN | IN OUT | OUT

The argument mode.

argtype

The data type(s) of the program’s arguments.

DEFAULT value

Default value of an input argument.

func_name

The name of a public function.

rettype

The return data type.

Examples
The package specification, `empinfo`, contains three public components - a public variable, a public procedure, and a public function. See the `CREATE PACKAGE BODY` command for the package body for this example.

```sql
CREATE OR REPLACE PACKAGE empinfo IS
  emp_name        VARCHAR2(10);
  PROCEDURE get_name (p_empno     NUMBER );
  FUNCTION display_counter RETURN INTEGER;
END;
```

See Also

DROP PACKAGE
CREATE PACKAGE BODY

Name

CREATE BODY PACKAGE -- define a new package body

Synopsis

CREATE [ OR REPLACE ] PACKAGE BODY name
{ IS | AS }
[ declaration; ] [, ...]
{ PROCEDURE proc_name
[ (argname [ IN | IN OUT | OUT ] argtype [ DEFAULT value ]
[, ...])] ]
{ IS | AS }
program_body
END [ proc_name ];
|
FUNCTION func_name
[ (argname [ IN | IN OUT | OUT ] argtype [ DEFAULT value ]
[, ...])] ]
RETURN rettype
{ IS | AS }
program_body
END [ func_name ];
}
[, ...]
BEGIN
statement; [, ... ]
END [ name ]

Description

CREATE PACKAGE BODY defines a new package body. CREATE OR REPLACE
PACKAGE BODY will either create a new package body, or replace an existing body.

If a schema name is included, then the package body is created in the specified schema.
Otherwise it is created in the current schema. The name of the new package body must
match an existing package specification in the same schema. The new package body
name must not match any existing package body in the same schema unless the intent is
to update the definition of an existing package body, in which case use CREATE OR
REPLACE PACKAGE BODY.

See Sections 6.1.2 and 6.2.2 for more information on the package body.
Parameters

name

The name (optionally schema-qualified) of the package body to create.

declaration

A private variable, type, cursor, or REF CURSOR declaration.

proc_name

The name of a public or private procedure. If proc_name exists in the package specification with an identical signature, then it is public, otherwise it is private.

argname

The name of an argument.

IN | IN OUT | OUT

The argument mode.

argtype

The data type(s) of the program’s arguments.

DEFAULT value

Default value of an input argument.

program_body

The declarations and SPL statements that comprise the body of the function or procedure.

func_name

The name of a public or private function. If func_name exists in the package specification with an identical signature, then it is public, otherwise it is private.

rettype

The return data type.
An SPL program statement. Statements in the package initialization section are executed once per session the first time the package is referenced.

Examples

The following is the package body for the empinfo package.

```sql
CREATE OR REPLACE PACKAGE BODY empinfo IS
  v_counter       INTEGER;
  PROCEDURE get_name ( 
    p_empno     NUMBER 
  ) IS 
BEGIN
  SELECT ename INTO emp_name FROM emp WHERE empno = p_empno;
  v_counter := v_counter + 1;
END;
FUNCTION display_counter RETURN INTEGER IS 
BEGIN
  RETURN v_counter;
END;
BEGIN 
  v_counter := 0;
  DBMS_OUTPUT.PUT_LINE('Initialized counter');
END;
BEGIN
  empinfo.get_name(7369);
  DBMS_OUTPUT.PUT_LINE('Employee Name    : ' || empinfo.emp_name);
  DBMS_OUTPUT.PUT_LINE('Number of queries: ' || empinfo.display_counter);
END;

Employee Name    : SMITH
Number of queries: 1

BEGIN
  empinfo.get_name(7900);
  DBMS_OUTPUT.PUT_LINE('Employee Name    : ' || empinfo.emp_name);
  DBMS_OUTPUT.PUT_LINE('Number of queries: ' || empinfo.display_counter);
END;

Employee Name    : JAMES
Number of queries: 2
```

See Also

CREATE PACKAGE, DROP PACKAGE
### 3.3.19 CREATE PROCEDURE

#### Name

CREATE PROCEDURE -- define a new stored procedure

#### Synopsis

```
CREATE [ OR REPLACE ] PROCEDURE name
   [ (argname [ IN | IN OUT | OUT ] argtype [ DEFAULT value ]
      [, ...]) ]
 [ AUTHID { DEFINER | CURRENT_USER } ]
{ IS | AS }
   [ declaration; ] [, ...]
BEGIN
   statement; [...] 
[ EXCEPTION
   { WHEN exception [ OR exception ] [...] THEN
      statement; [, ...] } [, ...] ]
] 
END [ name ]
```

#### Description

CREATE PROCEDURE defines a new stored procedure. CREATE OR REPLACE PROCEDURE will either create a new procedure, or replace an existing definition.

If a schema name is included, then the procedure is created in the specified schema. Otherwise it is created in the current schema. The name of the new procedure must not match any existing procedure in the same schema unless the intent is to update the definition of an existing procedure, in which case use CREATE OR REPLACE PROCEDURE.

The user that creates the procedure becomes the owner of the procedure.

See Section 4.2.3 for more information on procedures.

#### Parameters

**name**

The name (optionally schema-qualified) of the procedure to create.
argname

The name of an argument. The argument is referenced by this name within the procedure body.

IN | IN OUT | OUT

The argument mode. IN declares the argument for input only. This is the default. IN OUT allows the argument to receive a value as well as return a value. OUT specifies the argument is for output only.

argtype

The data type(s) of the procedure’s arguments. The argument types may be a base data type, a copy of the type of an existing column using %TYPE, or a user-defined type such as a nested table or an object type. A length must not be specified for any base type - for example, specify VARCHAR2, not VARCHAR2(10).

The type of a column is referenced by writing tablename.columnname%TYPE; using this can sometimes help make a procedure independent from changes to the definition of a table.

DEFAULT value

Supplies a default value for an input argument if one is not supplied in the procedure call. DEFAULT may not be specified for arguments with modes IN OUT or OUT.

DEFINER | CURRENT_USER

Specifies whether the privileges of the procedure owner (DEFINER) or the privileges of the current user executing the procedure (CURRENT_USER) are to be used to determine whether or not access is allowed to database objects referenced in the procedure. DEFINER is the default.

declaration

A variable, type, or REF CURSOR declaration.

statement
An SPL program statement. Note that a `DECLARE - BEGIN - END` block is considered an SPL statement unto itself. Thus, the function body may contain nested blocks.

```
exception

An exception condition name such as `NO_DATA_FOUND`, `OTHERS`, etc.
```

Examples

The following procedure lists the employees in the `emp` table:

```sql
CREATE OR REPLACE PROCEDURE list_emp
IS
  v_empno       NUMBER(4);
  v_ename       VARCHAR2(10);
  CURSOR emp_cur IS
    SELECT empno, ename FROM emp ORDER BY empno;
BEGIN
  OPEN emp_cur;
  DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
  DBMS_OUTPUT.PUT_LINE('-----    ------');
  LOOP
    FETCH emp_cur INTO v_empno, v_ename;
    EXIT WHEN emp_cur%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE(v_empno || '     ' || v_ename);
  END LOOP;
  CLOSE emp_cur;
END;
EXEC list_emp;
EMPNO    ENAME
-----    ------
7369     SMITH
7499     ALLEN
7521     WARD
7566     JONES
7654     MARTIN
7698     BLAKE
7782     CLARK
7788     SCOTT
7839     KING
7844     TURNER
7876     ADAMS
7900     JAMES
7902     FORD
7934     MILLER
```

The following procedure uses `IN OUT` and `OUT` arguments to return an employee’s number, name, and job based upon a search using first, the given employee number, and if that is not found, then using the given name. An anonymous block calls the procedure.

```sql
CREATE OR REPLACE PROCEDURE emp_job ( 
  p_empno       IN OUT emp.empno%TYPE,
  p_ename       IN OUT emp.ename%TYPE,
  p_job         OUT emp.job%TYPE
```
IS
v_empno    emp.empno%TYPE;
v_ename    emp.ename%TYPE;
v_job      emp.job%TYPE;
BEGIN
    SELECT ename, job INTO v_ename, v_job FROM emp WHERE empno = p_empno;
p_ename := v_ename;
p_job := v_job;
    DBMS_OUTPUT.PUT_LINE('Found employee # ' || p_empno);
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        BEGIN
            SELECT empno, job INTO v_empno, v_job FROM emp
                WHERE ename = p_ename;
p_empno := v_empno;
p_job := v_job;
            DBMS_OUTPUT.PUT_LINE('Found employee ' || p_ename);
        EXCEPTION
            WHEN NO_DATA_FOUND THEN
                DBMS_OUTPUT.PUT_LINE('Could not find an employee with ' || 'number, ' || p_empno || ' nor name, ' || p_ename);
p_empno := NULL;
p_ename := NULL;
p_job := NULL;
        END;
    END;
END;
DECLARE
v_empno    emp.empno%TYPE;
v_ename    emp.ename%TYPE;
v_job      emp.job%TYPE;
BEGIN
    v_empno := 0;
v_ename := 'CLARK';
    emp_job(v_empno, v_ename, v_job);
    DBMS_OUTPUT.PUT_LINE('Employee No: ' || v_empno);
    DBMS_OUTPUT.PUT_LINE('Name       : ' || v_ename);
    DBMS_OUTPUT.PUT_LINE('Job        : ' || v_job);
END;

Found employee CLARK
Employee No: 7782
Name       : CLARK
Job        : MANAGER

See Also

DROP PROCEDURE
3.3.20 CREATE ROLE

Name

CREATE ROLE -- define a new database role

Synopsis

CREATE ROLE name [ IDENTIFIED BY password ]

Description

CREATE ROLE adds a new role to a Postgres Plus Advanced Server database cluster. A role is an entity that can own database objects and have database privileges; a role can be considered a “user”, a “group”, or both depending on how it is used. The newly created role does not have the LOGIN attribute, so it cannot be used to start a session. Use the ALTER ROLE command to give the role LOGIN rights. You must have CREATEROLE privilege or be a database superuser to use the CREATE ROLE command.

If the IDENTIFIED BY clause is specified, the CREATE ROLE command also creates a schema owned by, and with the same name as the newly created role.

Note that roles are defined at the database cluster level, and so are valid in all databases in the cluster.

Parameters

name

The name of the new role.

IDENTIFIED BY password

Sets the role’s password. (A password is only of use for roles having the LOGIN attribute, but you can nonetheless define one for roles without it.) If you do not plan to use password authentication you can omit this option.

Notes

Use ALTER ROLE to change the attributes of a role, and DROP ROLE to remove a role. The attributes specified by CREATE ROLE can be modified by later ALTER ROLE commands.

Use GRANT and REVOKE to add and remove members of roles that are being used as groups.
The maximum length limit for role name and password is 63 characters.

Examples

Create a role (and a schema) named, *admins*, with a password:

```
CREATE ROLE admins IDENTIFIED BY Rt498zb;
```

See Also

*ALTER ROLE, DROP ROLE, GRANT, REVOKE, SET ROLE*
3.3.21 CREATE SCHEMA

Name

CREATE SCHEMA -- define a new schema

Synopsis

CREATE SCHEMA AUTHORIZATION username schema_element [ ... ]

Description

This variation of the CREATE SCHEMA command creates a new schema owned by username and populated with one or more objects. The creation of the schema and objects occur within a single transaction so either all objects are created or none of them including the schema. (Oracle compatibility note: In Oracle, no new schema is created – username, and therefore the schema, must pre-exist.)

A schema is essentially a namespace: it contains named objects (tables, views, etc.) whose names may duplicate those of other objects existing in other schemas. Named objects are accessed either by “qualifying” their names with the schema name as a prefix, or by setting a search path that includes the desired schema(s). Unqualified objects are created in the current schema (the one at the front of the search path, which can be determined with the function CURRENT_SCHEMA). (The search path concept and the CURRENT_SCHEMA function are not Oracle compatible.)

CREATE SCHEMA includes subcommands to create objects within the schema. The subcommands are treated essentially the same as separate commands issued after creating the schema. All the created objects will be owned by the specified user.

Parameters

username

The name of the user who will own the new schema. The schema will be named the same as username. Only superusers may create schemas owned by users other than themselves. (Oracle compatibility note: In Postgres Plus Advanced Server the role, username, must already exist, but the schema must not exist. In Oracle, the user (equivalently, the schema) must exist.)

schema_element

An SQL statement defining an object to be created within the schema. CREATE TABLE, CREATE VIEW, and GRANT are accepted as clauses within CREATE
**SCHEMA**. Other kinds of objects may be created in separate commands after the schema is created.

**Notes**

To create a schema, the invoking user must have the `CREATE` privilege for the current database. (Of course, superusers bypass this check.)

In Postgres Plus Advanced Server, there are other forms of the `CREATE SCHEMA` command that are not Oracle compatible.

**Examples**

```sql
CREATE SCHEMA AUTHORIZATION enterprisedb;
CREATE TABLE empjobs (ename VARCHAR2(10), job VARCHAR2(9))
CREATE VIEW managers AS SELECT ename FROM empjobs WHERE job = 'MANAGER'
GRANT SELECT ON managers TO PUBLIC;
```
3.3.22 CREATE SEQUENCE

Name

CREATE SEQUENCE -- define a new sequence generator

Synopsis

CREATE SEQUENCE name [ INCREMENT BY increment ]
[ { NOMINVALUE | MINVALUE minvalue } ]
[ { NOMAXVALUE | MAXVALUE maxvalue } ]
[ START WITH start ] [ CACHE cache | NOCACHE ] [ CYCLE ]

Description

CREATE SEQUENCE creates a new sequence number generator. This involves creating
and initializing a new special single-row table with the name, name. The generator will
be owned by the user issuing the command.

If a schema name is given then the sequence is created in the specified schema, otherwise
it is created in the current schema. The sequence name must be distinct from the name of
any other sequence, table, index, or view in the same schema.

After a sequence is created, use the functions NEXTVAL and CURRVAL to operate on the
sequence. These functions are documented in Section 3.5.9.

Parameters

name

The name (optionally schema-qualified) of the sequence to be created.

increment

The optional clause INCREMENT BY increment specifies the value to add to
the current sequence value to create a new value. A positive value will make an
ascending sequence, a negative one a descending sequence. The default value is 1.

NOMINVALUE | MINVALUE minvalue

The optional clause MINVALUE minvalue determines the minimum value a
sequence can generate. If this clause is not supplied, then defaults will be used.
The defaults are 1 and \(-2^{63}\)-1 for ascending and descending sequences,
respectively. Note that the key words, NOMINVALUE, may be used to set this behavior to the default.

NOMAXVALUE | MAXVALUE maxvalue

The optional clause MAXVALUE maxvalue determines the maximum value for the sequence. If this clause is not supplied, then default values will be used. The defaults are $2^{63}-1$ and -1 for ascending and descending sequences, respectively. Note that the key words, NOMAXVALUE, may be used to set this behavior to the default.

start

The optional clause START WITH start allows the sequence to begin anywhere. The default starting value is minvalue for ascending sequences and maxvalue for descending ones.

cache

The optional clause CACHE cache specifies how many sequence numbers are to be preallocated and stored in memory for faster access. The minimum value is 1 (only one value can be generated at a time, i.e., NOCACHE), and this is also the default.

CYCLE

The CYCLE option allows the sequence to wrap around when the maxvalue or minvalue has been reached by an ascending or descending sequence respectively. If the limit is reached, the next number generated will be the minvalue or maxvalue, respectively.

If CYCLE is omitted (the default), any calls to NEXTVAL after the sequence has reached its maximum value will return an error. Note that the key words, NO CYCLE, may be used to obtain the default behavior, however, this term is not Oracle compatible.

Notes

Sequences are based on big integer arithmetic, so the range cannot exceed the range of an eight-byte integer (-9223372036854775808 to 9223372036854775807). On some older platforms, there may be no compiler support for eight-byte integers, in which case sequences use regular INTEGER arithmetic (range -2147483648 to +2147483647).

Unexpected results may be obtained if a cache setting greater than one is used for a sequence object that will be used concurrently by multiple sessions. Each session will allocate and cache successive sequence values during one access to the sequence object.
and increase the sequence object’s last value accordingly. Then, the next cache-1 uses of NEXTVAL within that session simply return the preallocated values without touching the sequence object. So, any numbers allocated but not used within a session will be lost when that session ends, resulting in "holes" in the sequence.

Furthermore, although multiple sessions are guaranteed to allocate distinct sequence values, the values may be generated out of sequence when all the sessions are considered. For example, with a cache setting of 10, session A might reserve values 1..10 and return NEXTVAL=1, then session B might reserve values 11..20 and return NEXTVAL=11 before session A has generated NEXTVAL=2. Thus, with a cache setting of one it is safe to assume that NEXTVAL values are generated sequentially; with a cache setting greater than one you should only assume that the NEXTVAL values are all distinct, not that they are generated purely sequentially. Also, the last value will reflect the latest value reserved by any session, whether or not it has yet been returned by NEXTVAL.

**Examples**

Create an ascending sequence called `serial`, starting at 101:

```sql
CREATE SEQUENCE serial START WITH 101;
```

Select the next number from this sequence:

```sql
SELECT serial.NEXTVAL FROM DUAL;
```

```
  nextval
  --------
  101
(1 row)
```

Create a sequence called `supplier_seq` with the NOCACHE option:

```sql
CREATE SEQUENCE supplier_seq 
  MINVALUE 1 
  START WITH 1 
  INCREMENT BY 1 
  NOCACHE;
```

Select the next number from this sequence:

```sql
SELECT supplier_seq.NEXTVAL FROM DUAL;
```

```
  nextval
  --------
   1
(1 row)
```

**See Also**

`ALTER SEQUENCE`, `DROP SEQUENCE`
3.3.23 CREATE SYNONYM

Name

CREATE SYNONYM -- define a new synonym

Synopsis

CREATE [OR REPLACE] [PUBLIC] SYNONYM [schema.]syn_name
FOR object_schema.object_name;

Description

CREATE SYNONYM defines a synonym for certain types of database objects. Postgres
Plus Advanced Server supports synonyms for:

- tables
- views
- sequences
- stored procedures
- stored functions
- other synonyms

A synonym is an alternate name that refers to a database object. See Section 2.2.4 for
additional information on synonyms.

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 referenced 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 Oracle-compatible 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;
```

Notes

Access to the object referenced by the synonym is determined by the permissions of the current user of the synonym; the synonym user must have the appropriate permissions on the underlying database object.

Examples

Create a synonym for the emp table in a schema named, enterprisedb:

```
CREATE SYNONYM personnel FOR enterprisedb.emp;
```

See Also

DROP SYNONYM
3.3.24 CREATE TABLE

Name

CREATE TABLE -- define a new table

Synopsis

CREATE [ GLOBAL TEMPORARY ] TABLE table_name (  
   { column_name data_type [ DEFAULT default_expr ]  
   [ column_constraint [ ... ] ] | table_constraint } [, ...]  
)  
[ ON COMMIT { PRESERVE ROWS | DELETE ROWS } ]  
[ TABLESPACE tablespace ]

where column_constraint is:

[ CONSTRAINT constraint_name ]  
{ NOT NULL | NULL |  
UNIQUE [ USING INDEX TABLESPACE tablespace ] |  
PRIMARY KEY [ USING INDEX TABLESPACE tablespace ] |  
CHECK (expression) |  
REFERENCES reftable [ ( refcolumn ) ]  
   [ ON DELETE action ] }  
[ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]

and table_constraint is:

[ CONSTRAINT constraint_name ]  
{ UNIQUE ( column_name [, ...] )  
   [ USING INDEX TABLESPACE tablespace ] |  
PRIMARY KEY ( column_name [, ...] )  
   [ USING INDEX TABLESPACE tablespace ] |  
CHECK ( expression ) |  
FOREIGN KEY ( column_name [, ...] )  
   REFERENCES reftable [ ( refcolumn [, ...] ) ]  
   [ ON DELETE action ] }  
[ DEFERRABLE | NOT DEFERRABLE ]  
[ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]

Description

CREATE TABLE will create a new, initially empty table in the current database. The table will be owned by the user issuing the command.
If a schema name is given (for example, `CREATE TABLE myschema.mytable ...`) then the table is created in the specified schema. Otherwise it is created in the current schema. Temporary tables exist in a special schema, so a schema name may not be given when creating a temporary table. The table name must be distinct from the name of any other table, sequence, index, or view in the same schema.

`CREATE TABLE` also automatically creates a data type that represents the composite type corresponding to one row of the table. Therefore, tables cannot have the same name as any existing data type in the same schema.

A table cannot have more than 1600 columns. (In practice, the effective limit is lower because of tuple-length constraints).

The optional constraint clauses specify constraints (or tests) that new or updated rows must satisfy for an insert or update operation to succeed. A constraint is an SQL object that helps define the set of valid values in the table in various ways.

There are two ways to define constraints: table constraints and column constraints. A column constraint is defined as part of a column definition. A table constraint definition is not tied to a particular column, and it can encompass more than one column. Every column constraint can also be written as a table constraint; a column constraint is only a notational convenience if the constraint only affects one column.

**Parameters**

**GLOBAL TEMPORARY**

If specified, the table is created as a temporary table. Temporary tables are automatically dropped at the end of a session, or optionally at the end of the current transaction (see `ON COMMIT` below). Existing permanent tables with the same name are not visible to the current session while the temporary table exists, unless they are referenced with schema-qualified names. In addition, temporary tables are not visible outside the session in which it was created. (This aspect of global temporary tables is not Oracle compatible.) Any indexes created on a temporary table are automatically temporary as well.

*table_name*

The name (optionally schema-qualified) of the table to be created.

*column_name*

The name of a column to be created in the new table.
**data_type**

The data type of the column. This may include array specifiers. For more information on the data types included with Postgres Plus Advanced Server, refer to Section 3.2.

**DEFAULT default_expr**

The DEFAULT clause assigns a default data value for the column whose column definition it appears within. The value is any variable-free expression (subqueries and cross-references to other columns in the current table are not allowed). The data type of the default expression must match the data type of the column.

The default expression will be used in any insert operation that does not specify a value for the column. If there is no default for a column, then the default is null.

**CONSTRAINT constraint_name**

An optional name for a column or table constraint. If not specified, the system generates a name.

**NOT NULL**

The column is not allowed to contain null values.

**NULL**

The column is allowed to contain null values. This is the default.

This clause is only available for compatibility with non-standard SQL databases. Its use is discouraged in new applications.

**UNIQUE - column constraint**

**UNIQUE (column_name [, ...]) - table constraint**

The UNIQUE constraint specifies that a group of one or more distinct columns of a table may contain only unique values. The behavior of the unique table constraint is the same as that for column constraints, with the additional capability to span multiple columns.

For the purpose of a unique constraint, null values are not considered equal.

Each unique table constraint must name a set of columns that is different from the set of columns named by any other unique or primary key constraint defined for the table. (Otherwise it would just be the same constraint listed twice.)
The primary key constraint specifies that a column or columns of a table may contain only unique (non-duplicate), non-null values. Technically, PRIMARY KEY is merely a combination of UNIQUE and NOT NULL, but identifying a set of columns as primary key also provides metadata about the design of the schema, as a primary key implies that other tables may rely on this set of columns as a unique identifier for rows.

Only one primary key can be specified for a table, whether as a column constraint or a table constraint.

The primary key constraint should name a set of columns that is different from other sets of columns named by any unique constraint defined for the same table.

The CHECK clause specifies an expression producing a Boolean result which new or updated rows must satisfy for an insert or update operation to succeed. Expressions evaluating to TRUE or “unknown” succeed. Should any row of an insert or update operation produce a FALSE result an error exception is raised and the insert or update does not alter the database. A check constraint specified as a column constraint should reference that column’s value only, while an expression appearing in a table constraint may reference multiple columns.

Currently, CHECK expressions cannot contain subqueries nor refer to variables other than columns of the current row.

These clauses specify a foreign key constraint, which requires that a group of one or more columns of the new table must only contain values that match values in the referenced column(s) of some row of the referenced table. If refcolumn is omitted, the primary key of the reftable is used. The referenced columns must be the columns of a unique or primary key constraint in the referenced table.

In addition, when the data in the referenced columns is changed, certain actions are performed on the data in this table’s columns. The ON DELETE clause specifies the action to perform when a referenced row in the referenced table is being deleted. Referential actions cannot be deferred even if the constraint is deferrable. Here are the following possible actions for each clause:
CASCADE

Delete any rows referencing the deleted row, or update the value of the referencing column to the new value of the referenced column, respectively.

SET NULL

Set the referencing column(s) to NULL.

If the referenced column(s) are changed frequently, it may be wise to add an index to the foreign key column so that referential actions associated with the foreign key column can be performed more efficiently.

DEFERRABLE
NOT DEFERRABLE

This controls whether the constraint can be deferred. A constraint that is not deferrable will be checked immediately after every command. Checking of constraints that are deferrable may be postponed until the end of the transaction (using the SET CONSTRAINTS command). NOT DEFERRABLE is the default. Only foreign key constraints currently accept this clause. All other constraint types are not deferrable.

INITIALLY IMMEDIATE
INITIALLY DEFERRED

If a constraint is deferrable, this clause specifies the default time to check the constraint. If the constraint is INITIALLY IMMEDIATE, it is checked after each statement. This is the default. If the constraint is INITIALLY DEFERRED, it is checked only at the end of the transaction. The constraint check time can be altered with the SET CONSTRAINTS command.

ON COMMIT

The behavior of temporary tables at the end of a transaction block can be controlled using ON COMMIT. The two options are:

PRESERVE ROWS

No special action is taken at the ends of transactions. This is the default behavior. (Note that this aspect is not Oracle compatible. The Oracle default is DELETE ROWS.)

DELETE ROWS
All rows in the temporary table will be deleted at the end of each transaction block. Essentially, an automatic TRUNCATE is done at each commit.

**TABLESPACE** `tablespace`

The *tablespace* is the name of the tablespace in which the new table is to be created. If not specified, default tablespace is used, or the database’s default tablespace if `default_tablespace` is an empty string.

**USING INDEX** **TABLESPACE** `tablespace`

This clause allows selection of the tablespace in which the index associated with a UNIQUE or PRIMARY KEY constraint will be created. If not specified, default tablespace is used, or the database’s default tablespace if `default_tablespace` is an empty string.

**Notes**

Postgres Plus Advanced Server automatically creates an index for each unique constraint and primary key constraint to enforce the uniqueness. Thus, it is not necessary to create an explicit index for primary key columns. (See CREATE INDEX for more information.)

**Examples**

Create table `dept` and table `emp`:

```sql
CREATE TABLE dept (  
deptno  NUMBER(2) NOT NULL CONSTRAINT dept_pk PRIMARY KEY,  
dname   VARCHAR2(14),  
loc     VARCHAR2(13)
);  
CREATE TABLE emp (  
empno    NUMBER(4) NOT NULL CONSTRAINT emp_pk PRIMARY KEY,  
ename    VARCHAR2(10),  
job      VARCHAR2(9),  
mgr      NUMBER(4),  
hiredate DATE,  
sal      NUMBER(7,2),  
comm     NUMBER(7,2),  
deptno   NUMBER(2) CONSTRAINT emp_ref_dept_fk  
    REFERENCES dept(deptno)
);  
```

Define a unique table constraint for the table `dept`. Unique table constraints can be defined on one or more columns of the table.

```sql
CREATE TABLE dept (  
deptno  NUMBER(2) NOT NULL CONSTRAINT dept_pk PRIMARY KEY,  
dname   VARCHAR2(14) CONSTRAINT dept_dname_uq UNIQUE,  
loc     VARCHAR2(13)
);  
```
Define a check column constraint:

```
CREATE TABLE emp (  
  empno  NUMBER(4) NOT NULL CONSTRAINT emp_pk PRIMARY KEY,  
  ename  VARCHAR2(10),  
  job    VARCHAR2(9),  
  mgr    NUMBER(4),  
  hiredate DATE,  
  sal    NUMBER(7,2) CONSTRAINT emp_sal_ck CHECK (sal > 0),  
  comm   NUMBER(7,2),  
  deptno NUMBER(2) CONSTRAINT emp_ref_dept_fk  
    REFERENCES dept(deptno)  
);  
```

Define a check table constraint:

```
CREATE TABLE emp (  
  empno  NUMBER(4) NOT NULL CONSTRAINT emp_pk PRIMARY KEY,  
  ename  VARCHAR2(10),  
  job    VARCHAR2(9),  
  mgr    NUMBER(4),  
  hiredate DATE,  
  sal    NUMBER(7,2),  
  comm   NUMBER(7,2),  
  deptno NUMBER(2) CONSTRAINT emp_ref_dept_fk  
    REFERENCES dept(deptno),  
  CONSTRAINT new_emp_ck CHECK (ename IS NOT NULL AND empno > 7000)  
);  
```

Define a primary key table constraint for the table jobhist. Primary key table constraints can be defined on one or more columns of the table.

```
CREATE TABLE jobhist (  
  empno  NUMBER(4) NOT NULL,  
  startdate  DATE NOT NULL,  
  enddate   DATE,  
  job      VARCHAR2(9),  
  sal      NUMBER(7,2),  
  comm     NUMBER(7,2),  
  deptno   NUMBER(2),  
  chgdesc  VARCHAR2(80),  
  CONSTRAINT jobhist_pk PRIMARY KEY (empno, startdate)  
);  
```

This assigns a literal constant default value for the column, job and makes the default value of hiredate be the date at which the row is inserted.

```
CREATE TABLE emp (  
  empno  NUMBER(4) NOT NULL CONSTRAINT emp_pk PRIMARY KEY,  
  ename  VARCHAR2(10),  
  job    VARCHAR2(9) DEFAULT 'SALESMAN',  
  mgr    NUMBER(4),  
  hiredate DATE DEFAULT SYSDATE,  
  sal    NUMBER(7,2),  
  comm   NUMBER(7,2),  
  deptno NUMBER(2) CONSTRAINT emp_ref_dept_fk  
);  
```
Create table `dept` in `tablespace` `diskvol1`:

```sql
CREATE TABLE dept (
  deptno    NUMBER(2) NOT NULL CONSTRAINT dept_pk PRIMARY KEY,
  dname     VARCHAR2(14),
  loc       VARCHAR2(13)
) TABLESPACE diskvol1;
```

See Also

`ALTER TABLE`, `DROP TABLE`
3.3.25 CREATE TABLE AS

Name

CREATE TABLE AS -- define a new table from the results of a query

Synopsis

CREATE [ GLOBAL TEMPORARY ] TABLE table_name
   [ (column_name [, ...] ) ]
   [ ON COMMIT { PRESERVE ROWS | DELETE ROWS } ]
   [ TABLESPACE tablespace ]
AS query

Description

CREATE TABLE AS creates a table and fills it with data computed by a SELECT command. The table columns have the names and data types associated with the output columns of the SELECT (except that you can override the column names by giving an explicit list of new column names).

CREATE TABLE AS bears some resemblance to creating a view, but it is really quite different: it creates a new table and evaluates the query just once to fill the new table initially. The new table will not track subsequent changes to the source tables of the query. In contrast, a view re-evaluates its defining SELECT statement whenever it is queried.

Parameters

GLOBAL TEMPORARY

If specified, the table is created as a temporary table. Refer to CREATE TABLE for details.

table_name

The name (optionally schema-qualified) of the table to be created.

column_name

The name of a column in the new table. If column names are not provided, they are taken from the output column names of the query.
A query statement (a **SELECT** command). Refer to **SELECT** for a description of the allowed syntax.
### 3.3.26 CREATE TRIGGER

**Name**

CREATE TRIGGER -- define a new trigger

**Synopsis**

CREATE [ OR REPLACE ] TRIGGER name
{ BEFORE | AFTER | INSTEAD OF}
{ INSERT | UPDATE | DELETE }
{ OR { INSERT | UPDATE | DELETE } } [, ...]
ON table
[ FOR EACH ROW ]
[ WHEN condition ]
[ DECLARE
    declaration; [, ...] ]
BEGIN
    statement; [, ...]
[ EXCEPTION
    { WHEN exception [ OR exception ] [...] THEN
        statement; [, ...] } [, ...]
]
    END

**Description**

CREATE TRIGGER defines a new trigger. CREATE OR REPLACE TRIGGER will either create a new trigger, or replace an existing definition.

The name of the new trigger must not match any existing trigger defined on the same table unless the intent is to update the definition of an existing trigger, in which case use CREATE OR REPLACE TRIGGER.

The trigger is created in the same schema as the table on which the triggering event is defined.

See Section 5 for more information about triggers.

**Parameters**

*name*

The name of the trigger to create.
BEFORE | AFTER

Determines whether the trigger is fired before or after the triggering event.

INSERT | UPDATE | DELETE

Defines the triggering event.

table

The name of the table on which the triggering event occurs.

condition

*condition* is a Boolean expression that determines if the trigger will actually be executed; if *condition* evaluates to TRUE, the trigger will fire.

If the trigger definition includes the **FOR EACH ROW** keywords, the **WHEN** clause can refer to columns of the old and/or new row values by writing **OLD.column_name** or **NEW.column_name** respectively. **INSERT** triggers cannot refer to **OLD** and **DELETE** triggers cannot refer to **NEW**.

If the trigger includes the **INSTEAD OF** keywords, it may not include a **WHEN** clause.

**WHEN** clauses cannot contain subqueries.

**FOR EACH ROW**

Determines whether the trigger should be fired once for every row affected by the triggering event, or just once per SQL statement. If specified, the trigger is fired once for every affected row (row-level trigger), otherwise the trigger is a statement-level trigger.

declaration

A variable, type, or **REF CURSOR** declaration.

statement

An SPL program statement. Note that a **DECLARE - BEGIN - END** block is considered an SPL statement unto itself. Thus, the trigger body may contain nested blocks.
exception

An exception condition name such as NO_DATA_FOUND, OTHERS, etc.

Examples

The following is a statement-level trigger that fires after the triggering statement (insert, update, or delete on table `emp`) is executed.

```sql
CREATE OR REPLACE TRIGGER user_audit_trig
AFTER INSERT OR UPDATE OR DELETE ON emp
DECLARE
v_action VARCHAR2(24);
BEGIN
    IF INSERTING THEN
        v_action := ' added employee(s) on ';
    ELSIF UPDATING THEN
        v_action := ' updated employee(s) on ';
    ELSIF DELETING THEN
        v_action := ' deleted employee(s) on ';
    END IF;
    DBMS_OUTPUT.PUT_LINE('User ' || USER || v_action ||
            TO_CHAR(SYSDATE,'YYYY-MM-DD'));
END;
```

The following is a row-level trigger that fires before each row is either inserted, updated, or deleted on table `emp`.

```sql
CREATE OR REPLACE TRIGGER emp_sal_trig
BEFORE DELETE OR INSERT OR UPDATE ON emp
FOR EACH ROW
DECLARE
    sal_diff NUMBER;
BEGIN
    IF INSERTING THEN
        DBMS_OUTPUT.PUT_LINE('Inserting employee ' || :NEW.empno);
        DBMS_OUTPUT.PUT_LINE('..New salary: ' || :NEW.sal);
    END IF;
    IF UPDATING THEN
        sal_diff := :NEW.sal - :OLD.sal;
        DBMS_OUTPUT.PUT_LINE('Updating employee ' || :OLD.empno);
        DBMS_OUTPUT.PUT_LINE('..Old salary: ' || :OLD.sal);
        DBMS_OUTPUT.PUT_LINE('..New salary: ' || :NEW.sal);
        DBMS_OUTPUT.PUT_LINE('..Raise     : ' || sal_diff);
    END IF;
    IF DELETING THEN
        DBMS_OUTPUT.PUT_LINE('Deleting employee ' || :OLD.empno);
        DBMS_OUTPUT.PUT_LINE('..Old salary: ' || :OLD.sal);
    END IF;
END;
```

See Also

DROP TRIGGER
3.3.27 CREATE TYPE

Name

CREATE TYPE -- define a new user-defined type

Synopsis

CREATE [ OR REPLACE ] TYPE name
    [ AUTHID { DEFINER | CURRENT_USER } ]
    { IS | AS } OBJECT
    ( { attribute { datatype | objtype | collecttype } }
      [, ...]
      [ method_spec ] [, ...]
    ) [ [ NOT ] { FINAL | INSTANTIABLE } ] ... 

where method_spec is:

    [ [ NOT ] { FINAL | INSTANTIABLE } ] ... 
    [ OVERRIDING ]
    subprogram_spec

and subprogram_spec is:

    { MEMBER | STATIC }
    { PROCEDURE proc_name
      [ ( [ SELF [ IN | IN OUT ] name ]
        [ , argname [ IN | IN OUT | OUT ] argtype
          [ DEFAULT value ]
        ) ...)
      ]
      | 
      FUNCTION func_name
      [ ( [ SELF [ IN | IN OUT ] name ]
        [ , argname [ IN | IN OUT | OUT ] argtype
          [ DEFAULT value ]
        ) ...)
      ]
      RETURN rettype
    }

CREATE [ OR REPLACE ] TYPE name { IS | AS } TABLE OF
    { datatype | objtype | collecttype }

CREATE [ OR REPLACE ] TYPE name { IS | AS }
    { VARRAY | VARYING ARRAY } (maxsize) OF { datatype | objtype }
Description

CREATE TYPE defines a new user-defined data type. The types that can be created are an object type, a nested table type, or a varray type. (Nested table and varray types belong to the category of types known as collections. See Section 4.10 for information on collections.) CREATE OR REPLACE TYPE will either create a new type definition, or replace an existing type definition.

If a schema name is included, then the type is created in the specified schema, otherwise it is created in the current schema. The name of the new type must not match any existing type in the same schema unless the intent is to update the definition of an existing type, in which case use CREATE OR REPLACE TYPE.

Note: The OR REPLACE option cannot be currently used to add, delete, or modify the attributes of an existing object type. Use the DROP TYPE command to first delete the existing object type. The OR REPLACE option can be used to add, delete, or modify the methods in an existing object type.

Note: The PostgreSQL form of the ALTER TYPE ALTER ATTRIBUTE command can be used to change the data type of an attribute in an existing object type. However, the ALTER TYPE command cannot add or delete attributes in the object type.

The user that creates the type becomes the owner of the type.

See Section 4.10.2 for more information on nested table types. See Section 4.10.3 for more information on varray types. See Section 8 for more information on object types.

Parameters

name

The name (optionally schema-qualified) of the type to create.

DEFINER | CURRENT_USER

Specifies whether the privileges of the object type owner (DEFINER) or the privileges of the current user executing a method in the object type (CURRENT_USER) are to be used to determine whether or not access is allowed to database objects referenced in the object type. DEFINER is the default.

attribute

The name of an attribute in the object type.

datatype
The data type that defines an attribute of the object type or the elements of the collection type that is being created.

**objtype**

The name of an object type that defines an attribute of the object type or the elements of the collection type that is being created.

**collecttype**

The name of a collection type that defines an attribute of the object type or the elements of the collection type that is being created.

**FINAL**
**NOT FINAL**

For an object type, specifies whether or not a subtype can be derived from the object type. **FINAL** (subtype cannot be derived from the object type) is the default.

For **method_spec**, specifies whether or not the method may be overridden in a subtype. **NOT FINAL** (method may be overridden in a subtype) is the default.

**INSTANTIABLE**
**NOT INSTANTIABLE**

For an object type, specifies whether or not an object instance can be created of this object type. **INSTANTIABLE** (an instance of this object type can be created) is the default. If **NOT INSTANTIABLE** is specified, then **NOT FINAL** must be specified as well. If **method_spec** for any method in the object type contains the **NOT INSTANTIABLE** qualifier, then the object type, itself, must be defined with **NOT INSTANTIABLE** and **NOT FINAL** following the closing parenthesis of the object type specification.

For **method_spec**, specifies whether or not the object type definition provides an implementation for the method. **INSTANTIABLE** (the **CREATE TYPE BODY** command for the object type provides the implementation of the method) is the default. If **NOT INSTANTIABLE** is specified, then the **CREATE TYPE BODY** command for the object type must not contain the implementation of the method.

**OVERRIDING**

If **OVERRIDING** is specified, **method_spec** overrides an identically named method with the same number of identically named method arguments with the
same data types, in the same order, and the same return type (if the method is a function) as defined in a supertype.

MEMBER
STATIC

Specify MEMBER if the subprogram operates on an object instance. Specify STATIC if the subprogram operates independently of any particular object instance.

proc_name

The name of the procedure to create.

SELF [ IN | IN OUT ] name

For a member method there is an implicit, built-in parameter named SELF whose data type is that of the object type being defined. SELF refers to the object instance that is currently invoking the method. SELF can be explicitly declared as an IN or IN OUT parameter in the parameter list. If explicitly declared, SELF must be the first parameter in the parameter list. If SELF is not explicitly declared, its parameter mode defaults to IN OUT for member procedures and IN for member functions.

argname

The name of an argument. The argument is referenced by this name within the method body.

argtype

The data type(s) of the method’s arguments. The argument types may be a base data type or a user-defined type such as a nested table or an object type. A length must not be specified for any base type - for example, specify VARCHAR2, not VARCHAR2(10).

DEFAULT value

Supplies a default value for an input argument if one is not supplied in the method call. DEFAULT may not be specified for arguments with modes IN OUT or OUT.

func_name

The name of the function to create.
rettype

The return data type, which may be any of the types listed for argtype. As for argtype, a length must not be specified for rettype.

maxsize

The maximum number of elements permitted in the varray.

Examples

Create object type addr_obj_typ.

```sql
CREATE OR REPLACE TYPE addr_obj_typ AS OBJECT ( 
  street          VARCHAR2(30),
  city            VARCHAR2(20),
  state           CHAR(2),
  zip             NUMBER(5)
);
```

Create object type emp_obj_typ that includes a member method display_emp.

```sql
CREATE OR REPLACE TYPE emp_obj_typ AS OBJECT ( 
  empno           NUMBER(4),
  ename           VARCHAR2(20),
  addr            ADDR_OBJ_TYP,
  MEMBER PROCEDURE display_emp (SELF IN OUT emp_obj_typ)
);
```

Create object type dept_obj_typ that includes a static method get_dname.

```sql
CREATE OR REPLACE TYPE dept_obj_typ AS OBJECT ( 
  deptno          NUMBER(2),
  STATIC FUNCTION get_dname (p_deptno IN NUMBER) RETURN VARCHAR2,
  MEMBER PROCEDURE display_dept
);
```

Create a nested table type, budget_tbl_typ, of data type, NUMBER(8,2).

```sql
CREATE OR REPLACE TYPE budget_tbl_typ IS TABLE OF NUMBER(8,2);
```

See Also

CREATE TYPE BODY, DROP TYPE
3.3.28 CREATE TYPE BODY

Name

CREATE TYPE BODY -- define a new object type body

Synopsis

CREATE [ OR REPLACE ] TYPE BODY name
   { IS | AS }
   method_spec [...] END

where method_spec is:

   subprogram_spec

and subprogram_spec is:

   { MEMBER | STATIC }
   { PROCEDURE proc_name
     [ ( [ SELF [ IN | IN OUT ] name ]
     [ , argname [ IN | IN OUT | OUT ] argtype
     [ DEFAULT value ]
     ] ...) ]
   { IS | AS }
   program_body
   END;
   |
   FUNCTION func_name
     [ ( [ SELF [ IN | IN OUT ] name ]
     [ , argname [ IN | IN OUT | OUT ] argtype
     [ DEFAULT value ]
     ] ...) ]
     RETURN rettype
   { IS |AS }
   program_body
   END;
}

Description

CREATE TYPE BODY defines a new object type body. CREATE OR REPLACE TYPE
BODY will either create a new object type body, or replace an existing body.
If a schema name is included, then the object type body is created in the specified schema. Otherwise it is created in the current schema. The name of the new object type body must match an existing object type specification in the same schema. The new object type body name must not match any existing object type body in the same schema unless the intent is to update the definition of an existing object type body, in which case use CREATE OR REPLACE TYPE BODY.

See Section 8.2.2 for more information on the object type body.

**Parameters**

*name*

The name (optionally schema-qualified) of the object type for which a body is to be created.

*MEMBER*

*STATIC*

Specify MEMBER if the subprogram operates on an object instance. Specify STATIC if the subprogram operates independently of any particular object instance.

*proc_name*

The name of the procedure to create.

*SELF [ IN | IN OUT ] name*

For a member method there is an implicit, built-in parameter named SELF whose data type is that of the object type being defined. SELF refers to the object instance that is currently invoking the method. SELF can be explicitly declared as an IN or IN OUT parameter in the parameter list. If explicitly declared, SELF must be the first parameter in the parameter list. If SELF is not explicitly declared, its parameter mode defaults to IN OUT for member procedures and IN for member functions.

*argname*

The name of an argument. The argument is referenced by this name within the method body.

*argtype*

The data type(s) of the method’s arguments. The argument types may be a base data type or a user-defined type such as a nested table or an object type. A length
must not be specified for any base type - for example, specify VARCHAR2, not VARCHAR2(10).

**DEFAULT value**

Supplies a default value for an input argument if one is not supplied in the method call. DEFAULT may not be specified for arguments with modes IN OUT or OUT.

**program_body**

The declarations and SPL statements that comprise the body of the function or procedure.

**func_name**

The name of the function to create.

**rettype**

The return data type, which may be any of the types listed for argtype. As for argtype, a length must not be specified for rettype.

**Examples**

Create the object type body for object type `emp_obj_typ` given in the example for the CREATE TYPE command.

```sql
CREATE OR REPLACE TYPE BODY emp_obj_typ AS
  MEMBER PROCEDURE display_emp (SELF IN OUT emp_obj_typ)
  IS
    BEGIN
      DBMS_OUTPUT.PUT_LINE('Employee No : ' || empno);
      DBMS_OUTPUT.PUT_LINE('Name : ' || ename);
      DBMS_OUTPUT.PUT_LINE('Street : ' || addr.street);
      DBMS_OUTPUT.PUT_LINE('City/State/Zip: ' || addr.city || ', ' || addr.state || ' ' || LPAD(addr.zip,5,'0'));
    END;
END;
```

Create the object type body for object type `dept_obj_typ` given in the example for the CREATE TYPE command.

```sql
CREATE OR REPLACE TYPE BODY dept_obj_typ AS
  STATIC FUNCTION get_dname (p_deptno IN NUMBER) RETURN VARCHAR2
  IS
    v_dname VARCHAR2(14);
    BEGIN
      CASE p_deptno
        WHEN 10 THEN v_dname := 'ACCOUNTING';
        WHEN 20 THEN v_dname := 'RESEARCH';
        WHEN 30 THEN v_dname := 'SALES';
        WHEN 40 THEN v_dname := 'OPERATIONS';
      END CASE;
      RETURN v_dname;
    END;
END;
```
ELSE v_dname := 'UNKNOWN';
END CASE;
RETURN v_dname;
END;
MEMBER PROCEDURE display_dept
IS
BEGIN
  DBMS_OUTPUT.PUT_LINE('Dept No    : ' || SELF.deptno);
  DBMS_OUTPUT.PUT_LINE('Dept Name  : ' ||
    dept_obj_typ.get_dname(SELF.deptno));
END;
END;

See Also

CREATE TYPE, DROP TYPE
3.3.29 CREATE USER

Name

CREATE USER -- define a new database user account

Synopsis

CREATE USER name IDENTIFIED BY password

Description

CREATE USER adds a new user to a Postgres Plus Advanced Server database cluster. You must be a database superuser to use this command.

When the CREATE USER command is given, a schema will also be created with the same name as the new user and owned by the new user. Objects with unqualified names created by this user will be created in this schema.

Parameters

name

The name of the user.

password

Sets the user’s password. The password can be changed later using ALTER USER.

Notes

The maximum length allowed for the user name and password is 63 characters.

Examples

Create a user named, john.

CREATE USER john IDENTIFIED BY abc;

See Also

DROP USER
3.3.30 CREATE VIEW

Name

CREATE VIEW -- define a new view

Synopsis

CREATE [ OR REPLACE ] VIEW name [ ( column_name [, ...] ) ]
   AS query

Description

CREATE VIEW defines a view of a query. The view is not physically materialized. Instead, the query is run every time the view is referenced in a query.

CREATE OR REPLACE VIEW is similar, but if a view of the same name already exists, it is replaced.

If a schema name is given (for example, CREATE VIEW myschema.myview ...) then the view is created in the specified schema. Otherwise it is created in the current schema. The view name must be distinct from the name of any other view, table, sequence, or index in the same schema.

Parameters

name

The name (optionally schema-qualified) of a view to be created.

column_name

An optional list of names to be used for columns of the view. If not given, the column names are deduced from the query.

query

A query (that is, a SELECT statement) which will provide the columns and rows of the view.

Refer to SELECT for more information about valid queries.
Notes

Currently, views are read only - the system will not allow an insert, update, or delete on a view. You can get the effect of an updatable view by creating rules that rewrite inserts, etc. on the view into appropriate actions on other tables. See the CREATE RULE command in the Postgres Plus documentation set.

Access to tables referenced in the view is determined by permissions of the view owner. However, functions called in the view are treated the same as if they had been called directly from the query using the view. Therefore the user of a view must have permissions to call all functions used by the view.

Examples

Create a view consisting of all employees in department 30:

```
CREATE VIEW dept_30 AS SELECT * FROM emp WHERE deptno = 30;
```

See Also

DROP VIEW
3.3.31  **DELETE**

**Name**

DELETE -- delete rows of a table

**Synopsis**

DELETE [ optimizer_hint ] FROM table[@dblink ]
   [ WHERE condition ]
   [ RETURNING return_expression [, ...]
      { INTO { record | variable [, ...] }
        | BULK COLLECT INTO collection [, ...] } ]

**Description**

DELETE deletes rows that satisfy the WHERE clause from the specified table. If the WHERE clause is absent, the effect is to delete all rows in the table. The result is a valid, but empty table.

**Note:** The TRUNCATE command provides a faster mechanism to remove all rows from a table.

The RETURNING INTO { record | variable [, ...] } clause may only be specified if the DELETE command is used within an SPL program. In addition the result set of the DELETE command must not include more than one row, otherwise an exception is thrown. If the result set is empty, then the contents of the target record or variables are set to null.

The RETURNING BULK COLLECT INTO collection [, ...] clause may only be specified if the DELETE command is used within an SPL program. If more than one collection is specified as the target of the BULK COLLECT INTO clause, then each collection must consist of a single, scalar field – i.e., collection must not be a record. The result set of the DELETE command may contain none, one, or more rows. return_expression evaluated for each row of the result set, becomes an element in collection starting with the first element. Any existing rows in collection are deleted. If the result set is empty, then collection will be empty.

You must have the DELETE privilege on the table to delete from it, as well as the SELECT privilege for any table whose values are read in the condition.
Parameters

optimizer_hint

Comment-embedded hints to the optimizer for selection of an execution plan. See Section 3.4 for information on optimizer hints.

table

The name (optionally schema-qualified) of an existing table.

dblink

Database link name identifying a remote database. See the CREATE DATABASE LINK command for information on database links.

condition

A value expression that returns a value of type BOOLEAN that determines the rows which are to be deleted.

return_expression

An expression that may include one or more columns from table. If a column name from table is specified in return_expression, the value substituted for the column when return_expression is evaluated is the value from the deleted row.

record

A record whose field the evaluated return_expression is to be assigned. The first return_expression is assigned to the first field in record, the second return_expression is assigned to the second field in record, etc. The number of fields in record must exactly match the number of expressions and the fields must be type-compatible with their assigned expressions.

variable

A variable to which the evaluated return_expression is to be assigned. If more than one return_expression and variable are specified, the first return_expression is assigned to the first variable, the second return_expression is assigned to the second variable, etc. The number of variables specified following the INTO keyword must exactly match the number of expressions following the RETURNING keyword and the variables must be type-compatible with their assigned expressions.
A collection in which an element is created from the evaluated return_expression. There can be either a single collection which may be a collection of a single field or a collection of a record type, or there may be more than one collection in which case each collection must consist of a single field. The number of return expressions must match in number and order the number of fields in all specified collections. Each corresponding return_expression and collection field must be type-compatible.

Examples

Delete all rows for employee 7900 from the jobhist table:

```
DELETE FROM jobhist WHERE empno = 7900;
```

Clear the table jobhist:

```
DELETE FROM jobhist;
```

See Also

TRUNCATE
3.3.32 DROP DATABASE LINK

Name

DROP DATABASE LINK -- remove a database link

Synopsis

DROP [ PUBLIC ] DATABASE LINK name

Description

DROP DATABASE LINK drops existing database links. To execute this command you must be a superuser or the owner of the database link.

Parameters

name

The name of a database link to be removed.

PUBLIC

Indicates that name is a public database link.

Examples

Remove the public database link named, oralink:

    DROP PUBLIC DATABASE LINK oralink;

Remove the private database link named, edblink:

    DROP DATABASE LINK edblink;

See Also

CREATE DATABASE LINK
### 3.3.33 DROP DIRECTORY

**Name**

DROP DIRECTORY -- remove a directory alias for a file system directory path

**Synopsis**

DROP DIRECTORY *name*

**Description**

DROP DIRECTORY drops an existing alias for a file system directory path that was created with the CREATE DIRECTORY command. To execute this command you must be a superuser.

When a directory alias is deleted, the corresponding physical file system directory is not affected. The file system directory must be deleted using the appropriate operating system commands.

**Parameters**

*name*

The name of a directory alias to be removed.

**Examples**

Remove the directory alias named *empdir*:

```
DROP DIRECTORY empdir;
```

**See Also**

CREATE DIRECTORY
3.3.34 DROP FUNCTION

Name

DROP FUNCTION -- remove a function

Synopsis

DROP FUNCTION name
    [ ( [ argmode ] [ argname ] argtype ] [, ...]) ]

Description

DROP FUNCTION removes the definition of an existing function. To execute this command you must be a superuser or the owner of the function. All input (IN, IN OUT) argument data types to the function must be specified if there is at least one input argument. (This requirement is not Oracle compatible. In Oracle, only the function name is specified. Postgres Plus Advanced Server allows overloading of function names, so the function signature given by the input argument data types is required in the Postgres Plus Advanced Server DROP FUNCTION command.)

Parameters

name

The name (optionally schema-qualified) of an existing function.

argmode

The mode of an argument: IN, IN OUT, or OUT. If omitted, the default is IN. Note that DROP FUNCTION does not actually pay any attention to OUT arguments, since only the input arguments are needed to determine the function’s identity. So it is sufficient to list only the IN and IN OUT arguments. (Specification of argmode is not Oracle compatible and applies only to Postgres Plus Advanced Server.)

argname

The name of an argument. Note that DROP FUNCTION does not actually pay any attention to argument names, since only the argument data types are needed to determine the function’s identity. (Specification of argname is not Oracle compatible and applies only to Postgres Plus Advanced Server.)
The data type of an argument of the function. (Specification of argtype is not Oracle compatible and applies only to Postgres Plus Advanced Server.)

Examples

The following command removes the emp_comp function.

```
DROP FUNCTION emp_comp(NUMBER, NUMBER);
```

See Also

CREATE FUNCTION
3.3.35 DROP INDEX

Name

DROP INDEX -- remove an index

Synopsis

DROP INDEX name

Description

DROP INDEX drops an existing index from the database system. To execute this command you must be a superuser or the owner of the index. If any objects depend on the index, an error will be given and the index will not be dropped.

Parameters

name

The name (optionally schema-qualified) of an index to remove.

Examples

This command will remove the index, name_idx:

```
DROP INDEX name_idx;
```

See Also

ALTER INDEX, CREATE INDEX
### 3.3.36 DROP PACKAGE

**Name**

`DROP PACKAGE` -- remove a package

**Synopsis**

```
DROP PACKAGE [ BODY ] name
```

**Description**

`DROP PACKAGE` drops an existing package. To execute this command you must be a superuser or the owner of the package. If `BODY` is specified, only the package body is removed – the package specification is not dropped. If `BODY` is omitted, both the package specification and body are removed.

**Parameters**

`name`

The name (optionally schema-qualified) of a package to remove.

**Examples**

This command will remove the `emp_admin` package:

```
DROP PACKAGE emp_admin;
```

**See Also**

`CREATE PACKAGE`, `CREATE PACKAGE BODY`
3.3.37 DROP PROCEDURE

Name

DROP PROCEDURE -- remove a procedure

Synopsis

DROP PROCEDURE name

Description

DROP PROCEDURE removes the definition of an existing procedure. To execute this command you must be a superuser or the owner of the procedure.

Parameters

name

The name (optionally schema-qualified) of an existing procedure.

Examples

The following command removes the select_emp procedure.

DROP PROCEDURE select_emp;

See Also

CREATE PROCEDURE
3.3.38 DROP SYNONYM

Name

DROP SYNONYM -- remove a synonym

Synopsis

DROP [PUBLIC] SYNONYM [schema.]syn_name

Description

DROP SYNONYM deletes existing synonyms. To execute this command you must be a superuser or the owner of the synonym, and have USAGE privileges on the schema in which the synonym resides. See Section 2.2.4 for additional information about synonyms.

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.

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 Oracle-compatible 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;
3.3.39 DROP ROLE

Name

DROP ROLE -- remove a database role

Synopsis

DROP ROLE name [ CASCADE ]

Description

DROP ROLE removes the specified role. To drop a superuser role, you must be a superuser yourself; to drop non-superuser roles, you must have CREATEROLE privilege.

A role cannot be removed if it is still referenced in any database of the cluster; an error will be raised if so. Before dropping the role, you must drop all the objects it owns (or reassign their ownership) and revoke any privileges the role has been granted.

It is not necessary to remove role memberships involving the role; DROP ROLE automatically revokes any memberships of the target role in other roles, and of other roles in the target role. The other roles are not dropped nor otherwise affected.

Alternatively, if the only objects owned by the role belong within a schema that is owned by the role and has the same name as the role, the CASCADE option can be specified. In this case the issuer of the DROP ROLE name CASCADE command must be a superuser and the named role, the schema, and all objects within the schema will be deleted.

Parameters

name

The name of the role to remove.

CASCADE

If specified, also drops the schema owned by, and with the same name as the role (and all objects owned by the role belonging to the schema) as long as no other dependencies on the role or the schema exist.
Examples

To drop a role:

```sql
DROP ROLE admins;
```

See Also

CREATE ROLE, SET ROLE, GRANT, REVOKE
3.3.40 DROP SEQUENCE

Name

DROP SEQUENCE -- remove a sequence

Synopsis

DROP SEQUENCE name [, ...]

Description

DROP SEQUENCE removes sequence number generators. To execute this command you must be a superuser or the owner of the sequence.

Parameters

name

The name (optionally schema-qualified) of a sequence.

Examples

To remove the sequence, serial:

    DROP SEQUENCE serial;

See Also

ALTER SEQUENCE, CREATE SEQUENCE
3.3.41 DROP TABLE

Name

DROP TABLE -- remove a table

Synopsis

DROP TABLE name [CASCADE | RESTRICT | CASCADE CONSTRAINTS]

Description

DROP TABLE removes tables from the database. Only its owner may destroy a table. To empty a table of rows, without destroying the table, use DELETE. DROP TABLE always removes any indexes, rules, triggers, and constraints that exist for the target table.

Parameters

name

The name (optionally schema-qualified) of the table to drop.

Include the RESTRICT keyword to specify that the server should refuse to drop the table if any objects depend on it. This is the default behavior; the DROP TABLE command will report an error if any objects depend on the table.

Include the CASCADE clause to drop any objects that depend on the table.

Include the CASCADE CONSTRAINTS clause to specify that Advanced Server should drop any dependent constraints (excluding other object types) on the specified table.

Examples

The following command drops a table named emp that has no dependencies:

DROP TABLE emp;

The outcome of a DROP TABLE command will vary depending on whether the table has any dependencies - you can control the outcome by specifying a drop behavior. For example, if you create two tables, orders and items, where the items table is dependent on the orders table:

CREATE TABLE orders
  (order_id int PRIMARY KEY, order_date date, ...);
Advanced Server will perform one of the following actions when dropping the `orders` table, depending on the drop behavior that you specify:

- If you specify `DROP TABLE orders RESTRICT`, Advanced Server will report an error.

- If you specify `DROP TABLE orders CASCADE`, Advanced Server will drop the `orders` table and the `items` table.

- If you specify `DROP TABLE orders CASCADE CONSTRAINTS`, Advanced Server will drop the `orders` table and remove the foreign key specification from the `items` table, but not drop the `items` table.

See Also

`ALTER TABLE`, `CREATE TABLE`
3.3.42 DROP TABLESPACE

Name

DROP TABLESPACE -- remove a tablespace

Synopsis

DROP TABLESPACE tablespace

Description

DROP TABLESPACE removes a tablespace from the system.

A tablespace can only be dropped by its owner or a superuser. The tablespace must be empty of all database objects before it can be dropped. It is possible that objects in other databases may still reside in the tablespace even if no objects in the current database are using the tablespace.

Parameters

tablespace

The name of a tablespace.

Examples

To remove tablespace employee_space from the system:

DROP TABLESPACE employee_space;

See Also

ALTER TABLESPACE
3.3.43 DROP TRIGGER

Name

DROP TRIGGER -- remove a trigger

Synopsis

DROP TRIGGER name

Description

DROP TRIGGER removes a trigger from its associated table. The command must be run by a superuser or the owner of the table on which the trigger is defined.

Parameters

name

The name of a trigger to remove.

Examples

Remove trigger emp_sal_trig:

DROP TRIGGER emp_sal_trig;

See Also

CREATE TRIGGER
3.3.44 DROP TYPE

Name

DROP TYPE -- remove a type definition

Synopsis

DROP TYPE [ BODY ] name

Description

DROP TYPE removes the type definition. To execute this command you must be a superuser or the owner of the type.

The optional BODY qualifier applies only to object type definitions, not to collection types. If BODY is specified, only the object type body is removed – the object type specification is not dropped. If BODY is omitted, both the object type specification and body are removed.

The type will not be deleted if there are other database objects dependent upon the named type.

Parameters

name

The name of a type definition to remove.

Examples

Drop object type addr_obj_typ.

DROP TYPE addr_obj_typ;

Drop nested table type budget_tbl_typ.

DROP TYPE budget_tbl_typ;

See Also

CREATE TYPE, CREATE TYPE BODY
3.3.45 DROP USER

Name

DROP USER -- remove a database user account

Synopsis

DROP USER name [ CASCADE ]

Description

DROP USER removes the specified user. To drop a superuser, you must be a superuser yourself; to drop non-superusers, you must have CREATEROLE privilege.

A user cannot be removed if it is still referenced in any database of the cluster; an error will be raised if so. Before dropping the user, you must drop all the objects it owns (or reassign their ownership) and revoke any privileges the user has been granted.

However, it is not necessary to remove role memberships involving the user; DROP USER automatically revokes any memberships of the target user in other roles, and of other roles in the target user. The other roles are not dropped nor otherwise affected.

Alternatively, if the only objects owned by the user belong within a schema that is owned by the user and has the same name as the user, the CASCADE option can be specified. In this case the issuer of the DROP USER name CASCADE command must be a superuser and the named user, the schema, and all objects within the schema will be deleted.

Parameters

name

The name of the user to remove.

CASCADE

If specified, also drops the schema owned by, and with the same name as the user (and all objects owned by the user belonging to the schema) as long as no other dependencies on the user or the schema exist.

Examples

To drop a user account who owns no objects nor has been granted any privileges on other objects:
To drop user account, `john`, who has not been granted any privileges on any objects, and does not own any objects outside of a schema named, `john`, that is owned by user, `john`:

```sql
DROP USER john CASCADE;
```

**See Also**

`CREATE USER`, `ALTER USER`
3.3.46 DROP VIEW

Name

DROP VIEW -- remove a view

Synopsis

DROP VIEW name

Description

DROP VIEW drops an existing view. To execute this command you must be the owner of the view. The named view will not be deleted if other objects are dependent upon this view (such as a view of a view).

Parameters

name

The name (optionally schema-qualified) of the view to remove.

Examples

This command will remove the view called dept_30:

```
DROP VIEW dept_30;
```

See Also

CREATE VIEW
3.3.47 EXEC

Name

EXEC

Synopsis

EXEC function_name ['(' [argument_list] ')']

Description

EXECUTE.

Parameters

procedure_name

procedure_name is the (optionally schema-qualified) function name.

argument_list

argument_list specifies a comma-separated list of arguments required by the function. Note that each member of argument_list corresponds to a formal argument expected by the function. Each formal argument may be an IN parameter, an OUT parameter, or an INOUT parameter.

Examples

The EXEC statement may take one of several forms, depending on the arguments required by the function:

EXEC update_balance;
EXEC update_balance();
EXEC update_balance(1,2,3);
3.3.48  GRANT

Name

GRANT -- define access privileges

Synopsis

GRANT { { SELECT | INSERT | UPDATE | DELETE | REFERENCES } 
   [, ...] | ALL [ PRIVILEGES ] } 
ON tablename 
TO { username | groupname | PUBLIC } [, ...] 
[ WITH GRANT OPTION ]

GRANT { { INSERT | UPDATE | REFERENCES } (column [, ...]) } 
   [, ...] 
ON tablename 
TO { username | groupname | PUBLIC } [, ...] 
[ WITH GRANT OPTION ]

GRANT { SELECT | ALL [ PRIVILEGES ] } 
ON sequencename 
TO { username | groupname | PUBLIC } [, ...] 
[ WITH GRANT OPTION ]

GRANT { EXECUTE | ALL [ PRIVILEGES ] } 
ON FUNCTION progname 
   ( [ [ argmode ] [ argnname ] argtype ] [, ...] ) 
TO { username | groupname | PUBLIC } [, ...] 
[ WITH GRANT OPTION ]

GRANT { EXECUTE | ALL [ PRIVILEGES ] } 
ON PROCEDURE progname 
   [ ( [ [ argmode ] [ argnname ] argtype ] [, ...] ) ] 
TO { username | groupname | PUBLIC } [, ...] 
[ WITH GRANT OPTION ]

GRANT { EXECUTE | ALL [ PRIVILEGES ] } 
ON PACKAGE packagename 
TO { username | groupname | PUBLIC } [, ...] 
[ WITH GRANT OPTION ]

GRANT role [, ...] 
   TO { username | groupname | PUBLIC } [, ...] 
[ WITH ADMIN OPTION ]

GRANT { CONNECT | RESOURCE | DBA } [, ...] 
   TO { username | groupname } [, ...]
[ WITH ADMIN OPTION ]

GRANT CREATE [ PUBLIC ] DATABASE LINK
   TO { username | groupname }

GRANT DROP PUBLIC DATABASE LINK
   TO { username | groupname }

GRANT EXEMPT ACCESS POLICY
   TO { username | groupname }

Description

The **GRANT** command has three basic variants: one that grants privileges on a database object (table, view, sequence, or program), one that grants membership in a role, and one that grants system privileges. These variants are similar in many ways, but they are different enough to be described separately.

In Postgres Plus Advanced Server, the concept of users and groups has been unified into a single type of entity called a **role**. In this context, a **user** is a role that has the **LOGIN** attribute – the role may be used to create a session and connect to an application. A **group** is a role that does not have the **LOGIN** attribute – the role may not be used to create a session or connect to an application.

A role may be a member of one or more other roles, so the traditional concept of users belonging to groups is still valid. However, with the generalization of users and groups, users may “belong” to users, groups may “belong” to groups, and groups may “belong” to users, forming a general multi-level hierarchy of roles. User names and group names share the same namespace therefore it is not necessary to distinguish whether a grantee is a user or a group in the **GRANT** command.
3.3.49 GRANT on Database Objects

This variant of the GRANT command gives specific privileges on a database object to a role. These privileges are added to those already granted, if any.

The key word PUBLIC indicates that the privileges are to be granted to all roles, including those that may be created later. PUBLIC may be thought of as an implicitly defined group that always includes all roles. Any particular role will have the sum of privileges granted directly to it, privileges granted to any role it is presently a member of, and privileges granted to PUBLIC.

If the WITH GRANT OPTION is specified, the recipient of the privilege may in turn grant it to others. Without a grant option, the recipient cannot do that. Grant options cannot be granted to PUBLIC.

There is no need to grant privileges to the owner of an object (usually the user that created it), as the owner has all privileges by default. (The owner could, however, choose to revoke some of his own privileges for safety.) The right to drop an object or to alter its definition in any way is not described by a grantable privilege; it is inherent in the owner, and cannot be granted or revoked. The owner implicitly has all grant options for the object as well.

Depending on the type of object, the initial default privileges may include granting some privileges to PUBLIC. The default is no public access for tables and EXECUTE privilege for functions, procedures, and packages. The object owner may of course revoke these privileges. (For maximum security, issue the REVOKE in the same transaction that creates the object; then there is no window in which another user may use the object.)

The possible privileges are:

SELECT

Allows SELECT from any column of the specified table, view, or sequence. For sequences, this privilege also allows the use of the currval function.

INSERT

Allows INSERT of a new row into the specified table.

UPDATE

Allows UPDATE of a column of the specified table. SELECT ... FOR UPDATE also requires this privilege (besides the SELECT privilege).
DELETE

Allows **DELETE** of a row from the specified table.

REFERENCES

To create a foreign key constraint, it is necessary to have this privilege on both the referencing and referenced tables.

EXECUTE

Allows the use of the specified package, procedure, or function. When applied to a package, allows the use of all of the package’s public procedures, public functions, public variables, records, cursors and other public objects and object types. This is the only type of privilege that is applicable to functions, procedures, and packages.

The Postgres Plus Advanced Server syntax for granting the **EXECUTE** privilege is not fully Oracle compatible. Postgres Plus Advanced Server requires qualification of the program name by one of the keywords, **FUNCTION**, **PROCEDURE**, or **PACKAGE** whereas these keywords must be omitted in Oracle. In addition for functions, Postgres Plus Advanced Server requires all input (**IN**, **IN OUT**) argument data types after the function name (including an empty parenthesis if there are no function arguments). For procedures, all input argument data types must be specified if the procedure has one or more input arguments. In Oracle, function and procedure signatures must be omitted. This is due to the fact that all programs share the same namespace in Oracle, whereas functions, procedures, and packages have their own individual namespace in Postgres Plus Advanced Server to allow program name overloading to a certain extent.

ALL PRIVILEGES

Grant all of the available privileges at once.

The privileges required by other commands are listed on the reference page of the respective command.
3.3.50 GRANT on Roles

This variant of the GRANT command grants membership in a role to one or more other roles. Membership in a role is significant because it conveys the privileges granted to a role to each of its members.

If the WITH ADMIN OPTION is specified, the member may in turn grant membership in the role to others, and revoke membership in the role as well. Without the admin option, ordinary users cannot do that.

Database superusers can grant or revoke membership in any role to anyone. Roles having the CREATEROLE privilege can grant or revoke membership in any role that is not a superuser.

There are three pre-defined roles that have the following meanings:

CONNECT

Granting the CONNECT role is equivalent to giving the grantee the LOGIN privilege. The grantor must have the CREATEROLE privilege.

RESOURCE

Granting the RESOURCE role is equivalent to granting the CREATE and USAGE privileges on a schema that has the same name as the grantee. This schema must exist before the grant is given. The grantor must have the privilege to grant CREATE or USAGE privileges on this schema to the grantee.

DBA

Granting the DBA role is equivalent to making the grantee a superuser. The grantor must be a superuser.

Notes

The REVOKE command is used to revoke access privileges.

When a non-owner of an object attempts to GRANT privileges on the object, the command will fail outright if the user has no privileges whatsoever on the object. As long as a privilege is available, the command will proceed, but it will grant only those privileges for which the user has grant options. The GRANT ALL PRIVILEGES forms will issue a warning message if no grant options are held, while the other forms will issue a warning if grant options for any of the privileges specifically named in the command are not held.
(In principle these statements apply to the object owner as well, but since the owner is always treated as holding all grant options, the cases can never occur.)

It should be noted that database superusers can access all objects regardless of object privilege settings. This is comparable to the rights of root in a Unix system. As with root, it’s unwise to operate as a superuser except when absolutely necessary.

If a superuser chooses to issue a `GRANT` or `REVOKE` command, the command is performed as though it were issued by the owner of the affected object. In particular, privileges granted via such a command will appear to have been granted by the object owner. (For role membership, the membership appears to have been granted by the containing role itself.)

`GRANT` and `REVOKE` can also be done by a role that is not the owner of the affected object, but is a member of the role that owns the object, or is a member of a role that holds privileges `WITH GRANT OPTION` on the object. In this case the privileges will be recorded as having been granted by the role that actually owns the object or holds the privileges `WITH GRANT OPTION`.

For example, if table `t1` is owned by role `g1`, of which role `u1` is a member, then `u1` can grant privileges on `t1` to `u2`, but those privileges will appear to have been granted directly by `g1`. Any other member of role `g1` could revoke them later.

If the role executing `GRANT` holds the required privileges indirectly via more than one role membership path, it is unspecified which containing role will be recorded as having done the grant. In such cases it is best practice to use `SET ROLE` to become the specific role you want to do the `GRANT` as.

Currently, Postgres Plus Advanced Server does not support granting or revoking privileges for individual columns of a table. One possible workaround is to create a view having just the desired columns and then grant privileges to that view.

**Examples**

Grant insert privilege to all users on table `emp`:

```
GRANT INSERT ON emp TO PUBLIC;
```

Grant all available privileges to user `mary` on view `salesemp`:

```
GRANT ALL PRIVILEGES ON salesemp TO mary;
```

Note that while the above will indeed grant all privileges if executed by a superuser or the owner of `emp`, when executed by someone else it will only grant those permissions for which the someone else has grant options.
Grant membership in role `admins` to user `joe`:

```
GRANT admins TO joe;
```

Grant `CONNECT` privilege to user `joe`:

```
GRANT CONNECT TO joe;
```

**See Also**

`REVOKE`, `SET ROLE`
3.3.51 GRANT on System Privileges

This variant of the GRANT command gives a role the ability to perform certain system operations within a database. System privileges relate to the ability to create or delete certain database objects that are not necessarily within the confines of one schema. Only database superusers can grant system privileges.

CREATE [PUBLIC] DATABASE LINK

The CREATE [PUBLIC] DATABASE LINK privilege allows the specified role to create a database link. Include the PUBLIC keyword to allow the role to create public database links; omit the PUBLIC keyword to allow the specified role to create private database links.

DROP PUBLIC DATABASE LINK

The DROP PUBLIC DATABASE LINK privilege allows a role to drop a public database link. System privileges are not required to drop a private database link. A private database link may be dropped by the link owner or a database superuser.

EXEMPT ACCESS POLICY

The EXEMPT ACCESS POLICY privilege allows a role to execute a SQL command without invoking any policy function that may be associated with the target database object. That is, the role is exempt from all security policies in the database. See Section 7.7 for information about DBMS_RLS security policies.

The EXEMPT ACCESS POLICY privilege is not inheritable by membership to a role that has the EXEMPT ACCESS POLICY privilege. For example, the following sequence of GRANT commands does not result in user joe obtaining the EXEMPT ACCESS POLICY privilege even though joe is granted membership to the enterprisedb role, which has been granted the EXEMPT ACCESS POLICY privilege:

```
GRANT EXEMPT ACCESS POLICY TO enterprisedb;
GRANT enterprisedb TO joc;
```

The rolpolicyexempt column of the system catalog table pg_authid is set to true if a role has the EXEMPT ACCESS POLICY privilege.
Examples

Grant CREATE PUBLIC DATABASE LINK privilege to user joe:

```
GRANT CREATE PUBLIC DATABASE LINK TO joe;
```

Grant DROP PUBLIC DATABASE LINK privilege to user joe:

```
GRANT DROP PUBLIC DATABASE LINK TO joe;
```

Grant the EXEMPT ACCESS POLICY privilege to user joe:

```
GRANT EXEMPT ACCESS POLICY TO joe;
```

Using the ALTER ROLE Command to Assign System Privileges

The Advanced Server ALTER ROLE command also supports syntax that you can use to assign:

- the privilege required to create a public or private database link.
- the privilege required to drop a public database link.
- the EXEMPT ACCESS POLICY privilege.

The ALTER ROLE syntax is functionally equivalent to the respective Oracle compatible commands. For more information about using the ALTER ROLE command to manage database link privileges, see Section 3.3.3, ALTER ROLE.

See Also

REVOKE
3.3.52 INSERT

Name

INSERT -- create new rows in a table

Synopsis

```
INSERT INTO table[@dblink ] [ ( column [, ...] ) ]
{ VALUES ( { expression | DEFAULT } [, ...] )
 [ RETURNING return_expression [, ...]
   { INTO { record | variable [, ...] }
    | BULK COLLECT INTO collection [, ...] } ]
 | query }
```

Description

INSERT allows you to insert new rows into a table. You can insert a single row at a time
or several rows as a result of a query.

The columns in the target list may be listed in any order. Each column not present in the
target list will be inserted using a default value, either its declared default value or null.

If the expression for each column is not of the correct data type, automatic type
conversion will be attempted.

The RETURNING INTO { record | variable [, ...] } clause may only be
specified when the INSERT command is used within an SPL program and only when the
VALUES clause is used.

The RETURNING BULK COLLECT INTO collection [, ...] clause may only be
specified if the INSERT command is used within an SPL program. If more than one
collection is specified as the target of the BULK COLLECT INTO clause, then each
collection must consist of a single, scalar field – i.e., collection must not be a
record. return_expression evaluated for each inserted row, becomes an element in
collection starting with the first element. Any existing rows in collection are
deleted. If the result set is empty, then collection will be empty.

You must have INSERT privilege to a table in order to insert into it. If you use the query
clause to insert rows from a query, you also need to have SELECT privilege on any table
used in the query.
Parameters

`table`

The name (optionally schema-qualified) of an existing table.

`dblink`

Database link name identifying a remote database. See the `CREATE DATABASE LINK` command for information on database links.

`column`

The name of a column in `table`.

`expression`

An expression or value to assign to `column`.

`DEFAULT`

This column will be filled with its default value.

`query`

A query (SELECT statement) that supplies the rows to be inserted. Refer to the SELECT command for a description of the syntax.

`return_expression`

An expression that may include one or more columns from `table`. If a column name from `table` is specified in `return_expression`, the value substituted for the column when `return_expression` is evaluated is determined as follows:

If the column specified in `return_expression` is assigned a value in the INSERT command, then the assigned value is used in the evaluation of `return_expression`.

If the column specified in `return_expression` is not assigned a value in the INSERT command and there is no default value for the column in the table’s column definition, then null is used in the evaluation of `return_expression`.

If the column specified in `return_expression` is not assigned a value in the INSERT command and there is a default value for the column in the
table’s column definition, then the default value is used in the evaluation of `return_expression`.

**record**

A record whose field the evaluated `return_expression` is to be assigned. The first `return_expression` is assigned to the first field in `record`, the second `return_expression` is assigned to the second field in `record`, etc. The number of fields in `record` must exactly match the number of expressions and the fields must be type-compatible with their assigned expressions.

**variable**

A variable to which the evaluated `return_expression` is to be assigned. If more than one `return_expression` and `variable` are specified, the first `return_expression` is assigned to the first `variable`, the second `return_expression` is assigned to the second `variable`, etc. The number of variables specified following the INTO keyword must exactly match the number of expressions following the RETURNING keyword and the variables must be type-compatible with their assigned expressions.

**collection**

A collection in which an element is created from the evaluated `return_expression`. There can be either a single collection which may be a collection of a single field or a collection of a record type, or there may be more than one collection in which case each collection must consist of a single field. The number of return expressions must match in number and order the number of fields in all specified collections. Each corresponding `return_expression` and `collection` field must be type-compatible.

**Examples**

Insert a single row into table `emp`:

```sql
INSERT INTO emp VALUES (8021,'JOHN','SALESMAN',7698,'22-FEB-07',1250,500,30);
```

In this second example, the column, `comm`, is omitted and therefore it will have the default value of null:

```sql
INSERT INTO emp (empno, ename, job, mgr, hiredate, sal, deptno) VALUES (8022,'PETERS','CLERK',7698,'03-DEC-06',950,30);
```

The third example uses the DEFAULT clause for the `hiredate` and `comm` columns rather than specifying a value:

```sql
INSERT INTO emp VALUES (8023,'FORD','ANALYST',7566,NULL,3000,NULL,20);
```
This example creates a table for the department names and then inserts into the table by selecting from the dname column of the dept table:

```sql
CREATE TABLE deptnames (  
depname VARCHAR2(14)
);

INSERT INTO deptnames SELECT dname FROM dept;
```
3.3.53 LOCK

Name

LOCK -- lock a table

Synopsis

LOCK TABLE name [, ...] IN lockmode MODE [ NOWAIT ]

where lockmode is one of:

ROW SHARE | ROW EXCLUSIVE | SHARE | SHARE ROW EXCLUSIVE | EXCLUSIVE

Description

LOCK TABLE obtains a table-level lock, waiting if necessary for any conflicting locks to be released. If NOWAIT is specified, LOCK TABLE does not wait to acquire the desired lock: if it cannot be acquired immediately, the command is aborted and an error is emitted. Once obtained, the lock is held for the remainder of the current transaction. (There is no UNLOCK TABLE command; locks are always released at transaction end.)

When acquiring locks automatically for commands that reference tables, Postgres Plus Advanced Server always uses the least restrictive lock mode possible. LOCK TABLE provides for cases when you might need more restrictive locking. For example, suppose an application runs a transaction at the isolation level read committed and needs to ensure that data in a table remains stable for the duration of the transaction. To achieve this you could obtain SHARE lock mode over the table before querying. This will prevent concurrent data changes and ensure subsequent reads of the table see a stable view of committed data, because SHARE lock mode conflicts with the ROW EXCLUSIVE lock acquired by writers, and your LOCK TABLE name IN SHARE MODE statement will wait until any concurrent holders of ROW EXCLUSIVE mode locks commit or roll back. Thus, once you obtain the lock, there are no uncommitted writes outstanding; furthermore none can begin until you release the lock.

To achieve a similar effect when running a transaction at the isolation level serializable, you have to execute the LOCK TABLE statement before executing any data modification statement. A serializable transaction’s view of data will be frozen when its first data modification statement begins. A later LOCK TABLE will still prevent concurrent writes - but it won’t ensure that what the transaction reads corresponds to the latest committed values.
If a transaction of this sort is going to change the data in the table, then it should use
SHARE ROW EXCLUSIVE lock mode instead of SHARE mode.

This ensures that only one transaction of this type runs at a time. Without this, a deadlock
is possible: two transactions might both acquire SHARE mode, and then be unable to also
acquire ROW EXCLUSIVE mode to actually perform their updates. (Note that a
transaction’s own locks never conflict, so a transaction can acquire ROW EXCLUSIVE
mode when it holds SHARE mode - but not if anyone else holds SHARE mode.) To avoid
deadlocks, make sure all transactions acquire locks on the same objects in the same order,
and if multiple lock modes are involved for a single object, then transactions should
always acquire the most restrictive mode first.

Parameters

name

The name (optionally schema-qualified) of an existing table to lock.

The command LOCK TABLE a, b; is equivalent to LOCK TABLE a; LOCK
TABLE b. The tables are locked one-by-one in the order specified in the LOCK
TABLE command.

lockmode

The lock mode specifies which locks this lock conflicts with.

If no lock mode is specified, then ACCESS EXCLUSIVE, the most restrictive
mode, is used. (ACCESS EXCLUSIVE is not an Oracle compatible term. In
Postgres Plus Advanced Server, this mode ensures that no other transaction can
access the locked table in any manner.)

NOWAIT

Specifies that LOCK TABLE should not wait for any conflicting locks to be
released: if the specified lock cannot be immediately acquired without waiting,
the transaction is aborted.

Notes

All forms of LOCK require UPDATE and/or DELETE privileges.

LOCK TABLE is useful only inside a transaction block since the lock is dropped as soon
as the transaction ends. A LOCK TABLE command appearing outside any transaction
block forms a self-contained transaction, so the lock will be dropped as soon as it is
obtained.
LOCK TABLE only deals with table-level locks, and so the mode names involving ROW are all misnomers. These mode names should generally be read as indicating the intention of the user to acquire row-level locks within the locked table. Also, ROW EXCLUSIVE mode is a sharable table lock. Keep in mind that all the lock modes have identical semantics so far as LOCK TABLE is concerned, differing only in the rules about which modes conflict with which.
### 3.3.54 REVOKE

**Name**

REVOKE -- remove access privileges

**Synopsis**

REVOKE { { SELECT | INSERT | UPDATE | DELETE | REFERENCES } } [, ...] | ALL [ PRIVILEGES ] ]
ON tablename
FROM { username | groupname | PUBLIC } [, ...]
[ CASCADE | RESTRICT ]

REVOKE { SELECT | ALL [ PRIVILEGES ] ]
ON sequencename
FROM { username | groupname | PUBLIC } [, ...]
[ CASCADE | RESTRICT ]

REVOKE { EXECUTE | ALL [ PRIVILEGES ] ]
ON FUNCTION progname
( ( [ [ argmode ] [ argname ] argtype ] [, ...] ) )
FROM { username | groupname | PUBLIC } [, ...]
[ CASCADE | RESTRICT ]

REVOKE { EXECUTE | ALL [ PRIVILEGES ] ]
ON PROCEDURE progname
( ( [ [ argmode ] [ argname ] argtype ] [, ...] ) )
FROM { username | groupname | PUBLIC } [, ...]
[ CASCADE | RESTRICT ]

REVOKE { EXECUTE | ALL [ PRIVILEGES ] ]
ON PACKAGE packagename
FROM { username | groupname | PUBLIC } [, ...]
[ CASCADE | RESTRICT ]

REVOKE role [, ...] FROM { username | groupname | PUBLIC } [, ...]
[ CASCADE | RESTRICT ]

REVOKE { CONNECT | RESOURCE | DBA | DBA } [, ...]
FROM { username | groupname } [, ...]

REVOKE CREATE [ PUBLIC ] DATABASE LINK
FROM { username | groupname }

REVOKE DROP PUBLIC DATABASE LINK
FROM { username | groupname }
REVOKE EXEMPT ACCESS POLICY
    FROM { username | groupname }

Description

The REVOKE command revokes previously granted privileges from one or more roles.
The key word PUBLIC refers to the implicitly defined group of all roles.

See the description of the GRANT command for the meaning of the privilege types.

Note that any particular role will have the sum of privileges granted directly to it,
privileges granted to any role it is presently a member of, and privileges granted to
PUBLIC. Thus, for example, revoking SELECT privilege from PUBLIC does not
necessarily mean that all roles have lost SELECT privilege on the object: those who have
it granted directly or via another role will still have it.

If the privilege had been granted with the grant option, the grant option for the privilege
is revoked as well as the privilege, itself.

If a user holds a privilege with grant option and has granted it to other users then the
privileges held by those other users are called dependent privileges. If the privilege or the
grant option held by the first user is being revoked and dependent privileges exist, those
dependent privileges are also revoked if CASCADE is specified, else the revoke action will
fail. This recursive revocation only affects privileges that were granted through a chain of
users that is traceable to the user that is the subject of this REVOKE command. Thus, the
affected users may effectively keep the privilege if it was also granted through other
users.

Note: CASCADE is not an Oracle compatible option. By default Oracle always cascades
dependent privileges, but Postgres Plus Advanced Server requires the CASCADE keyword
to be explicitly given, otherwise the REVOKE command will fail.

When revoking membership in a role, GRANT OPTION is instead called ADMIN OPTION,
but the behavior is similar.

Notes

A user can only revoke privileges that were granted directly by that user. If, for example,
user A has granted a privilege with grant option to user B, and user B has in turned
granted it to user C, then user A cannot revoke the privilege directly from C. Instead, user
A could revoke the grant option from user B and use the CASCADE option so that the
privilege is in turn revoked from user C. For another example, if both A and B have
granted the same privilege to C, A can revoke his own grant but not B’s grant, so C will
still effectively have the privilege.
When a non-owner of an object attempts to REVOKE privileges on the object, the command will fail outright if the user has no privileges whatsoever on the object. As long as some privilege is available, the command will proceed, but it will revoke only those privileges for which the user has grant options. The REVOKE ALL PRIVILEGES forms will issue a warning message if no grant options are held, while the other forms will issue a warning if grant options for any of the privileges specifically named in the command are not held. (In principle these statements apply to the object owner as well, but since the owner is always treated as holding all grant options, the cases can never occur.)

If a superuser chooses to issue a GRANT or REVOKE command, the command is performed as though it were issued by the owner of the affected object. Since all privileges ultimately come from the object owner (possibly indirectly via chains of grant options), it is possible for a superuser to revoke all privileges, but this may require use of CASCADE as stated above.

REVOKE can also be done by a role that is not the owner of the affected object, but is a member of the role that owns the object, or is a member of a role that holds privileges WITH GRANT OPTION on the object. In this case the command is performed as though it were issued by the containing role that actually owns the object or holds the privileges WITH GRANT OPTION. For example, if table t1 is owned by role g1, of which role u1 is a member, then u1 can revoke privileges on t1 that are recorded as being granted by g1. This would include grants made by u1 as well as by other members of role g1.

If the role executing REVOKE holds privileges indirectly via more than one role membership path, it is unspecified which containing role will be used to perform the command. In such cases it is best practice to use SET ROLE to become the specific role you want to do the REVOKE as. Failure to do so may lead to revoking privileges other than the ones you intended, or not revoking anything at all.

Please Note: The Advanced Server ALTER ROLE command also supports syntax that revokes the system privileges required to create a public or private database link, or exemptions from fine-grained access control policies (DBMS_RLS). The ALTER ROLE syntax is functionally equivalent to the respective Oracle compatible REVOKE command. For more information about using the ALTER ROLE command to manage system privileges, see Section 3.3.3, ALTER ROLE.

Examples

Revoke insert privilege for the public on table emp:

```sql
REVOKE INSERT ON emp FROM PUBLIC;
```

Revoke all privileges from user mary on view salesemp:

```sql
REVOKE ALL PRIVILEGES ON salesemp FROM mary;
```
Note that this actually means “revoke all privileges that I granted”.

Revoke membership in role admins from user joe:

```
REVOKE admins FROM joe;
```

Revoke CONNECT privilege from user joe:

```
REVOKE CONNECT FROM joe;
```

Revoke CREATE DATABASE LINK privilege from user joe:

```
REVOKE CREATE DATABASE LINK FROM joe;
```

Revoke the EXEMPT ACCESS POLICY privilege from user joe:

```
REVOKE EXEMPT ACCESS POLICY FROM joe;
```

See Also

GRANT, SET ROLE
3.3.55 ROLLBACK

Name

ROLLBACK -- abort the current transaction

Synopsis

ROLLBACK [ WORK ]

Description

ROLLBACK rolls back the current transaction and causes all the updates made by the transaction to be discarded.

Parameters

WORK

Optional key word - has no effect.

Notes

Use COMMIT to successfully terminate a transaction.

Issuing ROLLBACK when not inside a transaction does no harm.

Examples

To abort all changes:

ROLLBACK;

See Also

COMMIT, ROLLBACK TO SAVEPOINT, SAVEPOINT
3.3.56 ROLLBACK TO SAVEPOINT

Name

ROLLBACK TO SAVEPOINT -- roll back to a savepoint

Synopsis

ROLLBACK [ WORK ] TO [ SAVEPOINT ] savepoint_name

Description

Roll back all commands that were executed after the savepoint was established. The savepoint remains valid and can be rolled back to again later, if needed.

ROLLBACK TO SAVEPOINT implicitly destroys all savepoints that were established after the named savepoint.

Parameters

savepoint_name

The savepoint to which to roll back.

Notes

Specifying a savepoint name that has not been established is an error.

ROLLBACK TO SAVEPOINT is not supported within SPL programs.

Examples

To undo the effects of the commands executed savepoint depts was established:

```sql
\set AUTOCOMMIT off
INSERT INTO dept VALUES (50, 'HR', 'NEW YORK');
SAVEPOINT depts;
INSERT INTO emp (empno, ename, deptno) VALUES (9001, 'JONES', 50);
INSERT INTO emp (empno, ename, deptno) VALUES (9002, 'ALICE', 50);
ROLLBACK TO SAVEPOINT depts;
```

See Also

COMMIT, ROLLBACK, SAVEPOINT
3.3.57 **SAVEPOINT**

**Name**

SAVEPOINT -- define a new savepoint within the current transaction

**Synopsis**

SAVEPOINT `savepoint_name`

**Description**

SAVEPOINT establishes a new savepoint within the current transaction.

A savepoint is a special mark inside a transaction that allows all commands that are executed after it was established to be rolled back, restoring the transaction state to what it was at the time of the savepoint.

**Parameters**

`savepoint_name`

The name to be given to the savepoint.

**Notes**

Use **ROLLBACK TO SAVEPOINT** to roll back to a savepoint.

Savepoints can only be established when inside a transaction block. There can be multiple savepoints defined within a transaction.

When another savepoint is established with the same name as a previous savepoint, the old savepoint is kept, though only the more recent one will be used when rolling back.

**SAVEPOINT** is not supported within SPL programs.

**Examples**

To establish a savepoint and later undo the effects of all commands executed after it was established:

```
\set AUTOCOMMIT off
INSERT INTO dept VALUES (50, 'HR', 'NEW YORK');
SAVEPOINT depts;
INSERT INTO emp (empno, ename, deptno) VALUES (9001, 'JONES', 50);
INSERT INTO emp (empno, ename, deptno) VALUES (9002, 'ALICE', 50);
```
SAVEPOINT emps;
INSERT INTO jobhist VALUES (9001,'17-SEP-07',NULL,'CLERK',800,NULL,50,'New Hire');
INSERT INTO jobhist VALUES (9002,'20-SEP-07',NULL,'CLERK',700,NULL,50,'New Hire');
ROLLBACK TO depts;
COMMIT;

The above transaction will commit a row into the dept table, but the inserts into the emp and jobhist tables are rolled back.

See Also

COMMIT, ROLLBACK, ROLLBACK TO SAVEPOINT
**3.3.58** SELECT

**Name**

SELECT -- retrieve rows from a table or view

**Synopsis**

SELECT [ optimizer_hint ] [ ALL | DISTINCT ]
  * | expression [ AS output_name ] [, ...]
FROM from_item [, ...]
  [ WHERE condition ]
  [ [ START WITH start_expression ]
    CONNECT BY { PRIOR parent_expr = child_expr |
    child_expr = PRIOR parent_expr }
    [ ORDER SIBILINGS BY expression [ ASC | DESC ] [, ... ] ]
  [ GROUP BY expression [, ...] [ LEVEL ] ]
  [ HAVING condition [, ... ] ]
  [ { UNION [ ALL ] | INTERSECT | MINUS } select ]
  [ ORDER BY expression [ ASC | DESC ] [, ... ] ]
  [ FOR UPDATE ]

where from_item can be one of:

  table_name[@dblink ] [ alias ]
  ( select ) alias
  from_item [ NATURAL ] join_type from_item
    [ ON join_condition | USING ( join_column [, ... ] ) ]

**Description**

SELECT retrieves rows from one or more tables. The general processing of SELECT is as follows:

1. All elements in the FROM list are computed. (Each element in the FROM list is a real or virtual table.) If more than one element is specified in the FROM list, they are cross-joined together. (See FROM clause, below.)
2. If the WHERE clause is specified, all rows that do not satisfy the condition are eliminated from the output. (See WHERE clause, below.)
3. If the GROUP BY clause is specified, the output is divided into groups of rows that match on one or more values. If the HAVING clause is present, it eliminates groups that do not satisfy the given condition. (See GROUP BY clause and HAVING clause below.)
4. Using the operators UNION, INTERSECT, and MINUS, the output of more than one SELECT statement can be combined to form a single result set. The UNION
operator returns all rows that are in one or both of the result sets. The **INTERSECT** operator returns all rows that are strictly in both result sets. The **MINUS** operator returns the rows that are in the first result set but not in the second. In all three cases, duplicate rows are eliminated. In the case of the **UNION** operator, if **ALL** is specified then duplicates are not eliminated. (See **UNION** clause, **INTERSECT** clause, and **MINUS** clause below.)

5. The actual output rows are computed using the **SELECT** output expressions for each selected row. (See **SELECT** list below.)

6. The **CONNECT BY** clause is used to select data that has a hierarchical relationship. Such data has a parent-child relationship between rows. (See **CONNECT BY** clause.)

7. If the **ORDER BY** clause is specified, the returned rows are sorted in the specified order. If **ORDER BY** is not given, the rows are returned in whatever order the system finds fastest to produce. (See **ORDER BY** clause below.)

8. **DISTINCT** eliminates duplicate rows from the result. **ALL** (the default) will return all candidate rows, including duplicates. (See **DISTINCT** clause below.)

9. The **FOR UPDATE** clause causes the **SELECT** statement to lock the selected rows against concurrent updates. (See **FOR UPDATE** clause below.)

You must have **SELECT** privilege on a table to read its values. The use of **FOR UPDATE** requires **UPDATE** privilege as well.

**Parameters**

*optimizer_hint*

Comment-embedded hints to the optimizer for selection of an execution plan. See **Section 3.4** for information about optimizer hints.

The remaining parameters are discussed within the following sections.

### 3.3.58.1 FROM Clause

The **FROM** clause specifies one or more source tables for the **SELECT**. If multiple sources are specified, the result is the Cartesian product (cross join) of all the sources. Usually qualification conditions are added to restrict the returned rows to a small subset of the Cartesian product.

The **FROM** clause can contain the following elements:

*table_name[@dblink]*
The name (optionally schema-qualified) of an existing table or view. *dblink* is a
database link name identifying a remote database. See the CREATE DATABASE
LINK command for information on database links.

**alias**

A substitute name for the FROM item containing the alias. An alias is used for
brevity or to eliminate ambiguity for self-joins (where the same table is scanned
multiple times). When an alias is provided, it completely hides the actual name of
the table or function; for example given FROM foo AS f, the remainder of the
SELECT must refer to this FROM item as f not foo.

**select**

A sub-SELECT can appear in the FROM clause. This acts as though its output were
created as a temporary table for the duration of this single SELECT command.
Note that the sub-SELECT must be surrounded by parentheses, and an alias must
be provided for it.

**join_type**

One of

* [ INNER ] JOIN
* LEFT [ OUTER ] JOIN
* RIGHT [ OUTER ] JOIN
* FULL [ OUTER ] JOIN
* CROSS JOIN

For the INNER and OUTER join types, a join condition must be specified, namely
exactly one of NATURAL, ON join_condition, or USING (join_column [,,
... ] ). See below for the meaning. For CROSS JOIN, none of these clauses
may appear.

A JOIN clause combines two FROM items. Use parentheses if necessary to
determine the order of nesting. In the absence of parentheses, JOINS nest left-to-
right. In any case JOIN binds more tightly than the commas separating FROM
items.

CROSS JOIN and INNER JOIN produce a simple Cartesian product, the same
result as you get from listing the two items at the top level of FROM, but restricted
by the join condition (if any). CROSS JOIN is equivalent to INNER JOIN ON
(TRUE), that is, no rows are removed by qualification. These join types are just a
notational convenience, since they do nothing you couldn’t do with plain FROM
and WHERE.
LEFT OUTER JOIN returns all rows in the qualified Cartesian product (i.e., all combined rows that pass its join condition), plus one copy of each row in the left-hand table for which there was no right-hand row that passed the join condition. This left-hand row is extended to the full width of the joined table by inserting null values for the right-hand columns. Note that only the JOIN clause’s own condition is considered while deciding which rows have matches. Outer conditions are applied afterwards.

Conversely, RIGHT OUTER JOIN returns all the joined rows, plus one row for each unmatched right-hand row (extended with nulls on the left). This is just a notational convenience, since you could convert it to a LEFT OUTER JOIN by switching the left and right inputs.

FULL OUTER JOIN returns all the joined rows, plus one row for each unmatched left-hand row (extended with nulls on the right), plus one row for each unmatched right-hand row (extended with nulls on the left).

ON join_condition

join_condition is an expression resulting in a value of type BOOLEAN (similar to a WHERE clause) that specifies which rows in a join are considered to match.

USING (join_column [, ... ] )

A clause of the form USING (a, b, ... ) is shorthand for ON left_table.a = right_table.a AND left_table.b = right_table.b .... Also, USING implies that only one of each pair of equivalent columns will be included in the join output, not both.

NATURAL

NATURAL is shorthand for a USING list that mentions all columns in the two tables that have the same names.

3.3.58.2 WHERE Clause

The optional WHERE clause has the general form

WHERE condition

where condition is any expression that evaluates to a result of type BOOLEAN. Any row that does not satisfy this condition will be eliminated from the output. A row satisfies the condition if it returns TRUE when the actual row values are substituted for any variable references.
3.3.58.3 GROUP BY Clause

The optional GROUP BY clause has the general form

```
GROUP BY expression [, ...]
```

GROUP BY will condense into a single row all selected rows that share the same values for the grouped expressions. expression can be an input column name, or the name or ordinal number of an output column (SELECT list item), or an arbitrary expression formed from input-column values. In case of ambiguity, a GROUP BY name will be interpreted as an input-column name rather than an output column name.

Aggregate functions, if any are used, are computed across all rows making up each group, producing a separate value for each group (whereas without GROUP BY, an aggregate produces a single value computed across all the selected rows). When GROUP BY is present, it is not valid for the SELECT list expressions to refer to ungrouped columns except within aggregate functions, since there would be more than one possible value to return for an ungrouped column.

3.3.58.4 HAVING Clause

The optional HAVING clause has the general form

```
HAVING condition
```

where condition is the same as specified for the WHERE clause.

HAVING eliminates group rows that do not satisfy the condition. HAVING is different from WHERE; WHERE filters individual rows before the application of GROUP BY, while HAVING filters group rows created by GROUP BY. Each column referenced in condition must unambiguously reference a grouping column, unless the reference appears within an aggregate function.

3.3.58.5 SELECT List

The SELECT list (between the key words SELECT and FROM) specifies expressions that form the output rows of the SELECT statement. The expressions can (and usually do) refer to columns computed in the FROM clause. Using the clause AS output_name, another name can be specified for an output column. This name is primarily used to label the column for display. It can also be used to refer to the column’s value in ORDER BY
and GROUP BY clauses, but not in the WHERE or HAVING clauses; there you must write out the expression instead.

Instead of an expression, * can be written in the output list as a shorthand for all the columns of the selected rows.

### 3.3.58.6 UNION Clause

The UNION clause has this general form:

```
select_statement UNION [ ALL ] select_statement
```

`select_statement` is any SELECT statement without an ORDER BY or FOR UPDATE clause. (ORDER BY can be attached to a sub-expression if it is enclosed in parentheses. Without parentheses, these clauses will be taken to apply to the result of the UNION, not to its right-hand input expression.)

The UNION operator computes the set union of the rows returned by the involved SELECT statements. A row is in the set union of two result sets if it appears in at least one of the result sets. The two SELECT statements that represent the direct operands of the UNION must produce the same number of columns, and corresponding columns must be of compatible data types.

The result of UNION does not contain any duplicate rows unless the ALL option is specified. ALL prevents elimination of duplicates.

Multiple UNION operators in the same SELECT statement are evaluated left to right, unless otherwise indicated by parentheses.

Currently, FOR UPDATE may not be specified either for a UNION result or for any input of a UNION.

### 3.3.58.7 INTERSECT Clause

The INTERSECT clause has this general form:

```
select_statement INTERSECT select_statement
```

`select_statement` is any SELECT statement without an ORDER BY or FOR UPDATE clause.
The `INTERSECT` operator computes the set intersection of the rows returned by the involved `SELECT` statements. A row is in the intersection of two result sets if it appears in both result sets.

The result of `INTERSECT` does not contain any duplicate rows.

Multiple `INTERSECT` operators in the same `SELECT` statement are evaluated left to right, unless parentheses dictate otherwise. `INTERSECT` binds more tightly than `UNION`. That is, `A UNION B INTERSECT C` will be read as `A UNION (B INTERSECT C)`.

### 3.3.58.8 MINUS Clause

The `MINUS` clause has this general form:

```
select_statement MINUS select_statement
```

*`select_statement`* is any `SELECT` statement without an `ORDER BY` or `FOR UPDATE` clause.

The `MINUS` operator computes the set of rows that are in the result of the left `SELECT` statement but not in the result of the right one.

The result of `MINUS` does not contain any duplicate rows.

Multiple `MINUS` operators in the same `SELECT` statement are evaluated left to right, unless parentheses dictate otherwise. `MINUS` binds at the same level as `UNION`.

### 3.3.58.9 CONNECT BY Clause

The `CONNECT BY` clause determines the parent-child relationship of rows when performing a hierarchical query. It has the general form:

```
CONNECT BY { PRIOR parent_expr = child_expr | 
    child_expr = PRIOR parent_expr }
```

*`parent_expr`* is evaluated on a candidate parent row. If `parent_expr = child_expr` results in `TRUE` for a row returned by the `FROM` clause, then this row is considered a child of the parent.

The following optional clauses may be specified in conjunction with the `CONNECT BY` clause:
START WITH \textit{start}\_expression

The rows returned by the \texttt{FROM} clause on which \textit{start}\_expression evaluates to \texttt{TRUE} become the root nodes of the hierarchy.

ORDER SIBLINGS BY \textit{expression} [ ASC | DESC ] [, ...]

Sibling rows of the hierarchy are ordered by \textit{expression} in the result set.

\textbf{Note:} Advanced Server does not support the use of \texttt{AND} (or other operators) in the \texttt{CONNECT BY} clause.

(See Section 2.2.5 for additional information on hierarchical queries.)

\textbf{3.3.58.10 ORDER BY Clause}

The optional \texttt{ORDER BY} clause has this general form:

\texttt{ORDER BY \textit{expression} [ ASC | DESC ] [, ...]}

\textit{expression} can be the name or ordinal number of an output column (\texttt{SELECT} list item), or it can be an arbitrary expression formed from input-column values.

The \texttt{ORDER BY} clause causes the result rows to be sorted according to the specified expressions. If two rows are equal according to the leftmost expression, they are compared according to the next expression and so on. If they are equal according to all specified expressions, they are returned in an implementation-dependent order.

The ordinal number refers to the ordinal (left-to-right) position of the result column. This feature makes it possible to define an ordering on the basis of a column that does not have a unique name. This is never absolutely necessary because it is always possible to assign a name to a result column using the \texttt{AS} clause.

It is also possible to use arbitrary expressions in the \texttt{ORDER BY} clause, including columns that do not appear in the \texttt{SELECT} result list. Thus the following statement is valid:

\texttt{SELECT ename FROM emp ORDER BY empno;}

A limitation of this feature is that an \texttt{ORDER BY} clause applying to the result of a \texttt{UNION}, \texttt{INTERSECT}, or \texttt{MINUS} clause may only specify an output column name or number, not an expression.

If an \texttt{ORDER BY} expression is a simple name that matches both a result column name and an input column name, \texttt{ORDER BY} will interpret it as the result column name. This is the
opposite of the choice that GROUP BY will make in the same situation. This inconsistency is made to be compatible with the SQL standard.

Optionally one may add the key word ASC (ascending) or DESC (descending) after any expression in the ORDER BY clause. If not specified, ASC is assumed by default.

The null value sorts higher than any other value. In other words, with ascending sort order, null values sort at the end, and with descending sort order, null values sort at the beginning.

Character-string data is sorted according to the locale-specific collation order that was established when the database cluster was initialized.

### 3.3.58.11 DISTINCT Clause

If DISTINCT is specified, all duplicate rows are removed from the result set (one row is kept from each group of duplicates). ALL specifies the opposite: all rows are kept; that is the default.

### 3.3.58.12 FOR UPDATE Clause

The FOR UPDATE clause has this form:

```
FOR UPDATE
```

FOR UPDATE causes the rows retrieved by the SELECT statement to be locked as though for update. This prevents them from being modified or deleted by other transactions until the current transaction ends. That is, other transactions that attempt UPDATE, DELETE, or SELECT FOR UPDATE of these rows will be blocked until the current transaction ends. Also, if an UPDATE, DELETE, or SELECT FOR UPDATE from another transaction has already locked a selected row or rows, SELECT FOR UPDATE will wait for the other transaction to complete, and will then lock and return the updated row (or no row, if the row was deleted).

FOR UPDATE cannot be used in contexts where returned rows can’t be clearly identified with individual table rows; for example it can't be used with aggregation.

**Examples**

To join table, `dept` with table, `emp`:

```
SELECT d.deptno, d.dname, e.empno, e.ename, e.mgr, e.hiredate
```
FROM emp e, dept d
WHERE d.deptno = e.deptno;

<table>
<thead>
<tr>
<th>deptno</th>
<th>dname</th>
<th>empno</th>
<th>ename</th>
<th>mgr</th>
<th>hiredate</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>ACCOUNTING</td>
<td>7934</td>
<td>MILLER</td>
<td>7782</td>
<td>23-JAN-82 00:00:00</td>
</tr>
<tr>
<td>10</td>
<td>ACCOUNTING</td>
<td>7782</td>
<td>CLARK</td>
<td>7839</td>
<td>09-JUN-81 00:00:00</td>
</tr>
<tr>
<td>10</td>
<td>ACCOUNTING</td>
<td>7839</td>
<td>KING</td>
<td></td>
<td>17-NOV-81 00:00:00</td>
</tr>
<tr>
<td>20</td>
<td>RESEARCH</td>
<td>7788</td>
<td>SCOTT</td>
<td>7566</td>
<td>19-APR-87 00:00:00</td>
</tr>
<tr>
<td>20</td>
<td>RESEARCH</td>
<td>7566</td>
<td>JONES</td>
<td>7839</td>
<td>02-APR-81 00:00:00</td>
</tr>
<tr>
<td>20</td>
<td>RESEARCH</td>
<td>7369</td>
<td>SMITH</td>
<td>7902</td>
<td>17-DEC-80 00:00:00</td>
</tr>
<tr>
<td>20</td>
<td>RESEARCH</td>
<td>7876</td>
<td>ADAMS</td>
<td>7788</td>
<td>23-MAY-87 00:00:00</td>
</tr>
<tr>
<td>20</td>
<td>RESEARCH</td>
<td>7902</td>
<td>FORD</td>
<td>7566</td>
<td>03-DEC-81 00:00:00</td>
</tr>
<tr>
<td>30</td>
<td>SALES</td>
<td>7521</td>
<td>WARD</td>
<td>7698</td>
<td>22-FEB-81 00:00:00</td>
</tr>
<tr>
<td>30</td>
<td>SALES</td>
<td>7844</td>
<td>TURNER</td>
<td>7698</td>
<td>08-SEP-81 00:00:00</td>
</tr>
<tr>
<td>30</td>
<td>SALES</td>
<td>7499</td>
<td>ALLEN</td>
<td>7698</td>
<td>20-FEB-81 00:00:00</td>
</tr>
<tr>
<td>30</td>
<td>SALES</td>
<td>7698</td>
<td>BLAKE</td>
<td>7839</td>
<td>01-MAY-81 00:00:00</td>
</tr>
<tr>
<td>30</td>
<td>SALES</td>
<td>7654</td>
<td>MARTIN</td>
<td>7698</td>
<td>28-SEP-81 00:00:00</td>
</tr>
<tr>
<td>30</td>
<td>SALES</td>
<td>7900</td>
<td>JAMES</td>
<td>7698</td>
<td>03-DEC-81 00:00:00</td>
</tr>
</tbody>
</table>

(14 rows)

To sum the column, sal of all employees and group the results by department number:

```
SELECT deptno, SUM(sal) AS total
FROM emp
GROUP BY deptno;
```

<table>
<thead>
<tr>
<th>deptno</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>8750.00</td>
</tr>
<tr>
<td>20</td>
<td>10875.00</td>
</tr>
<tr>
<td>30</td>
<td>9400.00</td>
</tr>
</tbody>
</table>

(3 rows)

To sum the column, sal of all employees, group the results by department number and show those group totals that are less than 10000:

```
SELECT deptno, SUM(sal) AS total
FROM emp
GROUP BY deptno
HAVING SUM(sal) < 10000;
```

<table>
<thead>
<tr>
<th>deptno</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>8750.00</td>
</tr>
<tr>
<td>30</td>
<td>9400.00</td>
</tr>
</tbody>
</table>

(2 rows)

The following two examples are identical ways of sorting the individual results according to the contents of the second column (dname):

```
SELECT * FROM dept ORDER BY dname;
```

<table>
<thead>
<tr>
<th>deptno</th>
<th>dname</th>
<th>loc</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>ACCOUNTING</td>
<td>NEW YORK</td>
</tr>
<tr>
<td>40</td>
<td>OPERATIONS</td>
<td>BOSTON</td>
</tr>
<tr>
<td>20</td>
<td>RESEARCH</td>
<td>DALLAS</td>
</tr>
<tr>
<td>30</td>
<td>SALES</td>
<td>CHICAGO</td>
</tr>
</tbody>
</table>
```sql
SELECT * FROM dept ORDER BY 2;
```

<table>
<thead>
<tr>
<th>deptno</th>
<th>dname</th>
<th>loc</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>ACCOUNTING</td>
<td>NEW YORK</td>
</tr>
<tr>
<td>40</td>
<td>OPERATIONS</td>
<td>BOSTON</td>
</tr>
<tr>
<td>20</td>
<td>RESEARCH</td>
<td>DALLAS</td>
</tr>
<tr>
<td>30</td>
<td>SALES</td>
<td>CHICAGO</td>
</tr>
</tbody>
</table>

(4 rows)
3.3.59 SET CONSTRAINTS

Name

SET CONSTRAINTS -- set constraint checking modes for the current transaction

Synopsis

SET CONSTRAINTS { ALL | name [, ...] } { DEFERRED | IMMEDIATE }

Description

SET CONSTRAINTS sets the behavior of constraint checking within the current transaction. IMMEDIATE constraints are checked at the end of each statement. DEFERRED constraints are not checked until transaction commit. Each constraint has its own IMMEDIATE or DEFERRED mode.

Upon creation, a constraint is given one of three characteristics: DEFERRABLE INITIALLY DEFERRED, DEFERRABLE INITIALLY IMMEDIATE, or NOT DEFERRABLE. The third class is always IMMEDIATE and is not affected by the SET CONSTRAINTS command. The first two classes start every transaction in the indicated mode, but their behavior can be changed within a transaction by SET CONSTRAINTS.

SET CONSTRAINTS with a list of constraint names changes the mode of just those constraints (which must all be deferrable). If there are multiple constraints matching any given name, all are affected. SET CONSTRAINTS ALL changes the mode of all deferrable constraints.

When SET CONSTRAINTS changes the mode of a constraint from DEFERRED to IMMEDIATE, the new mode takes effect retroactively: any outstanding data modifications that would have been checked at the end of the transaction are instead checked during the execution of the SET CONSTRAINTS command. If any such constraint is violated, the SET CONSTRAINTS fails (and does not change the constraint mode). Thus, SET CONSTRAINTS can be used to force checking of constraints to occur at a specific point in a transaction.

Currently, only foreign key constraints are affected by this setting. Check and unique constraints are always effectively not deferrable.

Notes
This command only alters the behavior of constraints within the current transaction. Thus, if you execute this command outside of a transaction block it will not appear to have any effect.

### 3.3.60 SET ROLE

#### Name

**SET ROLE** -- set the current user identifier of the current session

#### Synopsis

```sql
SET ROLE { rolename | NONE }
```

#### Description

This command sets the current user identifier of the current SQL session context to be *rolename*. After **SET ROLE**, permissions checking for SQL commands is carried out as though the named role were the one that had logged in originally.

The specified *rolename* must be a role that the current session user is a member of. (If the session user is a superuser, any role can be selected.)

**NONE** resets the current user identifier to be the current session user identifier. These forms may be executed by any user.

#### Notes

Using this command, it is possible to either add privileges or restrict one’s privileges. If the session user role has the **INHERITS** attribute, then it automatically has all the privileges of every role that it could **SET ROLE** to; in this case **SET ROLE** effectively drops all the privileges assigned directly to the session user and to the other roles it is a member of, leaving only the privileges available to the named role. On the other hand, if the session user role has the **NOINHERITS** attribute, **SET ROLE** drops the privileges assigned directly to the session user and instead acquires the privileges available to the named role. In particular, when a superuser chooses to **SET ROLE** to a non-superuser role, she loses her superuser privileges.

#### Examples

User *mary* takes on the identity of role *admins*:

```sql
SET ROLE admins;
```
User mary reverts back to her own identity:

```
SET ROLE NONE;
```
3.3.61 SET TRANSACTION

Name

SET TRANSACTION -- set the characteristics of the current transaction

Synopsis

SET TRANSACTION transaction_mode

where transaction_mode is one of:

   ISOLATION LEVEL { SERIALIZABLE | READ COMMITTED }
   READ WRITE | READ ONLY

Description

The SET TRANSACTION command sets the characteristics of the current transaction. It has no effect on any subsequent transactions. The available transaction characteristics are the transaction isolation level and the transaction access mode (read/write or read-only). The transaction isolation level of a transaction determines what data the transaction can see when other transactions are running concurrently:

READ COMMITTED

A statement can only see rows committed before it began. This is the default.

SERIALIZABLE

All statements of the current transaction can only see rows committed before the first query or data-modification statement was executed in this transaction.

The transaction isolation level cannot be changed after the first query or data-modification statement (SELECT, INSERT, DELETE, UPDATE, or FETCH) of a transaction has been executed. The transaction access mode determines whether the transaction is read/write or read-only. Read/write is the default.

When a transaction is read-only, the following SQL commands are disallowed: INSERT, UPDATE, and DELETE if the table they would write to is not a temporary table; all CREATE, ALTER, and DROP commands; COMMENT, GRANT, REVOKE, TRUNCATE; and EXECUTE if the command it would execute is among those listed. This is a high-level notion of read-only that does not prevent all writes to disk.
3.3.62 TRUNCATE

Name

TRUNCATE -- empty a table

Synopsis

TRUNCATE TABLE name

Description

TRUNCATE quickly removes all rows from a table. It has the same effect as an unqualified DELETE but since it does not actually scan the table, it is faster. This is most useful on large tables.

Parameters

name

The name (optionally schema-qualified) of the table to be truncated.

Notes

TRUNCATE cannot be used if there are foreign-key references to the table from other tables. Checking validity in such cases would require table scans, and the whole point is not to do one.

TRUNCATE will not run any user-defined ON DELETE triggers that might exist for the table.

Examples

Truncate the table bigtable:

```
TRUNCATE TABLE bigtable;
```

See Also

DROP VIEW, DELETE
3.3.63 UPDATE

Name

UPDATE -- update rows of a table

Synopsis

UPDATE [ optimizer_hint ] table[@dblink ]
   SET column = { expression | DEFAULT } [, ...]
   [ WHERE condition ]
   [ RETURNING return_expression [, ...]
      { INTO { record | variable [, ...] } }[
       | BULK COLLECT INTO collection [, ...] } ]

Description

UPDATE changes the values of the specified columns in all rows that satisfy the condition. Only the columns to be modified need be mentioned in the SET clause; columns not explicitly modified retain their previous values.

The RETURNING INTO { record | variable [, ...] } clause may only be specified within an SPL program. In addition the result set of the UPDATE command must not return more than one row, otherwise an exception is thrown. If the result set is empty, then the contents of the target record or variables are set to null.

The RETURNING BULK COLLECT INTO collection [, ...] clause may only be specified if the UPDATE command is used within an SPL program. If more than one collection is specified as the target of the BULK COLLECT INTO clause, then each collection must consist of a single, scalar field – i.e., collection must not be a record. The result set of the UPDATE command may contain none, one, or more rows. return_expression evaluated for each row of the result set, becomes an element in collection starting with the first element. Any existing rows in collection are deleted. If the result set is empty, then collection will be empty.

You must have the UPDATE privilege on the table to update it, as well as the SELECT privilege to any table whose values are read in expression or condition.

Parameters

optimizer_hint

Comment-embedded hints to the optimizer for selection of an execution plan. See Section 3.4 for information on optimizer hints.
table

The name (optionally schema-qualified) of the table to update.

dblink

Database link name identifying a remote database. See the CREATE DATABASE LINK for information on database links.

column

The name of a column in table.

eexpression

An expression to assign to the column. The expression may use the old values of this and other columns in the table.

DEFAULT

Set the column to its default value (which will be null if no specific default expression has been assigned to it).

condition

An expression that returns a value of type BOOLEAN. Only rows for which this expression returns true will be updated.

return_expression

An expression that may include one or more columns from table. If a column name from table is specified in return_expression, the value substituted for the column when return_expression is evaluated is determined as follows:

If the column specified in return_expression is assigned a value in the UPDATE command, then the assigned value is used in the evaluation of return_expression.

If the column specified in return_expression is not assigned a value in the UPDATE command, then the column’s current value in the affected row is used in the evaluation of return_expression.

record

A record whose field the evaluated return_expression is to be assigned. The first return_expression is assigned to the first field in record, the second
```
return_expression is assigned to the second field in record, etc. The number of fields in record must exactly match the number of expressions and the fields must be type-compatible with their assigned expressions.

variable

A variable to which the evaluated return_expression is to be assigned. If more than one return_expression and variable are specified, the first return_expression is assigned to the first variable, the second return_expression is assigned to the second variable, etc. The number of variables specified following the INTO keyword must exactly match the number of expressions following the RETURNING keyword and the variables must be type-compatible with their assigned expressions.

collection

A collection in which an element is created from the evaluated return_expression. There can be either a single collection which may be a collection of a single field or a collection of a record type, or there may be more than one collection in which case each collection must consist of a single field. The number of return expressions must match in number and order the number of fields in all specified collections. Each corresponding return_expression and collection field must be type-compatible.

Examples

Change the location to AUSTIN for department 20 in the dept table:

UPDATE dept SET loc = 'AUSTIN' WHERE deptno = 20;

For all employees with job = SALESMAN in the emp table, update the salary by 10% and increase the commission by 500.

UPDATE emp SET sal = sal * 1.1, comm = comm + 500 WHERE job = 'SALESMAN';
```
3.4 Optimizer 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 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 ] } */
[...]
"
```

```
{ DELETE | INSERT | SELECT | UPDATE } +{ hint [ comment ] }
[...]
"
```

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 `Postgres Plus` 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. See Section 3.1.2 for more information on identifiers. 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.
3.4.1 Default Optimization Modes

There are a number of optimization modes that can be chosen as the default setting for a Postgres Plus 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>
<thead>
<tr>
<th>Hint</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALL_ROWS</td>
<td>Optimizes for retrieval of all rows of the result set.</td>
</tr>
<tr>
<td>CHOOSE</td>
<td>Does no default optimization based on assumed number of rows to be retrieved from the result set. This is the default.</td>
</tr>
<tr>
<td>FIRST_ROWS</td>
<td>Optimizes for retrieval of only the first row of the result set.</td>
</tr>
<tr>
<td>FIRST_ROWS_10</td>
<td>Optimizes for retrieval of the first 10 rows of the result set.</td>
</tr>
<tr>
<td>FIRST_ROWS_100</td>
<td>Optimizes for retrieval of the first 100 rows of the result set.</td>
</tr>
<tr>
<td>FIRST_ROWS_1000</td>
<td>Optimizes for retrieval of the first 1000 rows of the result set.</td>
</tr>
<tr>
<td>FIRST_ROWS(n)</td>
<td>Optimizes for retrieval of the first (n) rows of the result set.</td>
</tr>
<tr>
<td></td>
<td>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.</td>
</tr>
</tbody>
</table>

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 Oracle compatible SHOW command has the following syntax:
SHOW PARAMETER OPTIMIZER_MODE;

<table>
<thead>
<tr>
<th>NAME</th>
<th>VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>optimizer_mode</td>
<td>first_rows_10</td>
</tr>
</tbody>
</table>

The following example shows an optimization mode used in a `SELECT` command as a hint:

```
SELECT /*+ FIRST_ROWS(7) */ * FROM emp;
```

<table>
<thead>
<tr>
<th>empno</th>
<th>ename</th>
<th>job</th>
<th>mgr</th>
<th>hiredate</th>
<th>sal</th>
<th>comm</th>
<th>deptno</th>
</tr>
</thead>
<tbody>
<tr>
<td>7369</td>
<td>SMITH</td>
<td>CLERK</td>
<td>7902</td>
<td>17-DEC-80</td>
<td>800.00</td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>7499</td>
<td>ALLEN</td>
<td>SALESMAN</td>
<td>7698</td>
<td>20-FEB-81</td>
<td>1600.00</td>
<td>300.00</td>
<td>30</td>
</tr>
<tr>
<td>7521</td>
<td>WARD</td>
<td>SALESMAN</td>
<td>7698</td>
<td>22-FEB-81</td>
<td>1250.00</td>
<td>500.00</td>
<td>30</td>
</tr>
<tr>
<td>7566</td>
<td>JONES</td>
<td>MANAGER</td>
<td>7839</td>
<td>02-APR-81</td>
<td>2975.00</td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>7654</td>
<td>MARTIN</td>
<td>SALESMAN</td>
<td>7698</td>
<td>28-SEP-81</td>
<td>1250.00</td>
<td>1400.00</td>
<td>30</td>
</tr>
<tr>
<td>7698</td>
<td>BLAKE</td>
<td>MANAGER</td>
<td>7839</td>
<td>01-MAY-81</td>
<td>2850.00</td>
<td></td>
<td>30</td>
</tr>
<tr>
<td>7782</td>
<td>CLARK</td>
<td>MANAGER</td>
<td>7839</td>
<td>09-JUN-81</td>
<td>2450.00</td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>7788</td>
<td>SCOTT</td>
<td>ANALYST</td>
<td>7566</td>
<td>19-APR-87</td>
<td>3000.00</td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>7839</td>
<td>KING</td>
<td>PRESIDENT</td>
<td>7839</td>
<td>17-NOV-81</td>
<td>5000.00</td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>7844</td>
<td>TURNER</td>
<td>SALESMAN</td>
<td>7698</td>
<td>08-SEP-81</td>
<td>1500.00</td>
<td>0.00</td>
<td>30</td>
</tr>
<tr>
<td>7876</td>
<td>ADAMS</td>
<td>CLERK</td>
<td>7782</td>
<td>23-MAY-87</td>
<td>1100.00</td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>7900</td>
<td>JAMES</td>
<td>CLERK</td>
<td>7698</td>
<td>03-DEC-81</td>
<td>950.00</td>
<td></td>
<td>30</td>
</tr>
<tr>
<td>7902</td>
<td>FORD</td>
<td>ANALYST</td>
<td>7566</td>
<td>03-DEC-81</td>
<td>3000.00</td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>7934</td>
<td>MILLER</td>
<td>CLERK</td>
<td>7782</td>
<td>23-JAN-82</td>
<td>1300.00</td>
<td></td>
<td>10</td>
</tr>
</tbody>
</table>
```
3.4.2 Access Method Hints

The following hints influence how the optimizer accesses relations to create the result set.

<table>
<thead>
<tr>
<th>Hint</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FULL(table)</td>
<td>Perform a full sequential scan on table.</td>
</tr>
<tr>
<td>INDEX(table [ index ] [...])</td>
<td>Use index on table to access the relation.</td>
</tr>
<tr>
<td>NO_INDEX(table [ index ] [...])</td>
<td>Do not use index on table to access the relation.</td>
</tr>
</tbody>
</table>

In addition, the ALL_ROWS, FIRST_ROWS, and FIRST_ROWS(n) hints of Table 3-3-10 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/9.2AS/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.

```bash
export PATH=/opt/PostgresPlus/9.2AS/bin:$PATH
```

The following example was run in Windows.

```bash
SET PATH=C:\PostgresPlus\9.2AS\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.
```
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.

```
490000 tuples done.
500000 tuples done.
set primary key...
vacuum...done.
```

```
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.branche
s"
  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"
```
<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Modifiers</th>
</tr>
</thead>
<tbody>
<tr>
<td>tid</td>
<td>integer</td>
<td></td>
</tr>
<tr>
<td>bid</td>
<td>integer</td>
<td></td>
</tr>
<tr>
<td>aid</td>
<td>integer</td>
<td></td>
</tr>
<tr>
<td>delta</td>
<td>integer</td>
<td></td>
</tr>
<tr>
<td>mtime</td>
<td>timestamp without time zone</td>
<td></td>
</tr>
<tr>
<td>filler</td>
<td>character(22)</td>
<td></td>
</tr>
</tbody>
</table>

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.

```sql
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:

```sql
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:

```sql
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:

```sql
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.

```sql
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.

```sql
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)
```
3.4.3 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 Oracle-style outer joins (those joins that contain a '+' sign).
3.4.4 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>
<thead>
<tr>
<th>Hint</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>USE_HASH(table [...])</td>
<td>Use a hash join with a hash table created from the join attributes of table.</td>
</tr>
<tr>
<td>NO_USE_HASH(table [...])</td>
<td>Do not use a hash join created from the join attributes of table.</td>
</tr>
<tr>
<td>USE_MERGE(table [...])</td>
<td>Use a merge sort join for table.</td>
</tr>
<tr>
<td>NO_USE_MERGE(table [...])</td>
<td>Do not use a merge sort join for table.</td>
</tr>
<tr>
<td>USE_NL(table [...])</td>
<td>Use a nested loop join for table.</td>
</tr>
<tr>
<td>NO_USE_NL(table [...])</td>
<td>Do not use a nested loop join for table.</td>
</tr>
</tbody>
</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.

```sql
EXPLAIN SELECT b.bid, a.aid, abalance FROM branches b, accounts a WHERE b.bid = a.bid;
```

```
<table>
<thead>
<tr>
<th>QUERY PLAN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hash Join  (cost=1.11..20092.70 rows=500488 width=12)</td>
</tr>
<tr>
<td>Hash Cond: (a.bid = b.bid)</td>
</tr>
<tr>
<td>-&gt; Seq Scan on accounts a (cost=0.00..13209.88 rows=500488 width=12)</td>
</tr>
<tr>
<td>-&gt; Hash (cost=1.05..1.05 rows=5 width=4)</td>
</tr>
<tr>
<td>-&gt; Seq Scan on branches b (cost=0.00..1.05 rows=5 width=4)</td>
</tr>
</tbody>
</table>
```

(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.

```sql
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.

```sql
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.

```sql
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.
This plan is altered by using hints to force a combination of a merge sort join and a hash join.

EXPLAIN SELECT /*+ USE_ME */ 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;

---

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)
3.4.5 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 3-3-11 or Table 3-3-12.

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:

```
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)
```
The same hints that were applied to this join at the end of Section 3.4.3 can be applied to the view as follows:

```
EXPLAIN SELECT /*+ USE_MERGE(tx.h tx.b) USE_HASH(tx.a) */ * FROM tx;
```

```
<table>
<thead>
<tr>
<th>empno</th>
<th>ename</th>
<th>mgr empno</th>
<th>mgr ename</th>
</tr>
</thead>
<tbody>
<tr>
<td>7902</td>
<td>FORD</td>
<td>7566</td>
<td>JONES</td>
</tr>
<tr>
<td>7788</td>
<td>SCOTT</td>
<td>7566</td>
<td>JONES</td>
</tr>
<tr>
<td>7521</td>
<td>WARD</td>
<td>7698</td>
<td>BLAKE</td>
</tr>
<tr>
<td>7844</td>
<td>TURNER</td>
<td>7698</td>
<td>BLAKE</td>
</tr>
<tr>
<td>7654</td>
<td>MARTIN</td>
<td>7698</td>
<td>BLAKE</td>
</tr>
<tr>
<td>7900</td>
<td>JAMES</td>
<td>7698</td>
<td>BLAKE</td>
</tr>
<tr>
<td>7499</td>
<td>ALLEN</td>
<td>7698</td>
<td>BLAKE</td>
</tr>
<tr>
<td>7934</td>
<td>MILLER</td>
<td>7782</td>
<td>CLARK</td>
</tr>
<tr>
<td>7876</td>
<td>ADAMS</td>
<td>7788</td>
<td>SCOTT</td>
</tr>
<tr>
<td>7782</td>
<td>CLARK</td>
<td>7839</td>
<td>KING</td>
</tr>
<tr>
<td>7698</td>
<td>BLAKE</td>
<td>7839</td>
<td>KING</td>
</tr>
<tr>
<td>7566</td>
<td>JONES</td>
<td>7839</td>
<td>KING</td>
</tr>
<tr>
<td>7369</td>
<td>SMITH</td>
<td>7902</td>
<td>FORD</td>
</tr>
</tbody>
</table>
```

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;
```

```
The plan chosen by the query planner is shown below:
```

```
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
```

Copyright © 2007 - 2015 EnterpriseDB Corporation. All rights reserved.
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)
3.4.6 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 Oracle-compatible command instructs the server to append the data in the INSERT statement to the end of the sales table:

```sql
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:

```sql
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:

```sql
INSERT INTO sales_history SELECT /*+APPEND*/ FROM sales;
```
3.4.7 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>
<thead>
<tr>
<th>Hint</th>
<th>Conflicting Hint</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALL_ROWS</td>
<td>FIRST_ROWS - all formats</td>
</tr>
<tr>
<td>FULL(table)</td>
<td>INDEX(table [ index ])</td>
</tr>
<tr>
<td>INDEX(table)</td>
<td>FULL(table)</td>
</tr>
<tr>
<td></td>
<td>NO_INDEX(table)</td>
</tr>
<tr>
<td>INDEX(table index)</td>
<td>FULL(table)</td>
</tr>
<tr>
<td></td>
<td>NO_INDEX(table index)</td>
</tr>
<tr>
<td>USE_HASH(table)</td>
<td>NO_USE_HASH(table)</td>
</tr>
<tr>
<td>USE_MERGE(table)</td>
<td>NO_USE_MERGE(table)</td>
</tr>
<tr>
<td>USE_NL(table)</td>
<td>NO_USE_NL(table)</td>
</tr>
</tbody>
</table>
3.5 Functions and Operators

Postgres Plus Advanced Server provides a large number of functions and operators for the built-in data types.

3.5.1 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 3-3-14 AND/OR Truth Table

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>a AND b</td>
<td>a OR b</td>
</tr>
<tr>
<td>True</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>Null</td>
<td>Null</td>
<td>True</td>
</tr>
<tr>
<td>False</td>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>Null</td>
<td>False</td>
<td>Null</td>
</tr>
<tr>
<td>Null</td>
<td>Null</td>
<td>Null</td>
<td>Null</td>
</tr>
</tbody>
</table>

Table 3-3-15 NOT Truth Table

<table>
<thead>
<tr>
<th></th>
<th>NOT a</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
</tr>
<tr>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>Null</td>
<td>Null</td>
</tr>
</tbody>
</table>

The operators AND and OR are commutative, that is, you can switch the left and right operand without affecting the result.
3.5.2 Comparison Operators

The usual comparison operators are shown in the following table.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;</td>
<td>Less than</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or equal to</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than or equal to</td>
</tr>
<tr>
<td>=</td>
<td>Equal</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>Not equal</td>
</tr>
<tr>
<td>!=</td>
<td>Not equal</td>
</tr>
</tbody>
</table>

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.
3.5.3 Mathematical Functions and Operators

Mathematical operators are provided for many Postgres Plus 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 3-3-17 Mathematical Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Addition</td>
<td>2 + 3</td>
<td>5</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction</td>
<td>2 - 3</td>
<td>-1</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication</td>
<td>2 * 3</td>
<td>6</td>
</tr>
<tr>
<td>/</td>
<td>Division (integer division truncates results)</td>
<td>4 / 2</td>
<td>2</td>
</tr>
<tr>
<td>**</td>
<td>Exponentiation Operator</td>
<td>2 ** 3</td>
<td>8</td>
</tr>
</tbody>
</table>

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 3-3-18 Mathematical Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Return Type</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABS(x)</td>
<td>Same as x</td>
<td>Absolute value</td>
<td>ABS(-17.4)</td>
<td>17.4</td>
</tr>
<tr>
<td>CEIL(DOUBLE PRECISION or NUMBER)</td>
<td>Same as input</td>
<td>Smallest integer not less than argument</td>
<td>CEIL(-42.8)</td>
<td>-42</td>
</tr>
<tr>
<td>EXP(DOUBLE PRECISION or NUMBER)</td>
<td>Same as input</td>
<td>Exponential</td>
<td>EXP(1.0)</td>
<td>2.7182818284590452</td>
</tr>
<tr>
<td>FLOOR(DOUBLE PRECISION or NUMBER)</td>
<td>Same as input</td>
<td>Largest integer not greater than argument</td>
<td>FLOOR(-42.8)</td>
<td>43</td>
</tr>
<tr>
<td>LN(DOUBLE PRECISION or NUMBER)</td>
<td>Same as input</td>
<td>Natural logarithm</td>
<td>LN(2.0)</td>
<td>0.6931471805599453</td>
</tr>
<tr>
<td>LOG(b NUMBER, x NUMBER)</td>
<td>NUMBER</td>
<td>Logarithm to base b</td>
<td>LOG(2.0, 64.0)</td>
<td>6.000000000000000000</td>
</tr>
<tr>
<td>MOD(y, x)</td>
<td>Same as argument types</td>
<td>Remainder of y/x</td>
<td>MOD(9, 4)</td>
<td>1</td>
</tr>
<tr>
<td>NVL(x, y)</td>
<td>Same as argument types; where both arguments are of the same data type</td>
<td>If x is null, then NVL returns y</td>
<td>NVL(9, 0)</td>
<td>9</td>
</tr>
<tr>
<td>POWER(a DOUBLE PRECISION, b DOUBLE PRECISION)</td>
<td>DOUBLE PRECISION</td>
<td>a raised to the power of b</td>
<td>POWER(9.0, 3.0)</td>
<td>729.000000000000000000</td>
</tr>
</tbody>
</table>
### Database Compatibility for Oracle® Developer’s Guide

<table>
<thead>
<tr>
<th>Function</th>
<th>Return Type</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>POWER(a NUMBER, b NUMBER)</td>
<td>NUMBER</td>
<td>a raised to the power of b</td>
<td>POWER(9.0, 3.0)</td>
<td>729.0000000000000000</td>
</tr>
<tr>
<td>ROUND(DOUBLE PRECISION or NUMBER)</td>
<td>Same as input</td>
<td>Round to nearest integer</td>
<td>ROUND(42.4)</td>
<td>42</td>
</tr>
<tr>
<td>ROUND(v NUMBER, s INTEGER)</td>
<td>NUMBER</td>
<td>Round to s decimal places</td>
<td>ROUND(42.4382, 2)</td>
<td>42.44</td>
</tr>
<tr>
<td>SIGN(DOUBLE PRECISION or NUMBER)</td>
<td>Same as input</td>
<td>Sign of the argument (-1, 0, +1)</td>
<td>SIGN(-8.4)</td>
<td>-1</td>
</tr>
<tr>
<td>SQRT(DOUBLE PRECISION or NUMBER)</td>
<td>Same as input</td>
<td>Square root</td>
<td>SQRT(2.0)</td>
<td>1.414213562373095</td>
</tr>
<tr>
<td>TRUNC(DOUBLE PRECISION or NUMBER)</td>
<td>Same as input</td>
<td>Truncate toward zero</td>
<td>TRUNC(42.8)</td>
<td>42</td>
</tr>
<tr>
<td>TRUNC(v NUMBER, s INTEGER)</td>
<td>NUMBER</td>
<td>Truncate to s decimal places</td>
<td>TRUNC(42.4382, 2)</td>
<td>42.43</td>
</tr>
<tr>
<td>WIDTH_BUCKET(op NUMBER, b1 NUMBER, b2 NUMBER, count INTEGER)</td>
<td>INTEGER</td>
<td>Return the bucket to which op would be assigned in an equidepth histogram with count buckets, in the range b1 to b2</td>
<td>WIDTH_BUCKET(5.35, 0.024, 10.06, 5)</td>
<td>3</td>
</tr>
</tbody>
</table>

The following table shows the available trigonometric functions. All trigonometric functions take arguments and return values of type DOUBLE PRECISION.

### Table 3-3-19 Trigonometric Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACOS(x)</td>
<td>Inverse cosine</td>
</tr>
<tr>
<td>ASIN(x)</td>
<td>Inverse sine</td>
</tr>
<tr>
<td>ATAN(x)</td>
<td>Inverse tangent</td>
</tr>
<tr>
<td>ATAN2(x, y)</td>
<td>Inverse tangent of x/y</td>
</tr>
<tr>
<td>COS(x)</td>
<td>Cosine</td>
</tr>
<tr>
<td>SIN(x)</td>
<td>Sine</td>
</tr>
<tr>
<td>TAN(x)</td>
<td>Tangent</td>
</tr>
</tbody>
</table>
3.5.4 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 3-3-20 SQL String Functions and Operators

<table>
<thead>
<tr>
<th>Function</th>
<th>Return Type</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td></td>
<td>string</td>
<td>CLOB</td>
<td>String concatenation</td>
</tr>
<tr>
<td>CONCAT(string, string)</td>
<td>CLOB</td>
<td>String concatenation</td>
<td>'a'</td>
<td></td>
</tr>
<tr>
<td>HEXTORAW(varchar2)</td>
<td>RAW</td>
<td>Converts a VARCHAR2 value to a RAW value</td>
<td>HEXTORAW('303132')</td>
<td>'012'</td>
</tr>
<tr>
<td>RAWTOHEX(raw)</td>
<td>VARCHAR2</td>
<td>Converts a RAW value to a HEXADECIMAL value</td>
<td>RAWTOHEX('012')</td>
<td>'303132'</td>
</tr>
<tr>
<td>INSTR(string, set, [ start [, occurrence ] ] )</td>
<td>INTEGER</td>
<td>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.</td>
<td>INSTR('PETER PIPER PICKED a PECK of PICKLED PEPPERS','PI',1,3)</td>
<td>30</td>
</tr>
<tr>
<td>INSTRB(string, set)</td>
<td>INTEGER</td>
<td>Returns the position of the set within the string. Returns 0 if set is not found.</td>
<td>INSTRB('PETER PIPER PICKED a PECK of PICKLED PEPPERS','PI')</td>
<td>13</td>
</tr>
<tr>
<td>INSTRB(string, set, start)</td>
<td>INTEGER</td>
<td>Returns the position of the set within the string, beginning at start. Returns 0 if set is not found.</td>
<td>INSTRB('PETER PIPER PICKED a PECK of PICKLED PEPPERS','PI',14)</td>
<td>30</td>
</tr>
<tr>
<td>INSTRB(string, set, start, occurrence)</td>
<td>INTEGER</td>
<td>Returns the position of the specified occurrence of set within the string, beginning at start. Returns 0 if set is not found.</td>
<td>INSTRB('PETER PIPER PICKED a PECK of PICKLED PEPPERS','PI',1,2)</td>
<td>30</td>
</tr>
<tr>
<td>LOWER(string)</td>
<td>CLOB</td>
<td>Convert string to lower case</td>
<td>LOWER('TOM')</td>
<td>tom</td>
</tr>
<tr>
<td>SUBSTR(string, start [, count ] )</td>
<td>CLOB</td>
<td>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.</td>
<td>SUBSTR('This is a test',6,2)</td>
<td>is</td>
</tr>
<tr>
<td>SUBSTRB(string, start [, count ] )</td>
<td>CLOB</td>
<td>Same as SUBSTR except start and count are in RAW.</td>
<td>SUBSTRB('abc',3,3) (assuming a double-byte)</td>
<td>c</td>
</tr>
<tr>
<td>Function</td>
<td>Return Type</td>
<td>Description</td>
<td>Example</td>
<td>Result</td>
</tr>
<tr>
<td>------------------------------</td>
<td>-------------</td>
<td>-------------------------------------------------------</td>
<td>--------------------------</td>
<td>--------</td>
</tr>
<tr>
<td>SUBSTR2(string, start [, count ])</td>
<td>CLOB</td>
<td>Accesses a substring of the given string, represented as CLOB.</td>
<td>SUBSTR2(‘This is a test’,6,2)</td>
<td>is</td>
</tr>
<tr>
<td>SUBSTR4(string, start [, count ])</td>
<td>CLOB</td>
<td>Accesses a substring of the given string, represented as CLOB.</td>
<td>SUBSTR4(‘abc’,3)</td>
<td>c</td>
</tr>
<tr>
<td>SUBSTRC(string, start [, count ])</td>
<td>CLOB</td>
<td>Accesses a substring of the given string, represented as CLOB.</td>
<td>SUBSTRC(‘This is a test’,6,2)</td>
<td>is</td>
</tr>
<tr>
<td>TRIM([ LEADING</td>
<td>TRAILING</td>
<td>BOTH ] [ characters ] FROM string)</td>
<td>CLOB</td>
<td>Removes the longest string consisting only the characters (a space by default) from the start/end/both ends of the string.</td>
</tr>
<tr>
<td>LTRIM(string [, set])</td>
<td>CLOB</td>
<td>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.</td>
<td>LTRIM('abcdefg', 'abc')</td>
<td>defghi</td>
</tr>
<tr>
<td>RTRIM(string [, set])</td>
<td>CLOB</td>
<td>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.</td>
<td>RTRIM('abcdefg', 'ghi')</td>
<td>abcdef</td>
</tr>
<tr>
<td>UPPER(string)</td>
<td>CLOB</td>
<td>Converts the string to uppercase.</td>
<td>UPPER('tom')</td>
<td>TOM</td>
</tr>
</tbody>
</table>

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 3-3-20.

Table 3-3-21 Other String Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Return Type</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASCII(string)</td>
<td>INTEGER</td>
<td>ASCII code of the first byte of the argument</td>
<td>ASCII('x')</td>
<td>120</td>
</tr>
<tr>
<td>CHR(INTEGER)</td>
<td>CLOB</td>
<td>Character with the given ASCII code</td>
<td>CHR(65)</td>
<td>A</td>
</tr>
<tr>
<td>DECODE(expr, expr1a, expr1b [, expr2a, expr2b]...) [, default ]</td>
<td>Same as argument types of expr1b, expr2b, ..., default</td>
<td>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</td>
<td>DECODE(3, 1,'One', 2,'Two', 3,'Three', 'Not found')</td>
<td>Three</td>
</tr>
<tr>
<td>Function</td>
<td>Return Type</td>
<td>Description</td>
<td>Example</td>
<td>Result</td>
</tr>
<tr>
<td>--------------</td>
<td>-------------</td>
<td>------------------------------------------------------------------------------</td>
<td>-----------------------</td>
<td>------------</td>
</tr>
<tr>
<td>INITCAP(string)</td>
<td>CLOB</td>
<td>Converts the first letter of each word to uppercase and the rest to lowercase. Words are sequences of alphanumeric characters separated by non-alphanumeric characters.</td>
<td>INITCAP('hi THOMAS')</td>
<td>Hi Thomas</td>
</tr>
<tr>
<td>LENGTH</td>
<td>INTEGER</td>
<td>Returns the number of characters in a string value.</td>
<td>LENGTH('Côte d''Azur')</td>
<td>11</td>
</tr>
<tr>
<td>LENGTHC</td>
<td>INTEGER</td>
<td>This function is identical in functionality to LENGTH; the function name is supported for compatibility.</td>
<td>LENGTHC('Côte d''Azur')</td>
<td>11</td>
</tr>
<tr>
<td>LENGTH2</td>
<td>INTEGER</td>
<td>This function is identical in functionality to LENGTH; the function name is supported for compatibility.</td>
<td>LENGTH2('Côte d''Azur')</td>
<td>11</td>
</tr>
<tr>
<td>LENGTH4</td>
<td>INTEGER</td>
<td>This function is identical in functionality to LENGTH; the function name is supported for compatibility.</td>
<td>LENGTH4('Côte d''Azur')</td>
<td>11</td>
</tr>
<tr>
<td>LENGTHB</td>
<td>INTEGER</td>
<td>Returns the number of bytes required to hold the given value.</td>
<td>LENGTHB('Côte d''Azur')</td>
<td>12</td>
</tr>
<tr>
<td>LPAD(string, length INTEGER [, fill ])</td>
<td>CLOB</td>
<td>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).</td>
<td>LPAD('hi', 5, 'xy')</td>
<td>xxyhi</td>
</tr>
<tr>
<td>REPLACE(string, search_string [, replace_string ]</td>
<td>CLOB</td>
<td>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.</td>
<td>REPLACE('GEORGE', 'GE', 'EG')</td>
<td>EGOREG</td>
</tr>
<tr>
<td>RPAD(string, length INTEGER [, fill ])</td>
<td>CLOB</td>
<td>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.</td>
<td>RPAD('hi', 5, 'xy')</td>
<td>hixyx</td>
</tr>
<tr>
<td>TRANSLATE(string, from, to)</td>
<td>CLOB</td>
<td>Any character in string that matches a character in the from set is replaced by the corresponding character in the to set.</td>
<td>TRANSLATE('12345', '14', 'ax')</td>
<td>a23x5</td>
</tr>
</tbody>
</table>
3.5.5 Pattern Matching Using the LIKE Operator

Postgres Plus Advanced Server provides pattern matching using the traditional SQL LIKE operator. The syntax for the LIKE operator is as follows.

\[
\text{string LIKE pattern [ ESCAPE escape-character ]}
\]

\[
\text{string NOT LIKE pattern [ ESCAPE escape-character ]}
\]

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:

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>'abc' LIKE 'abc'</td>
<td>true</td>
</tr>
<tr>
<td>'abc' LIKE 'a%'</td>
<td>true</td>
</tr>
<tr>
<td>'abc' LIKE '<em>b</em>'</td>
<td>true</td>
</tr>
<tr>
<td>'abc' LIKE 'c'</td>
<td>false</td>
</tr>
</tbody>
</table>

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.
### 3.5.6 Data Type Formatting Functions

The Postgres Plus Advanced Server formatting functions (described in Table 3-3-22) 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 3-3-22 Formatting Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Return Type</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>TO_CHAR(DATE [,, format ])</td>
<td>VARCHAR2</td>
<td>Convert a date/time to a string with output, format. If omitted default format is DD-MON-YY.</td>
<td>TO_CHAR(SYSDATE, 'MM/DD/YYYY HH24:MI:SS AM')</td>
<td>07/25/2007 09:43:02 AM</td>
</tr>
<tr>
<td>TO_CHAR(INTEGER [,, format ])</td>
<td>VARCHAR2</td>
<td>Convert an integer to a string with output, format</td>
<td>TO_CHAR(2412, '999,999S')</td>
<td>2,412+</td>
</tr>
<tr>
<td>TO_CHAR(NUMBER [,, format ])</td>
<td>VARCHAR2</td>
<td>Convert a decimal number to a string with output, format</td>
<td>TO_CHAR(10125.35, '999,999.99')</td>
<td>10,125.35</td>
</tr>
<tr>
<td>TO_CHAR(DOUBLE PRECISION, format)</td>
<td>VARCHAR2</td>
<td>Convert a floating-point number to a string with output, format</td>
<td>TO_CHAR(CAST(123.5282 AS REAL), '999.99')</td>
<td>123.53</td>
</tr>
<tr>
<td>TO_DATE(string [,, format ])</td>
<td>DATE</td>
<td>Convert a date formatted string to a DATE data type</td>
<td>TO_DATE('2007-07-04 13:39:10', 'YYYY-MM-DD HH24:MI:SS')</td>
<td>04-JUL-07 13:39:10</td>
</tr>
<tr>
<td>TO_NUMBER(string [,, format ])</td>
<td>NUMBER</td>
<td>Convert a number formatted string to a NUMBER data type</td>
<td>TO_NUMBER('2,412-', '999,999S')</td>
<td>-2412</td>
</tr>
<tr>
<td>TO_TIMESTAMP(string, format)</td>
<td>TIMESTAMP</td>
<td>Convert a timestamp formatted string to a TIMESTAMP data type</td>
<td>TO_TIMESTAMP('05 Dec 2000 08:30:25 pm', 'DD Mon YYYY HH12:mi:ss pm')</td>
<td>05-DEC-00 20:30:25</td>
</tr>
</tbody>
</table>

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 3-3-23 Template Date/Time Format Patterns

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HH</td>
<td>Hour of day (01-12)</td>
</tr>
<tr>
<td>HH12</td>
<td>Hour of day (01-12)</td>
</tr>
<tr>
<td>HH24</td>
<td>Hour of day (00-23)</td>
</tr>
<tr>
<td>MI</td>
<td>Minute (00-59)</td>
</tr>
<tr>
<td>SS</td>
<td>Second (00-59)</td>
</tr>
<tr>
<td>SSSSSS</td>
<td>Seconds past midnight (0-86399)</td>
</tr>
<tr>
<td>AM or A.M. or PM or P.M.</td>
<td>Meridian indicator (uppercase)</td>
</tr>
<tr>
<td>am or a.m. or pm or p.m.</td>
<td>Meridian indicator (lowercase)</td>
</tr>
<tr>
<td>Y, YYYY</td>
<td>Year (4 and more digits) with comma</td>
</tr>
<tr>
<td>YEAR</td>
<td>Year (spelled out)</td>
</tr>
<tr>
<td>SYEAR</td>
<td>Year (spelled out) (BC dates prefixed by a minus sign)</td>
</tr>
<tr>
<td>YYYY</td>
<td>Year (4 and more digits)</td>
</tr>
<tr>
<td>SYYYYY</td>
<td>Year (4 and more digits) (BC dates prefixed by a minus sign)</td>
</tr>
<tr>
<td>YYY</td>
<td>Last 3 digits of year</td>
</tr>
<tr>
<td>YY</td>
<td>Last 2 digits of year</td>
</tr>
<tr>
<td>Y</td>
<td>Last digit of year</td>
</tr>
<tr>
<td>IYYYY</td>
<td>ISO year (4 and more digits)</td>
</tr>
<tr>
<td>IYY</td>
<td>Last 3 digits of ISO year</td>
</tr>
<tr>
<td>IY</td>
<td>Last 2 digits of ISO year</td>
</tr>
<tr>
<td>I</td>
<td>Last 1 digit of ISO year</td>
</tr>
<tr>
<td>BC or B.C. or AD or A.D.</td>
<td>Era indicator (uppercase)</td>
</tr>
<tr>
<td>bc or b.c. or ad or a.d.</td>
<td>Era indicator (lowercase)</td>
</tr>
<tr>
<td>MONTH</td>
<td>Full uppercase month name</td>
</tr>
<tr>
<td>Month</td>
<td>Full mixed-case month name</td>
</tr>
<tr>
<td>month</td>
<td>Full lowercase month name</td>
</tr>
<tr>
<td>MON</td>
<td>Abbreviated uppercase month name (3 chars in English, localized lengths vary)</td>
</tr>
<tr>
<td>Mon</td>
<td>Abbreviated mixed-case month name (3 chars in English, localized lengths vary)</td>
</tr>
<tr>
<td>mon</td>
<td>Abbreviated lowercase month name (3 chars in English, localized lengths vary)</td>
</tr>
<tr>
<td>MM</td>
<td>Month number (01-12)</td>
</tr>
<tr>
<td>DAY</td>
<td>Full uppercase day name</td>
</tr>
<tr>
<td>Day</td>
<td>Full mixed-case day name</td>
</tr>
<tr>
<td>day</td>
<td>Full lowercase day name</td>
</tr>
<tr>
<td>DY</td>
<td>Abbreviated uppercase day name (3 chars in English, localized lengths vary)</td>
</tr>
<tr>
<td>Dy</td>
<td>Abbreviated mixed-case day name (3 chars in English, localized lengths vary)</td>
</tr>
<tr>
<td>dy</td>
<td>Abbreviated lowercase day name (3 chars in English, localized lengths vary)</td>
</tr>
<tr>
<td>DDD</td>
<td>Day of year (001-366)</td>
</tr>
<tr>
<td>DD</td>
<td>Day of month (01-31)</td>
</tr>
<tr>
<td>D</td>
<td>Day of week (1-7; Sunday is 1)</td>
</tr>
<tr>
<td>W</td>
<td>Week of month (1-5) (The first week starts on the first day of the month)</td>
</tr>
<tr>
<td>WW</td>
<td>Week number of year (1-53) (The first week starts on the first day of the year)</td>
</tr>
</tbody>
</table>
Database Compatibility for Oracle® Developer’s Guide

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IW</td>
<td>ISO week number of year; the first Thursday of the new year is in week 1</td>
</tr>
<tr>
<td>CC</td>
<td>Century (2 digits); the 21st century starts on 2001-01-01</td>
</tr>
<tr>
<td>SCC</td>
<td>Same as CC except BC dates are prefixed by a minus sign</td>
</tr>
<tr>
<td>J</td>
<td>Julian Day (days since January 1, 4712 BC)</td>
</tr>
<tr>
<td>Q</td>
<td>Quarter</td>
</tr>
<tr>
<td>RM</td>
<td>Month in Roman numerals (I-XII; I=January) (uppercase)</td>
</tr>
<tr>
<td>rm</td>
<td>Month in Roman numerals (i-xii; I=January) (lowercase)</td>
</tr>
</tbody>
</table>
| 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 3-3-24 Template Pattern Modifiers for Date/Time Formatting**

<table>
<thead>
<tr>
<th>Modifier</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>FM prefix</td>
<td>Fill mode (suppress padding blanks and zeros)</td>
<td>FMMonth</td>
</tr>
<tr>
<td>TH suffix</td>
<td>Uppercase ordinal number suffix</td>
<td>DDTH</td>
</tr>
<tr>
<td>th suffix</td>
<td>Lowercase ordinal number suffix</td>
<td>DDth</td>
</tr>
<tr>
<td>FX prefix</td>
<td>Fixed format global option (see usage notes)</td>
<td>FX Month DD Day</td>
</tr>
<tr>
<td>SP suffix</td>
<td>Spell mode</td>
<td>DDSP</td>
</tr>
</tbody>
</table>

Usage notes for date/time formatting:

- FM suppresses leading zeroes and trailing blanks that would otherwise be added to 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, YYYY field. If CC is used with YY or Y then the year is computed as (CC-1)*100+YY.

Copyright © 2007 - 2015 EnterpriseDB Corporation. All rights reserved. 253
The following table shows the template patterns available for formatting numeric values.

### Table 3-3-25 Template Patterns for Numeric Formatting

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>Value with the specified number of digits</td>
</tr>
<tr>
<td>0</td>
<td>Value with leading zeroes</td>
</tr>
<tr>
<td>.</td>
<td>Decimal point</td>
</tr>
<tr>
<td>,</td>
<td>Group (thousand) separator</td>
</tr>
<tr>
<td>$</td>
<td>Dollar sign</td>
</tr>
<tr>
<td>&lt;PR&gt;</td>
<td>Negative value in angle brackets</td>
</tr>
<tr>
<td>S</td>
<td>Sign anchored to number (uses locale)</td>
</tr>
<tr>
<td>L</td>
<td>Currency symbol (uses locale)</td>
</tr>
<tr>
<td>D</td>
<td>Decimal point (uses locale)</td>
</tr>
<tr>
<td>G</td>
<td>Group separator (uses locale)</td>
</tr>
<tr>
<td>-MI</td>
<td>Minus sign specified in right-most position (if number &lt; 0)</td>
</tr>
<tr>
<td>RN or rn</td>
<td>Roman numeral (input between 1 and 3999)</td>
</tr>
<tr>
<td>V</td>
<td>Shift specified number of digits (see notes)</td>
</tr>
</tbody>
</table>

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 $10^n$, 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 3-3-26 TO_CHAR Examples

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>TO_CHAR(CURRENT_TIMESTAMP, 'Day, DD HH12:MI:SS')</td>
<td>'Tuesday, 06 05:39:18'</td>
</tr>
<tr>
<td>TO_CHAR(CURRENT_TIMESTAMP, 'FMDay, FMDD HH12:MI:SS')</td>
<td>'Tuesday, 6 05:39:18'</td>
</tr>
<tr>
<td>TO_CHAR(-0.1, '99.99')</td>
<td>'-.10'</td>
</tr>
<tr>
<td>TO_CHAR(-0.1, 'FM9.99')</td>
<td>'-.1'</td>
</tr>
<tr>
<td>TO_CHAR(0.1, '0.9')</td>
<td>'0.1'</td>
</tr>
<tr>
<td>TO_CHAR(12, '9990999.9')</td>
<td>'0012.0'</td>
</tr>
<tr>
<td>TO_CHAR(12, 'FM9990999.9')</td>
<td>'0012.'</td>
</tr>
<tr>
<td>TO_CHAR(485, '999')</td>
<td>'485'</td>
</tr>
<tr>
<td>TO_CHAR(-485, '999')</td>
<td>'-485'</td>
</tr>
<tr>
<td>TO_CHAR(1485, '9,999')</td>
<td>'1,485'</td>
</tr>
<tr>
<td>TO_CHAR(1485, '9G9999')</td>
<td>'1,485'</td>
</tr>
<tr>
<td>TO_CHAR(148.5, '999.999')</td>
<td>'148.500'</td>
</tr>
<tr>
<td>TO_CHAR(148.5, 'FM999.999')</td>
<td>'148.5'</td>
</tr>
<tr>
<td>Expression</td>
<td>Result</td>
</tr>
<tr>
<td>------------------------------------------------</td>
<td>-------------------</td>
</tr>
<tr>
<td>TO_CHAR(148.5, 'FM999.990')</td>
<td>'148.500'</td>
</tr>
<tr>
<td>TO_CHAR(148.5, '999D999')</td>
<td>'148.500'</td>
</tr>
<tr>
<td>TO_CHAR(3148.5, '9G999D999')</td>
<td>'3148.500'</td>
</tr>
<tr>
<td>TO_CHAR(-485, '999S')</td>
<td>'-485-'</td>
</tr>
<tr>
<td>TO_CHAR(-485, '999MI')</td>
<td>'-485-'</td>
</tr>
<tr>
<td>TO_CHAR(485, '999MI')</td>
<td>'485 '</td>
</tr>
<tr>
<td>TO_CHAR(485, 'FM999MI')</td>
<td>'485 '</td>
</tr>
<tr>
<td>TO_CHAR(-485, '999PR')</td>
<td>'&lt;485&gt;'</td>
</tr>
<tr>
<td>TO_CHAR(485, 'L999')</td>
<td>'$ 485'</td>
</tr>
<tr>
<td>TO_CHAR(485, 'RN')</td>
<td>'CDLXXXV'</td>
</tr>
<tr>
<td>TO_CHAR(485, 'FMRN')</td>
<td>'CDLXXXV'</td>
</tr>
<tr>
<td>TO_CHAR(5.2, 'FMRN')</td>
<td>'V'</td>
</tr>
<tr>
<td>TO_CHAR(12, '99V999')</td>
<td>'12000'</td>
</tr>
<tr>
<td>TO_CHAR(12.4, '99V999')</td>
<td>'12400'</td>
</tr>
<tr>
<td>TO_CHAR(12.45, '99V9')</td>
<td>'125'</td>
</tr>
</tbody>
</table>
3.5.7 Date/Time Functions and Operators

Table 3-3-28 shows the available functions for date/time value processing, with details appearing in the following subsections. Table 3-3-27 illustrates the behaviors of the basic arithmetic operators (+, -). For formatting functions, refer to Section 3.5.6. You should be familiar with the background information on date/time data types from Section 3.5.7.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>DATE '2001-09-28' + 7</td>
<td>05-OCT-01 00:00:00</td>
</tr>
<tr>
<td>+</td>
<td>TIMESTAMP '2001-09-28 13:30:00' + 3</td>
<td>01-OCT-01 13:30:00</td>
</tr>
<tr>
<td>-</td>
<td>DATE '2001-10-01' - 7</td>
<td>24-SEP-01 00:00:00</td>
</tr>
<tr>
<td>-</td>
<td>TIMESTAMP '2001-09-28 13:30:00' - 3</td>
<td>25-SEP-01 13:30:00</td>
</tr>
<tr>
<td>-</td>
<td>TIMESTAMP '2001-09-29 03:00:00' -</td>
<td>@ 1 day 15 hours</td>
</tr>
<tr>
<td></td>
<td>TIMESTAMP '2001-09-27 12:00:00'</td>
<td></td>
</tr>
</tbody>
</table>

In the date/time functions of Table 3-3-28 the use of the DATE and TIMESTAMP data types are interchangeable.

<table>
<thead>
<tr>
<th>Function</th>
<th>Return Type</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD_MONTHS(DATE, NUMBER)</td>
<td>DATE</td>
<td>Add months to a date; see Section 3.5.7.1</td>
<td>ADD_MONTHS('28-FEB-97', 3.8)</td>
<td>31-MAY-97 00:00:00</td>
</tr>
<tr>
<td>CURRENT_DATE</td>
<td>DATE</td>
<td>Current date; see Section 3.5.7.8</td>
<td>CURRENT_DATE</td>
<td>04-JUL-07</td>
</tr>
<tr>
<td>CURRENT_TIMESTAMP</td>
<td>TIMESTAMP</td>
<td>Returns current date and time; see Section 3.5.7.8</td>
<td>CURRENT_TIMESTAMP</td>
<td>04-JUL-07 15:33:23.484</td>
</tr>
<tr>
<td>EXTRACT(field FROM TIMESTAMP)</td>
<td>DOUBLE PRECISION</td>
<td>Get subfield; see Section 3.5.7.2</td>
<td>EXTRACT(hour FROM TIMESTAMP '2001-02-16 20:38:40')</td>
<td>20</td>
</tr>
<tr>
<td>LAST_DAY(DATE)</td>
<td>DATE</td>
<td>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.</td>
<td>LAST_DAY('14-APR-98')</td>
<td>30-APR-98 00:00:00</td>
</tr>
<tr>
<td>LOCALTIMESTAMP [ (precision) ]</td>
<td>TIMESTAMP</td>
<td>Current date and time (start of current transaction); see Section 3.5.7.8</td>
<td>LOCALTIMESTAMP</td>
<td>04-JUL-07 15:33:23.484</td>
</tr>
<tr>
<td>MONTHS_BETWEEN(DATE, DATE)</td>
<td>NUMBER</td>
<td>Number of months between two dates; see Section 3.5.7.3</td>
<td>MONTHS_BETWEEN('28-FEB-07', '30-NOV-06')</td>
<td>3</td>
</tr>
<tr>
<td>NEXT_DAY(DATE, dayofweek)</td>
<td>DATE</td>
<td>Date falling on dayofweek following specified date; see Section 3.5.7.4</td>
<td>NEXT_DAY('16-APR-07', 'FRI')</td>
<td>20-APR-07 00:00:00</td>
</tr>
<tr>
<td>Function</td>
<td>Return Type</td>
<td>Description</td>
<td>Example</td>
<td>Result</td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>-------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>-------------------------------------------------------------------------</td>
<td>-----------------------------</td>
</tr>
<tr>
<td>NEW_TIME(DATE, VARCHAR, VARCHAR)</td>
<td>DATE</td>
<td>Converts a date and time to an alternate time zone</td>
<td>NEW_TIME(TO_DATE '2005/05/29 01:45', 'AST', 'PST')</td>
<td>2005/05/29 21:45:00</td>
</tr>
<tr>
<td>ROUND(DATE [, format ])</td>
<td>DATE</td>
<td>Date rounded according to format: see Section 3.5.7.6</td>
<td>ROUND(TO_DATE('29-MAY-05'), 'MON')</td>
<td>01-JUN-05 00:00:00</td>
</tr>
<tr>
<td>SYS_EXTRACT_UTC(TIME STAMP WITH TIME ZONE)</td>
<td>TIMESTAMP</td>
<td>Returns Coordinated Universal Time</td>
<td>SYS_EXTRACT_UTC(CAST('24-MAR-11 12:30:00PM -04:00' AS TIMESTAMP WITH TIME ZONE))</td>
<td>24-MAR-11 16:30:00</td>
</tr>
<tr>
<td>SYSDATE</td>
<td>DATE</td>
<td>Returns current date and time</td>
<td>SYSDATE</td>
<td>01-AUG-12 11:12:34</td>
</tr>
<tr>
<td>SYSTIMESTAMP()</td>
<td>TIMESTAMP</td>
<td>Returns current date and time</td>
<td>SYSTIMESTAMP</td>
<td>01-AUG-12 11:11:23.665</td>
</tr>
<tr>
<td>TRUNC(DATE [format])</td>
<td>DATE</td>
<td>Truncate according to format; see Section 3.5.7.7</td>
<td>TRUNC(TO_DATE('29-MAY-05'), 'MON')</td>
<td>01-MAY-05 00:00:00</td>
</tr>
</tbody>
</table>

### 3.5.7.1 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.

```sql
SELECT ADD_MONTHS('13-JUN-07',4) FROM DUAL;
```

```
add_months
----------------------
13-OCT-07 00:00:00:00
(1 row)
```

```sql
SELECT ADD_MONTHS('31-DEC-06',2) FROM DUAL;
```

```
add_months
----------------------
28-FEB-07 00:00:00:00
(1 row)
```

```sql
SELECT ADD_MONTHS('31-MAY-04',-3) FROM DUAL;
```

```
add_months
----------------------
31-AUG-04 00:00:00:00
(1 row)
```
3.5.7.2 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

```sql
SELECT EXTRACT(YEAR FROM TIMESTAMP '2001-02-16 20:38:40') FROM DUAL;
```

<table>
<thead>
<tr>
<th>date_part</th>
<th>2001</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1 row)</td>
<td></td>
</tr>
</tbody>
</table>

### MONTH

The number of the month within the year (1 - 12)

```sql
SELECT EXTRACT(MONTH FROM TIMESTAMP '2001-02-16 20:38:40') FROM DUAL;
```

<table>
<thead>
<tr>
<th>date_part</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1 row)</td>
<td></td>
</tr>
</tbody>
</table>

### DAY

The day (of the month) field (1 - 31)

```sql
SELECT EXTRACT(DAY FROM TIMESTAMP '2001-02-16 20:38:40') FROM DUAL;
```

<table>
<thead>
<tr>
<th>date_part</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1 row)</td>
<td></td>
</tr>
</tbody>
</table>

### HOUR

The hour field (0 - 23)

```sql
SELECT EXTRACT(HOUR FROM TIMESTAMP '2001-02-16 20:38:40') FROM DUAL;
```
MINUTE

The minutes field (0 - 59)

```sql
SELECT EXTRACT(MINUTE FROM TIMESTAMP '2001-02-16 20:38:40') FROM DUAL;
```

SECOND

The seconds field, including fractional parts (0 - 59)

```sql
SELECT EXTRACT(SECOND FROM TIMESTAMP '2001-02-16 20:38:40') FROM DUAL;
```

3.5.7.3 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.

```sql
SELECT MONTHS_BETWEEN('15-DEC-06','15-OCT-06') FROM DUAL;
```

```sql
months_between
-------------
  2
(1 row)
```

```sql
SELECT MONTHS_BETWEEN('15-OCT-06','15-DEC-06') FROM DUAL;
```

```sql
months_between
-------------
 -2
(1 row)
```
3.5.7.4 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.

```sql
SELECT NEXT_DAY(TO_DATE('13-AUG-07','DD-MON-YY'),'SUNDAY') FROM DUAL;
```

```sql
next_day
-------------------
19-AUG-07 00:00:00
(1 row)
```

```sql
SELECT NEXT_DAY(TO_DATE('13-AUG-07','DD-MON-YY'),'MON') FROM DUAL;
```

```sql
next_day
-------------------
20-AUG-07 00:00:00
(1 row)
```

3.5.7.5 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

<table>
<thead>
<tr>
<th>Time Zone</th>
<th>Offset from UTC</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AST</td>
<td>UTC+4</td>
<td>Atlantic Standard Time</td>
</tr>
<tr>
<td>ADT</td>
<td>UTC+3</td>
<td>Atlantic Daylight Time</td>
</tr>
<tr>
<td>BST</td>
<td>UTC+11</td>
<td>Bering Standard Time</td>
</tr>
<tr>
<td>BDT</td>
<td>UTC+10</td>
<td>Bering Daylight Time</td>
</tr>
<tr>
<td>CST</td>
<td>UTC+6</td>
<td>Central Standard Time</td>
</tr>
<tr>
<td>CDT</td>
<td>UTC+5</td>
<td>Central Daylight Time</td>
</tr>
<tr>
<td>EST</td>
<td>UTC+5</td>
<td>Eastern Standard Time</td>
</tr>
<tr>
<td>EDT</td>
<td>UTC+4</td>
<td>Eastern Daylight Time</td>
</tr>
<tr>
<td>GMT</td>
<td>UTC</td>
<td>Greenwich Mean Time</td>
</tr>
<tr>
<td>HST</td>
<td>UTC+10</td>
<td>Alaska-Hawaii Standard Time</td>
</tr>
<tr>
<td>HDT</td>
<td>UTC+9</td>
<td>Alaska-Hawaii Daylight Time</td>
</tr>
<tr>
<td>MST</td>
<td>UTC+7</td>
<td>Mountain Standard Time</td>
</tr>
<tr>
<td>MDT</td>
<td>UTC+6</td>
<td>Mountain Daylight Time</td>
</tr>
<tr>
<td>NST</td>
<td>UTC+3:30</td>
<td>Newfoundland Standard Time</td>
</tr>
<tr>
<td>PST</td>
<td>UTC+8</td>
<td>Pacific Standard Time</td>
</tr>
<tr>
<td>PDT</td>
<td>UTC+7</td>
<td>Pacific Daylight Time</td>
</tr>
<tr>
<td>YST</td>
<td>UTC+9</td>
<td>Yukon Standard Time</td>
</tr>
<tr>
<td>YDT</td>
<td>UTC+8</td>
<td>Yukon Daylight Time</td>
</tr>
</tbody>
</table>

Following is an example of the `NEW_TIME` function.

```sql
```

Pacific Standard Time
---------------------
13-AUG-07 06:35:15
(1 row)

### 3.5.7.6 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 3-3-29 Template Date Patterns for the ROUND Function

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CC, SCC</td>
<td>Returns January 1, cc01 where cc is first 2 digits of the given year if last 2 digits &lt;= 50, or 1 greater than the first 2 digits of the given year if last 2 digits &gt; 50; (for AD years)</td>
</tr>
<tr>
<td>SYYY, YYYY, YEAR, SYEAR, YY, YY, Y</td>
<td>Returns January 1, yyyy where yyyy is rounded to the nearest year; rounds down on June 30, rounds up on July 1</td>
</tr>
<tr>
<td>IYYYY, IYY, IY, I</td>
<td>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</td>
</tr>
<tr>
<td>Q</td>
<td>Returns the first day of the quarter determined by rounding down if the month and...</td>
</tr>
</tbody>
</table>
Pattern | Description |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>MONTH, MON, MM, RM</td>
<td>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</td>
</tr>
<tr>
<td>WW</td>
<td>Round to the nearest date that corresponds to the same day of the week as the first day of the year</td>
</tr>
<tr>
<td>IW</td>
<td>Round to the nearest date that corresponds to the same day of the week as the first day of the ISO year</td>
</tr>
<tr>
<td>W</td>
<td>Round to the nearest date that corresponds to the same day of the week as the first day of the month</td>
</tr>
<tr>
<td>DDD, DD, J</td>
<td>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</td>
</tr>
<tr>
<td>DAY, DY, D</td>
<td>Rounds to the nearest Sunday</td>
</tr>
<tr>
<td>HH, HH12, HH24</td>
<td>Round to the nearest hour</td>
</tr>
<tr>
<td>MI</td>
<td>Round to the nearest minute</td>
</tr>
</tbody>
</table>

Following are examples of usage of the `ROUND` function.

The following examples round to the nearest hundred years.

```sql
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.

```sql
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 29\textsuperscript{th} of 2003. The second example rounds to 2005 and the ISO year for 2005 begins on January 3\textsuperscript{rd} 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;

<table>
<thead>
<tr>
<th>ISO Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>29-DEC-2003</td>
</tr>
<tr>
<td>(1 row)</td>
</tr>
</tbody>
</table>
```

```
SELECT TO_CHAR(ROUND(TO_DATE('01-JUL-2004','DD-MON-YYYY'),'IYYY'),'DD-MON-YYYY') "ISO Year" FROM DUAL;

<table>
<thead>
<tr>
<th>ISO Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>03-JAN-2005</td>
</tr>
<tr>
<td>(1 row)</td>
</tr>
</tbody>
</table>
```

The following examples round to the nearest quarter.

```
SELECT ROUND(TO_DATE('15-FEB-07','DD-MON-YY'),'Q') "Quarter" FROM DUAL;

<table>
<thead>
<tr>
<th>Quarter</th>
</tr>
</thead>
<tbody>
<tr>
<td>01-JAN-07 00:00:00</td>
</tr>
<tr>
<td>(1 row)</td>
</tr>
</tbody>
</table>
```

```
SELECT ROUND(TO_DATE('16-FEB-07','DD-MON-YY'),'Q') "Quarter" FROM DUAL;

<table>
<thead>
<tr>
<th>Quarter</th>
</tr>
</thead>
<tbody>
<tr>
<td>01-APR-07 00:00:00</td>
</tr>
<tr>
<td>(1 row)</td>
</tr>
</tbody>
</table>
```

The following examples round to the nearest month.

```
SELECT ROUND(TO_DATE('15-DEC-07','DD-MON-YY'),'MONTH') "Month" FROM DUAL;

<table>
<thead>
<tr>
<th>Month</th>
</tr>
</thead>
<tbody>
<tr>
<td>01-DEC-07 00:00:00</td>
</tr>
<tr>
<td>(1 row)</td>
</tr>
</tbody>
</table>
```

```
SELECT ROUND(TO_DATE('16-DEC-07','DD-MON-YY'),'MONTH') "Month" FROM DUAL;

<table>
<thead>
<tr>
<th>Month</th>
</tr>
</thead>
<tbody>
<tr>
<td>01-JAN-08 00:00:00</td>
</tr>
<tr>
<td>(1 row)</td>
</tr>
</tbody>
</table>
```
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.
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)
```

The following examples round to the nearest hour.

```
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)
```
3.5.7.7 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 3-3-30 Template Date Patterns for the TRUNC Function

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CC, SCC</td>
<td>Returns January 1, cc01 where cc is first 2 digits of the given year</td>
</tr>
<tr>
<td>SYYY, YYYY, YEAR, SYEAR, YY, Y</td>
<td>Returns January 1, yyyy where yyyy is the given year</td>
</tr>
<tr>
<td>IYYY, IYY, IY, I</td>
<td>Returns the start date of the ISO year containing the given date</td>
</tr>
<tr>
<td>Q</td>
<td>Returns the first day of the quarter containing the given date</td>
</tr>
<tr>
<td>MONTH, MON, MM, RM</td>
<td>Returns the first day of the specified month</td>
</tr>
<tr>
<td>WW</td>
<td>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</td>
</tr>
<tr>
<td>IW</td>
<td>Returns the start of the ISO week containing the given date</td>
</tr>
<tr>
<td>w</td>
<td>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</td>
</tr>
<tr>
<td>DDD, DD, J</td>
<td>Returns the start of the day for the given date</td>
</tr>
<tr>
<td>DAY, DY, D</td>
<td>Returns the start of the week (Sunday) containing the given date</td>
</tr>
<tr>
<td>HH, HH24, HH24</td>
<td>Returns the start of the hour</td>
</tr>
<tr>
<td>MI</td>
<td>Returns the start of the minute</td>
</tr>
</tbody>
</table>

Following are examples of usage of the TRUNC function.

The following example truncates down to the hundred years unit.

```sql
SELECT TO_CHAR(TRUNC(TO_DATE('1951','YYYY'),'CC'),'DD-MON-YYYY') "Century" FROM DUAL;
```

<table>
<thead>
<tr>
<th>Century</th>
<th>01-MAR-1901</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1 row)</td>
<td></td>
</tr>
</tbody>
</table>

The following example truncates down to the year.

```sql
SELECT TO_CHAR(TRUNC(TO_DATE('01-JUL-1999','DD-MON-YYYY'),'Y'),'DD-MON-YYYY') "Year" FROM DUAL;
```

---

Steven Feuerstein
Publisher's Copy Kendall and Company
Copyright © 2007 - 2015 EnterpriseDB Corporation. All rights reserved.
The following example truncates down to the beginning of the ISO year.

```sql
SELECT TO_CHAR(TRUNC(TO_DATE('01-JUL-2004', 'DD-MON-YYYY'), 'IYYY'), 'DD-MON-YYYY') "ISO Year" FROM DUAL;
```

<table>
<thead>
<tr>
<th>ISO Year</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>29-DEC-2003</td>
<td>(1 row)</td>
</tr>
</tbody>
</table>

The following example truncates down to the start date of the quarter.

```sql
SELECT TRUNC(TO_DATE('16-FEB-07', 'DD-MON-YY'), 'Q') "Quarter" FROM DUAL;
```

<table>
<thead>
<tr>
<th>Quarter</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>01-JAN-07 00:00:00</td>
<td>(1 row)</td>
</tr>
</tbody>
</table>

The following example truncates to the start of the month.

```sql
SELECT TRUNC(TO_DATE('16-DEC-07', 'DD-MON-YY'), 'MONTH') "Month" FROM DUAL;
```

<table>
<thead>
<tr>
<th>Month</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>01-DEC-07 00:00:00</td>
<td>(1 row)</td>
</tr>
</tbody>
</table>

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.

```sql
SELECT TRUNC(TO_DATE('19-JAN-07', 'DD-MON-YY'), 'WW') "Week" FROM DUAL;
```

<table>
<thead>
<tr>
<th>Week</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>15-JAN-07 00:00:00</td>
<td>(1 row)</td>
</tr>
</tbody>
</table>


```sql
SELECT TRUNC(TO_DATE('02-JAN-04', 'DD-MON-YY'), 'IW') "ISO Week" FROM DUAL;
```

<table>
<thead>
<tr>
<th>ISO Week</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>29-DEC-03 00:00:00</td>
<td>(1 row)</td>
</tr>
</tbody>
</table>
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)
3.5.7.8 CURRENT DATE/TIME

Postgres Plus 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 17:52:29.361473 +05:00 | 02-SEP-13 17:52:29.361474 +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.
3.5.8 Sequence Manipulation Functions

This section describes Postgres Plus 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.

\[ \text{sequence.NEXTVAL} \]
\[ \text{sequence.CURRVAL} \]

*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 aberts. This means that aborted transactions may leave unused "holes" in the sequence of assigned values.
3.5.9 Conditional Expressions

The following section describes the SQL-compliant conditional expressions available in Postgres Plus Advanced Server.

3.5.9.1 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;
```

<table>
<thead>
<tr>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
</tbody>
</table>
(3 rows)

```
SELECT a,
    CASE WHEN a=1 THEN 'one'
        WHEN a=2 THEN 'two'
    ELSE 'other'
END
FROM test;
```

<table>
<thead>
<tr>
<th>a</th>
<th>case</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>one</td>
</tr>
<tr>
<td>2</td>
<td>two</td>
</tr>
<tr>
<td>3</td>
<td>other</td>
</tr>
</tbody>
</table>
(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;
```

### 3.5.9.2 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.
3.5.9.3 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.

3.5.9.4 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)
```

3.5.9.5 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.

\[
\text{bonus} = \text{NVL2(} \text{emp.commission, emp.commission} \times 1.1, 0) \]

### 3.5.9.6 GREATEST and LEAST

The **GREATEST** and **LEAST** functions select the largest or smallest value from a list of any number of expressions.

\[
\text{GREATEST(value [, value2 ] ... )} \\
\text{LEAST(value [, value2 ] ... )}
\]

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.
3.5.10 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 3-3-31 General-Purpose Aggregate Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Argument Type</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVG(expression)</td>
<td>INTEGER, REAL, DOUBLE PRECISION, NUMBER</td>
<td>NUMBER for any integer type, DOUBLE PRECISION for a floating-point argument, otherwise the same as the argument data type</td>
<td>The average (arithmetic mean) of all input values</td>
</tr>
<tr>
<td>COUNT(*)</td>
<td></td>
<td>BIGINT</td>
<td>Number of input rows</td>
</tr>
<tr>
<td>COUNT(expression)</td>
<td>Any</td>
<td>BIGINT</td>
<td>Number of input rows for which the value of expression is not null</td>
</tr>
<tr>
<td>MAX(expression)</td>
<td>Any numeric, string, or date/time type</td>
<td>Same as argument type</td>
<td>Maximum value of expression across all input values</td>
</tr>
<tr>
<td>MIN(expression)</td>
<td>Any numeric, string, or date/time type</td>
<td>Same as argument type</td>
<td>Minimum value of expression across all input values</td>
</tr>
<tr>
<td>SUM(expression)</td>
<td>INTEGER, REAL, DOUBLE PRECISION, NUMBER</td>
<td>BIGINT for SMALLINT or INTEGER arguments, NUMBER for BIGINT arguments, DOUBLE PRECISION for floating-point arguments, otherwise the same as the argument data type</td>
<td>Sum of expression across all input values</td>
</tr>
</tbody>
</table>

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 3-3-32 Aggregate Functions for Statistics

<table>
<thead>
<tr>
<th>Function</th>
<th>Argument Type</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CORR(Y, X)</td>
<td>DOUBLE PRECISION</td>
<td>DOUBLE PRECISION</td>
<td>Correlation coefficient</td>
</tr>
<tr>
<td>COVAR_POP(Y, X)</td>
<td>DOUBLE PRECISION</td>
<td>DOUBLE PRECISION</td>
<td>Population covariance</td>
</tr>
<tr>
<td>COVAR_SAMP(Y, X)</td>
<td>DOUBLE PRECISION</td>
<td>DOUBLE PRECISION</td>
<td>Sample covariance</td>
</tr>
<tr>
<td>REGR_AVGX(Y, X)</td>
<td>DOUBLE PRECISION</td>
<td>DOUBLE PRECISION</td>
<td>Average of the independent variable ( \sum(x) / N )</td>
</tr>
<tr>
<td>Function</td>
<td>Argument Type</td>
<td>Return Type</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------</td>
<td>---------------------</td>
<td>------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>REGR_AVGY(Y, X)</td>
<td>DOUBLE PRECISION</td>
<td>DOUBLE PRECISION</td>
<td>Average of the dependent variable (sum(y) / n)</td>
</tr>
<tr>
<td>REGR_COUNT(Y, X)</td>
<td>DOUBLE PRECISION</td>
<td>DOUBLE PRECISION</td>
<td>Number of input rows in which both expressions are nonnull</td>
</tr>
<tr>
<td>REGR_INTERCEPT(Y, X)</td>
<td>DOUBLE PRECISION</td>
<td>DOUBLE PRECISION</td>
<td>y-intercept of the least-squares-fit linear equation determined by the (x, y) pairs</td>
</tr>
<tr>
<td>REGR_R2(Y, X)</td>
<td>DOUBLE PRECISION</td>
<td>DOUBLE PRECISION</td>
<td>Square of the correlation coefficient</td>
</tr>
<tr>
<td>REGR_SLOPE(Y, X)</td>
<td>DOUBLE PRECISION</td>
<td>DOUBLE PRECISION</td>
<td>Slope of the least-squares-fit linear equation determined by the (x, y) pairs</td>
</tr>
<tr>
<td>REGR_SXX(Y, X)</td>
<td>DOUBLE PRECISION</td>
<td>DOUBLE PRECISION</td>
<td>Sum (x^2) – sum(x)^2 / n (“sum of squares” of the independent variable)</td>
</tr>
<tr>
<td>REGR_SXY(Y, X)</td>
<td>DOUBLE PRECISION</td>
<td>DOUBLE PRECISION</td>
<td>Sum (x*y) – sum(x) * sum(y) / n (“sum of products” of independent times dependent variable)</td>
</tr>
<tr>
<td>REGR_SYY(Y, X)</td>
<td>DOUBLE PRECISION</td>
<td>DOUBLE PRECISION</td>
<td>Sum (y^2) – sum(y)^2 / n (“sum of squares” of the dependent variable)</td>
</tr>
<tr>
<td>STDDEV(expression)</td>
<td>INTEGER, REAL,</td>
<td>DOUBLE PRECISION for floating-point arguments, otherwise NUMBER</td>
<td>Historic alias for STDDEV_SAMP</td>
</tr>
<tr>
<td>STDDEV_POP(expression)</td>
<td>INTEGER, REAL,</td>
<td>DOUBLE PRECISION for floating-point arguments, otherwise NUMBER</td>
<td>Population standard deviation of the input values</td>
</tr>
<tr>
<td>STDDEV_SAMP(expression)</td>
<td>INTEGER, REAL,</td>
<td>DOUBLE PRECISION for floating-point arguments, otherwise NUMBER</td>
<td>Sample standard deviation of the input values</td>
</tr>
<tr>
<td>VARIANCE(expression)</td>
<td>INTEGER, REAL,</td>
<td>DOUBLE PRECISION for floating-point arguments, otherwise NUMBER</td>
<td>Historical alias for VAR_SAMP</td>
</tr>
<tr>
<td>VAR_POP(expression)</td>
<td>INTEGER, REAL,</td>
<td>DOUBLE PRECISION for floating-point arguments, otherwise NUMBER</td>
<td>Population variance of the input values (square of the population standard deviation)</td>
</tr>
<tr>
<td>VAR_SAMP(expression)</td>
<td>INTEGER, REAL,</td>
<td>DOUBLE PRECISION for floating-point arguments, otherwise NUMBER</td>
<td>Sample variance of the input values (square of the sample standard deviation)</td>
</tr>
</tbody>
</table>
3.5.11 Subquery Expressions

This section describes the SQL-compliant subquery expressions available in Postgres Plus Advanced Server. All of the expression forms documented in this section return Boolean (TRUE/FALSE) results.

3.5.11.1 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);
```

```
<table>
<thead>
<tr>
<th>dname</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCOUNTING</td>
</tr>
<tr>
<td>RESEARCH</td>
</tr>
<tr>
<td>SALES</td>
</tr>
</tbody>
</table>
(3 rows)
```

3.5.11.2 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 \texttt{IN} is \texttt{TRUE} if any equal subquery row is found. The result is \texttt{FALSE} if no equal row is found (including the special case where the subquery returns no rows).

\[
\text{expression \texttt{IN} (subquery)}
\]

Note that if the left-hand expression yields \texttt{NULL}, or if there are no equal right-hand values and at least one right-hand row yields \texttt{NULL}, the result of the \texttt{IN} construct will be \texttt{NULL}, not \texttt{FALSE}. This is in accordance with SQL’s normal rules for Boolean combinations of null values.

As with \texttt{EXISTS}, it’s unwise to assume that the subquery will be evaluated completely.

\subsection*{3.5.11.3 \texttt{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 \texttt{NOT IN} is \texttt{TRUE} if only unequal subquery rows are found (including the special case where the subquery returns no rows). The result is \texttt{FALSE} if any equal row is found.

\[
\text{expression \texttt{NOT IN} (subquery)}
\]

Note that if the left-hand expression yields \texttt{NULL}, or if there are no equal right-hand values and at least one right-hand row yields \texttt{NULL}, the result of the \texttt{NOT IN} construct will be \texttt{NULL}, not \texttt{TRUE}. This is in accordance with SQL’s normal rules for Boolean combinations of null values.

As with \texttt{EXISTS}, it’s unwise to assume that the subquery will be evaluated completely.

\subsection*{3.5.11.4 \texttt{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 \texttt{ANY} is \texttt{TRUE} if any true result is obtained. The result is \texttt{FALSE} if no true result is found (including the special case where the subquery returns no rows).

\[
\text{expression \ operator \texttt{ANY} (subquery)}
\]

\[
\text{expression \ operator \texttt{SOME} (subquery)}
\]

\texttt{SOME} is a synonym for \texttt{ANY}. \texttt{IN} is equivalent to \texttt{=} \texttt{ANY}.

Note that if there are no successes and at least one right-hand row yields \texttt{NULL} for the operator’s result, the result of the \texttt{ANY} construct will be \texttt{NULL}, not \texttt{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.

### 3.5.11.5 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.
4 Stored Procedure Language

This chapter describes the Stored Procedure Language - SPL. SPL is a highly productive, procedural programming language for writing custom procedures, functions, triggers, and packages for Postgres Plus Advanced Server. SPL:

- adds full procedural programming functionality to complement the SQL language.
- provides a single, common language to create stored procedures, functions, triggers, and packages for the Postgres Plus Advanced Server database.
- is integrated with Postgres Enterprise Manager Client to provide a seamless development and testing environment.
- promotes the use of reusable code.
- is easy to use.

This chapter describes the basic elements of an SPL program, before providing an overview of the organization of an SPL program and how it is used to create a procedure or a function. Triggers, while still utilizing SPL, are sufficiently different to warrant a separate discussion (see Section 5 for information about triggers). Packages are discussed in Section 6.

The remaining sections of this chapter delve into the details of the SPL language and provide examples of its application.

4.1 Basic SPL Elements

This section discusses the basic programming elements of an SPL program.

4.1.1 Character Set

SPL programs are written using the following set of characters:

- Uppercase letters A thru Z and lowercase letters a thru z
- Digits 0 thru 9
- Symbols ( ) + - * / < > = ! ~ ^ : . ' @ % , " # $ & _ | { } ? [ ]
- White space characters tabs, spaces, and carriage returns

Identifiers, expressions, statements, control structures, etc. that comprise the SPL language are written using these characters.

Note: The data that can be manipulated by an SPL program is determined by the character set supported by the database encoding.
4.1.2 Case Sensitivity

Keywords and user-defined identifiers that are used in an SPL program are case insensitive. So for example, the statement `DBMS_OUTPUT.PUT_LINE('Hello World');` is interpreted to mean the same thing as `dbms_output.put_line('Hello World');` or `Dbms_Output.Put_Line('Hello World');` or `DBMS_output.Put_line('Hello World');`.

Character and string constants, however, are case sensitive as well as any data retrieved from the Postgres Plus Advanced Server database or data obtained from other external sources. The statement `DBMS_OUTPUT.PUT_LINE('Hello World!');` produces the following output:

```
Hello World!
```

However the statement `DBMS_OUTPUT.PUT_LINE('HELLO WORLD!');` produces the output:

```
HELLO WORLD!
```

4.1.3 Identifiers

Identifiers are user-defined names that are used to identify various elements of an SPL program including variables, cursors, labels, programs, and parameters.

The syntax rules for valid identifiers are the same as for identifiers in the SQL language. See Section 3.1.2 for a discussion of SQL identifiers.

An identifier must not be the same as an SPL keyword or a keyword of the SQL language. The following are some examples of valid identifiers:

```
x
last___name
a_$_Sign
Many$$$$$$signs
THIS_IS_AN_EXTREMELY_LONG_NAME
A1
```

4.1.4 Qualifiers

A qualifier is a name that specifies the owner or context of an entity that is the object of the qualification. A qualified object is specified as the qualifier name followed by a dot with no intervening white space, followed by the name of the object being qualified with no intervening white space. This syntax is called dot notation.

The following is the syntax of a qualified object.

```
qualifier. [ qualifier. ]... object
```
**qualifier** is the name of the owner of the object. **object** is the name of the entity belonging to **qualifier**. It is possible to have a chain of qualifications where the preceding qualifier owns the entity identified by the subsequent qualifier(s) and object.

Almost any identifier can be qualified. What an identifier is qualified by depends upon what the identifier represents and the context of its usage.

Some examples of qualification follow:

- Procedure and function names qualified by the schema to which they belong - e.g., `schema_name.procedure_name(...)`
- Trigger names qualified by the schema to which they belong - e.g., `schema_name.trigger_name`
- Column names qualified by the table to which they belong - e.g., `emp.empno`
- Table names qualified by the schema to which they belong - e.g., `public.emp`
- Column names qualified by table and schema - e.g., `public.emp.empno`

As a general rule, wherever a name appears in the syntax of an SPL statement, its qualified name can be used as well.

Typically a qualified name would only be used if there is some ambiguity associated with the name. For example, if two procedures with the same name belonging to two different schemas are invoked from within a program or if the same name is used for a table column and SPL variable within the same program.

It is suggested that qualified names be avoided if at all possible. In this chapter, the following conventions are adopted to avoid such naming conflicts:

- All variables declared in the declaration section of an SPL program are prefixed by `v_`. E.g., `v_empno`
- All formal parameters declared in a procedure or function definition are prefixed by `p_`. E.g., `p_empno`
- Column names and table names do not have any special prefix conventions. E.g., `column empno in table emp`

### 4.1.5 Constants

**Constants** or **literals** are fixed values that can be used in SPL programs to represent values of various types - e.g., numbers, strings, dates, etc. Constants come in the following types:

- Numeric (Integer and Real) – see Section 3.1.3.2 for information on numeric constants.
- Character and String – see Section 3.1.3.1 for information on character and string constants.
Date/time – see Section 3.2.4 for information on date/time data types and constants.

4.1.6 User-Defined PL/SQL Subtypes

Advanced Server supports user-defined PL/SQL subtypes and (subtype) aliases. A subtype is a data type with an optional set of constraints that restrict the values that can be stored in a column of that type. The rules that apply to the type on which the subtype is based are still enforced, but you can use additional constraints to place limits on the precision or scale of values stored in the type.

You can define a subtype in the declaration of a PL function, procedure, anonymous block or package. The syntax is:

```
SUBTYPE subtype_name IS type_name[(constraint)] [NOT NULL]
```

Where `constraint` is:

```
{precision [, scale]} | length
```

Where:

- `subtype_name` specifies the name of the subtype.
- `type_name` specifies the name of the original type on which the subtype is based. `type_name` may be:
  - The name of any of the type supported by Advanced Server.
  - The name of any composite type.
  - A column anchored by a `%TYPE` operator.
  - The name of another subtype.

Include the `constraint` clause to define restrictions for types that support precision or scale.

- `precision` specifies the total number of digits permitted in a value of the subtype.
scale

scale specifies the number of fractional digits permitted in a value of the subtype.

length

length specifies the total length permitted in a value of CHARACTER, VARCHAR, or TEXT base types

Include the NOT NULL clause to specify that NULL values may not be stored in a column of the specified subtype.

Note that a subtype that is based on a column will inherit the column size constraints, but the subtype will not inherit NOT NULL or CHECK constraints.

Unconstrained Subtypes

To create an unconstrained subtype, use the SUBTYPE command to specify the new subtype name and the name of the type on which the subtype is based. For example, the following command creates a subtype named address that has all of the attributes of the type, CHAR:

    SUBTYPE address IS CHAR;

You can also create a subtype (constrained or unconstrained) that is a subtype of another subtype:

    SUBTYPE cust_address IS address NOT NULL;

This command creates a subtype named cust_address that shares all of the attributes of the address subtype. Include the NOT NULL clause to specify that a value of the cust_address may not be NULL.

Constrained Subtypes

Include a length value when creating a subtype that is based on a character type to define the maximum length of the subtype. For example:

    SUBTYPE acct_name IS VARCHAR (15);

This example creates a subtype named acct_name that is based on a VARCHAR data type, but is limited to 15 characters in length.
Include values for *precision* (to specify the maximum number of digits in a value of the subtype) and optionally, *scale* (to specify the number of digits to the right of the decimal point) when constraining a numeric base type. For example:

```
SUBTYPE acct_balance IS NUMBER (5, 2);
```

This example creates a subtype named `acct_balance` that shares all of the attributes of a `NUMBER` type, but that may not exceed 3 digits to the left of the decimal point and 2 digits to the right of the decimal.

An argument declaration (in a function or procedure header) is a *formal argument*. The value passed to a function or procedure is an *actual argument*. When invoking a function or procedure, the caller provides (0 or more) actual arguments. Each actual argument is assigned to a formal argument that holds the value within the body of the function or procedure.

If a formal argument is declared as a constrained subtype:

- Advanced Server does not enforce subtype constraints when assigning an actual argument to a formal argument when invoking a function.
- Advanced Server enforces subtype constraints when assigning an actual argument to a formal argument when invoking a procedure.

**Using the `%TYPE` Operator**

You can use `%TYPE` notation to declare a subtype anchored to a column. For example:

```
SUBTYPE emp_type IS emp.empno%TYPE
```

This command creates a subtype named `emp_type` whose base type matches the type of the `empno` column in the `emp` table. A subtype that is based on a column will share the column size constraints; **NOT NULL** and **CHECK** constraints are not inherited.

**Subtype Conversion**

Unconstrained subtypes are aliases for the type on which they are based. Any variable of type subtype (unconstrained) is interchangeable with a variable of the base type without conversion, and vice versa.

A variable of a constrained subtype may be interchanged with a variable of the base type without conversion, but a variable of the base type may only be interchanged with a constrained subtype if it complies with the constraints of the subtype. A variable of a constrained subtype may be implicitly converted to another subtype if it is based on the same subtype, and the constraint values are within the values of the subtype to which it is being converted.
4.2 SPL Programs

SPL is a procedural, block-structured language. There are four different types of programs that can be created using SPL, namely procedures, functions, triggers, and packages.

Procedures and functions are discussed in more detail later in this section. Triggers are discussed in Section 5 and packages are addressed in Section 6.

4.2.1 SPL Block Structure

Regardless of whether the program is a procedure, function, or trigger, an SPL program has the same block structure. A block consists of up to three sections - an optional declaration section, a mandatory executable section, and an optional exception section. Minimally, a block has an executable section that consists of one or more SPL statements within the keywords, BEGIN and END.

There may be an optional declaration section that is used to declare variables, cursors, and types that are used by the statements within the executable and exception sections. Declarations appear just prior to the BEGIN keyword of the executable section. Depending upon the context of where the block is used, the declaration section may begin with the keyword DECLARE.

Finally, there may be an optional exception section which appears within the BEGIN-END block. The exception section begins with the keyword, EXCEPTION, and continues until the end of the block in which it appears. If an exception is thrown by a statement within the block, program control goes to the exception section where the thrown exception may or may not be handled depending upon the exception and the contents of the exception section.

The following is the general structure of a block:

```
[ [ DECLARE ]
  declarations ]
BEGIN
  statements
[ EXCEPTION
  WHEN exception_condition THEN
  statements [], ... ]
END;
```

*declarations* are one or more variable, cursor, or type declarations that are local to the block. Each declaration must be terminated by a semicolon. The use of the keyword DECLARE depends upon the context in which the block appears.
**Database Compatibility for Oracle® Developer’s Guide**

*statements* are one or more SPL statements. Each statement must be terminated by a semicolon. The end of the block denoted by the keyword **END** must also be terminated by a semicolon.

If present, the keyword **EXCEPTION** marks the beginning of the exception section. **exception_condition** is a conditional expression testing for one or more types of exceptions. If a thrown exception matches one of the exceptions in **exception_condition**, the **statements** following the **WHEN** **exception_condition** clause are executed. There may be one or more **WHEN** **exception_condition** clauses, each followed by **statements**.

**Note:** A **BEGIN/END** block in itself, is considered a statement; thus, blocks may be nested. The exception section may also contain nested blocks.

The following is the simplest possible block consisting of the **NULL** statement within the executable section. The **NULL** statement is an executable statement that does nothing.

```
BEGIN
    NULL;
END;
```

The following block contains a declaration section as well as the executable section.

```
DECLARE
    v_numerator     NUMBER(2);
    v_denominator   NUMBER(2);
    v_result        NUMBER(5,2);
BEGIN
    v_numerator := 75;
    v_denominator := 14;
    v_result := v_numerator / v_denominator;
    DBMS_OUTPUT.PUT_LINE('75 divided by 14 is ' || v_result);
END;
```

In this example, three numeric variables are declared of data type **NUMBER**. In the executable section, values are assigned to two of the variables and then one number is divided by the other, storing the results in a third variable which is then displayed. If this block is executed the output would be as follows.

```
75 divided by 14 is 5.36
```

The following block consists of all three sections - the declaration, executable, and exception sections.

```
DECLARE
    v_numerator     NUMBER(2);
    v_denominator   NUMBER(2);
    v_result        NUMBER(5,2);
BEGIN
    v_numerator := 75;
END;
```

Copyright © 2007 - 2015 EnterpriseDB Corporation. All rights reserved.
v_denominator := 0;
v_result := v_numerator / v_denominator;
DBMS_OUTPUT.PUT_LINE(v_numerator || ' divided by ' || v_denominator || ' is ' || v_result);
EXCEPTION
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('An exception occurred');
END;

The following output shows that the statement within the exception section is executed as a result of the division by zero.

An exception occurred

4.2.2 Anonymous Blocks

The preceding section demonstrated the basic structure of a block. A block can simply be executed in Postgres Plus Advanced Server.

A block of this type is called an anonymous block. An anonymous block is unnamed and is not stored in the database. Once the block has been executed and erased from the application buffer, it cannot be re-executed unless the block code is re-entered into the application.

Anonymous blocks are useful for quick, one-time programs such as for testing.

Typically, however, the same block of code would be re-executed many times. In order to run a block of code repeatedly without the necessity of re-entering the code each time, with some simple modifications, an anonymous block can be turned into a procedure or function. The following sections discuss how to create a procedure or function that can be stored in the database and invoked repeatedly by another procedure, function, or application program.
4.2.3 Procedures Overview

Procedures are SPL programs that are invoked or called as an individual SPL program statement. When called, procedures may optionally receive values from the caller in the form of input parameters and optionally return values to the caller in the form of output parameters.

4.2.3.1 Creating a Procedure

The CREATE PROCEDURE command defines and names a procedure that will be stored in the database.

```
CREATE [ OR REPLACE ] PROCEDURE name [ (parameters) ]
[ AUTHID { DEFINER | CURRENT_USER } ]
{ IS | AS }
[ declarations ]
BEGIN
 statements
END [ name ];
```

**name** is the identifier of the procedure. If [ OR REPLACE ] is specified and a procedure with the same name already exists in the schema, the new procedure replaces the existing one. If [ OR REPLACE ] is not specified, the new procedure will not be allowed to replace an existing one with the same name in the same schema. **parameters** is a list of formal parameters. If the AUTHID clause is omitted or DEFINER is specified, the rights of the procedure owner are used to determine access privileges to database objects. If CURRENT_USER is specified, the rights of the current user executing the procedure are used to determine access privileges. **declarations** are variable, cursor, or type declarations. **statements** are SPL program statements. The BEGIN-END block may contain an EXCEPTION section.

The following is an example of a simple procedure that takes no parameters.

```
CREATE OR REPLACE PROCEDURE simple_procedure
IS
 BEGIN
 DBMS_OUTPUT.PUT_LINE('That''s all folks!');
 END simple_procedure;
```

The procedure is stored in the database by entering the procedure code in Postgres Plus Advanced Server.

See the CREATE PROCEDURE command for more information on creating a procedure.
4.2.3.2 Calling a Procedure

The procedure can be invoked from another SPL program by simply specifying the procedure name followed by its parameters, if any, followed by a semicolon.

```
name [ ([ parameters ]) ];
```

`name` is the identifier of the procedure. `parameters` is a list of actual parameters.

**Note:** If there are no actual parameters to be passed, the procedure may be called with an empty parameter list, or the opening and closing parenthesis may be omitted entirely.

**Note:** The syntax for calling a procedure is the same as in the preceding syntax diagram when executing it with the `EXEC` command in PSQL or EDB*Plus. See Section 11.1.2.12 for information on the `EXEC` command.

The following is an example of calling the procedure from an anonymous block:

```
BEGIN
  simple_procedure;
END;
```

That's all folks!

**Note:** Each application has its own unique way to call a procedure. In a Java application, the application programming interface, JDBC, is used.

4.2.3.3 Deleting a Procedure

A procedure can be deleted from the database using the `DROP PROCEDURE` command.

```
DROP PROCEDURE name;
```

`name` is the name of the procedure to be dropped.

The previously created procedure is dropped in this example:

```
DROP PROCEDURE simple_procedure;
```

See the `DROP PROCEDURE` command for more details.
4.2.4 Functions Overview

Functions are SPL programs that are invoked as expressions. When evaluated, a function returns a value that is substituted in the expression in which the function is embedded. Functions may optionally take values from the calling program in the form of input parameters. In addition to the fact that the function, itself, returns a value, a function may optionally return additional values to the caller in the form of output parameters. The use of output parameters in functions, however, is not an encouraged programming practice.

4.2.4.1 Creating a Function

The `CREATE FUNCTION` command defines and names a function that will be stored in the database.

```
CREATE [ OR REPLACE ] FUNCTION name [ (parameters) ]
    RETURN data_type
    [ AUTHID { DEFINER | CURRENT_USER } ]
{ IS | AS }
BEGIN
    statements
END [ name ];
```

`name` is the identifier of the function. If `OR REPLACE` is specified and a function with the same name already exists in the schema, the new function replaces the existing one. If `[ OR REPLACE ]` is not specified, the new function will not be allowed to replace an existing one with the same name in the same schema. `parameters` is a list of formal parameters. `data_type` is the data type of the value that is returned by the function. If the `AUTHID` clause is omitted or `DEFINER` is specified, the rights of the function owner are used to determine access privileges to database objects. If `CURRENT_USER` is specified, the rights of the current user executing the function are used to determine access privileges. `declarations` are variable, cursor, or type declarations. `statements` are SPL program statements. The `BEGIN-END` block may contain an `EXCEPTION` section.

The following is an example of a simple function that takes no parameters.

```
CREATE OR REPLACE FUNCTION simple_function
    RETURN VARCHAR2
IS
BEGIN
    RETURN 'That''s All Folks!';
END simple_function;
```

The following is another function that takes two input parameters. Parameters will be discussed in more detail in subsequent sections.
CREATE OR REPLACE FUNCTION emp_comp ( 
  p_sal           NUMBER, 
  p_comm          NUMBER 
) RETURN NUMBER 
IS 
BEGIN 
  RETURN (p_sal + NVL(p_comm, 0)) * 24; 
END emp_comp;

See the CREATE FUNCTION command for more information.

### 4.2.4.2 Calling a Function

A function can be used anywhere an expression can appear within an SPL statement. A function is invoked by simply specifying its name followed by its parameters enclosed in parenthesis, if any.

\[ \text{name \ [ (\ [ \text{parameters} \ ] \ ) \ ]} \]

\text{name} is the name of the function. \text{parameters} is a list of actual parameters.

\textbf{Note:} If there are no actual parameters to be passed, the function may be called with an empty parameter list, or the opening and closing parenthesis may be omitted entirely.

The following shows how the function can be called from another SPL program.

```
BEGIN 
  DBMS_OUTPUT.PUT_LINE(simple_function);
END;
```

That's All Folks!

A function is typically used within a SQL statement as shown in the following.

```
SELECT empno "EMPNO", ename "ENAME", sal "SAL", comm "COMM", 
  emp_comp(sal, comm) "YEARLY COMPENSATION" FROM emp;
```

<table>
<thead>
<tr>
<th>EMPNO</th>
<th>ENAME</th>
<th>SAL</th>
<th>COMM</th>
<th>YEARLY COMPENSATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>7369</td>
<td>SMITH</td>
<td>800.00</td>
<td></td>
<td>19200.00</td>
</tr>
<tr>
<td>7499</td>
<td>ALLEN</td>
<td>1600.00</td>
<td>300.00</td>
<td>45600.00</td>
</tr>
<tr>
<td>7521</td>
<td>WARD</td>
<td>1250.00</td>
<td>500.00</td>
<td>42000.00</td>
</tr>
<tr>
<td>7566</td>
<td>JONES</td>
<td>2975.00</td>
<td></td>
<td>71400.00</td>
</tr>
<tr>
<td>7654</td>
<td>MARTIN</td>
<td>1250.00</td>
<td>1400.00</td>
<td>63600.00</td>
</tr>
<tr>
<td>7698</td>
<td>BLAKE</td>
<td>2850.00</td>
<td></td>
<td>68400.00</td>
</tr>
<tr>
<td>7782</td>
<td>CLARK</td>
<td>2450.00</td>
<td></td>
<td>58800.00</td>
</tr>
<tr>
<td>7788</td>
<td>SCOTT</td>
<td>3000.00</td>
<td></td>
<td>72000.00</td>
</tr>
<tr>
<td>7839</td>
<td>KING</td>
<td>5000.00</td>
<td></td>
<td>120000.00</td>
</tr>
<tr>
<td>7844</td>
<td>TURNER</td>
<td>1500.00</td>
<td>0.00</td>
<td>36000.00</td>
</tr>
<tr>
<td>7876</td>
<td>ADAMS</td>
<td>1100.00</td>
<td></td>
<td>26400.00</td>
</tr>
<tr>
<td>7900</td>
<td>JAMES</td>
<td>950.00</td>
<td></td>
<td>22800.00</td>
</tr>
<tr>
<td>7902</td>
<td>FORD</td>
<td>3000.00</td>
<td></td>
<td>72000.00</td>
</tr>
<tr>
<td>7934</td>
<td>MILLER</td>
<td>1300.00</td>
<td></td>
<td>31200.00</td>
</tr>
</tbody>
</table>
```

(14 rows)
4.2.4.3 Deleting a Function

A function can be deleted from the database using the DROP FUNCTION command.

```
DROP FUNCTION name [ (parameters) ];
```

`name` is the name of the function to be dropped.

**Note:** The specification of the parameter list is required in Postgres Plus Advanced Server under certain circumstances. Oracle requires that the parameter list always be omitted.

The previously created function is dropped in this example:

```
DROP FUNCTION simple_function;
```

See the DROP FUNCTION command for more details.
4.2.5 Procedure and Function Parameters

An important aspect of using procedures and functions is the capability to pass data from the calling program to the procedure or function and to receive data back from the procedure or function. This is accomplished by using parameters.

Parameters are declared in the procedure or function definition, enclosed within parenthesis following the procedure or function name. Parameters declared in the procedure or function definition are known as formal parameters. When the procedure or function is invoked, the calling program supplies the actual data that is to be used in the called program’s processing as well as the variables that are to receive the results of the called program’s processing. The data and variables supplied by the calling program when the procedure or function is called are referred to as the actual parameters.

The following is the general format of a formal parameter declaration.

\[
\text{name [ IN | OUT | IN OUT ] data_type [ DEFAULT value ]}
\]

name is an identifier assigned to the formal parameter. If specified, IN defines the parameter for receiving input data into the procedure or function. An IN parameter can also be initialized to a default value. If specified, OUT defines the parameter for returning data from the procedure or function. If specified, IN OUT allows the parameter to be used for both input and output. If all of IN, OUT, and IN OUT are omitted, then the parameter acts as if it were defined as IN by default. Whether a parameter is IN, OUT, or IN OUT is referred to as the parameter’s mode. data_type defines the data type of the parameter. value is a default value assigned to an IN parameter in the called program if an actual parameter is not specified in the call.

The following is an example of a procedure that takes parameters:

```sql
CREATE OR REPLACE PROCEDURE emp_query (  
  p_deptno        IN     NUMBER,
  p_empno         IN OUT NUMBER,
  p_ename         IN OUT VARCHAR2,
  p_job           OUT    VARCHAR2,
  p_hiredate      OUT    DATE,
  p_sal           OUT    NUMBER
)  
IS  
BEGIN
  SELECT empno, ename, job, hiredate, sal  
  INTO p_empno, p_ename, p_job, p_hiredate, p_sal  
  FROM emp  
  WHERE deptno = p_deptno  
  AND (empno = p_empno  
    OR ename = UPPER(p_ename));
END;
```
In this example, \texttt{p_deptno} is an \texttt{IN} formal parameter, \texttt{p_empno} and \texttt{p_ename} are \texttt{IN OUT} formal parameters, and \texttt{p_job}, \texttt{p_hiredate}, and \texttt{p_sal} are \texttt{OUT} formal parameters.

**Note:** In the previous example, no maximum length was specified on the \texttt{VARCHAR2} parameters and no precision and scale were specified on the \texttt{NUMBER} parameters. It is illegal to specify a length, precision, scale or other constraints on parameter declarations. These constraints are automatically inherited from the actual parameters that are used when the procedure or function is called.

The \texttt{emp_query} procedure can be called by another program, passing it the actual parameters. The following is an example of another SPL program that calls \texttt{emp_query}.

\begin{verbatim}
DECLARE
  v_deptno        NUMBER(2);
  v_empno         NUMBER(4);
  v_ename         VARCHAR2(10);
  v_job           VARCHAR2(9);
  v_hiredate      DATE;
  v_sal           NUMBER;
BEGIN
  v_deptno := 30;
  v_empno  := 7900;
  v_ename  := '';
  emp_query(v_deptno, v_empno, v_ename, v_job, v_hiredate, v_sal);
  DBMS_OUTPUT.PUT_LINE('Department : ' || v_deptno);
  DBMS_OUTPUT.PUT_LINE('Employee No: ' || v_empno);
  DBMS_OUTPUT.PUT_LINE('Name       : ' || v_ename);
  DBMS_OUTPUT.PUT_LINE('Job        : ' || v_job);
  DBMS_OUTPUT.PUT_LINE('Hire Date  : ' || v_hiredate);
  DBMS_OUTPUT.PUT_LINE('Salary     : ' || v_sal);
END;
\end{verbatim}

In this example, \texttt{v_deptno}, \texttt{v_empno}, \texttt{v_ename}, \texttt{v_job}, \texttt{v_hiredate}, and \texttt{v_sal} are the actual parameters.

The output from the preceding example is shown as follows:

\begin{verbatim}
Department : 30
Employee No: 7900
Name       : JAMES
Job        : CLERK
Hire Date  : 03-DEC-81
Salary     : 950
\end{verbatim}

### 4.2.5.1 Positional vs. Named Parameter Notation

You can use either \textit{positional} or \textit{named} parameter notation when passing parameters to a function or procedure. If you specify parameters using positional notation, you must list the parameters in the order that they are declared; if you specify parameters with named notation, the order of the parameters is not significant.
To specify parameters using named notation, list the name of each parameter followed by an arrow (=>) and the parameter value. Named notation is more verbose, but makes your code easier to read and maintain.

A simple example that demonstrates using positional and named parameter notation follows:

```sql
CREATE OR REPLACE PROCEDURE emp_info (  
    p_deptno        IN     NUMBER,  
    p_empno         IN OUT NUMBER,  
    p_ename         IN OUT VARCHAR2,  
  )  
IS  
    BEGIN  
      dbms_output.put_line('Department Number ='||p_deptno);  
      dbms_output.put_line('Employee Number ='||p_empno);  
      dbms_output.put_line('Employee Name ='||p_ename);  
    END;  

To call the procedure using positional notation, pass the following:

    emp_info(30, 7455, 'Clark');

To call the procedure using named notation, pass the following:

    emp_info(p_ename =>'Clark', p_empno=>7455, p_deptno=>30);

Using named notation can alleviate the need to re-arrange a procedure’s parameter list if the parameter list changes, if the parameters are reordered or if a new optional parameter is added.

In a case where you have a default value for an argument and the argument is not a trailing argument, you must use named notation to call the procedure or function. The following case demonstrates a procedure with two, leading, default arguments.

```sql
CREATE OR REPLACE PROCEDURE check_balance (  
    p_customerID  IN NUMBER DEFAULT NULL,  
    p_balance     IN NUMBER DEFAULT NULL,  
    p_amount      IN NUMBER  
  )  
IS  
    DECLARE  
      balance NUMBER;  
    BEGIN  
      IF (p_balance IS NULL AND p_customerID IS NULL) THEN  
        RAISE_APPLICATION_ERROR  
          (-20010, 'Must provide balance or customer');  
      ELSEIF (p_balance IS NOT NULL AND p_customerID IS NOT NULL) THEN  
        RAISE_APPLICATION_ERROR  
          (-20020,'Must provide balance or customer, not both');  
      ELSEIF (p_balance IS NULL) THEN  
```

Copyright © 2007 - 2015 EnterpriseDB Corporation. All rights reserved.
balance := getCustomerBalance(p_customerID);
ELSE
    balance := p_balance;
END IF;

IF (amount > balance) THEN
    RAISE_APPLICATION_ERROR
    (-20030, 'Balance insufficient');
END IF;
END;

You can only omit non-trailing argument values (when you call this procedure) by using named notation; when using positional notation, only trailing arguments are allowed to default. You can call this procedure with the following arguments:

    check_balance(p_customerID => 10, p_amount = 500.00)
    check_balance(p_balance => 1000.00, p_amount = 500.00)

You can use a combination of positional and named notation (mixed notation) to specify parameters. A simple example that demonstrates using mixed parameter notation follows:

    CREATE OR REPLACE PROCEDURE emp_info (  
        p_deptno IN NUMBER,  
        p_empno IN OUT NUMBER,  
        p_ename IN OUT VARCHAR2,  
    ) 
    IS 
    BEGIN
        dbms_output.put_line('Department Number = ' || p_deptno);
        dbms_output.put_line('Employee Number = ' || p_empno);
        dbms_output.put_line('Employee Name = ' || p_ename);
    END;

You can call the procedure using mixed notation:

    emp_info(30, p_ename =>'Clark', p_empno=>7455);

If you do use mixed notation, remember that named arguments cannot precede positional arguments.

### 4.2.5.2 Parameter Modes

As previously discussed, a parameter has one of three possible modes - IN, OUT, or IN OUT. The following characteristics of a formal parameter are dependent upon its mode:

- Its initial value when the procedure or function is called.
- Whether or not the called procedure or function can modify the formal parameter.
- How the actual parameter value is passed from the calling program to the called program.
- What happens to the formal parameter value when an unhandled exception occurs in the called program.

The following table summarizes the behavior of parameters according to their mode.

### Table 4-4-1 Parameter Modes

<table>
<thead>
<tr>
<th>Mode Property</th>
<th>IN</th>
<th>IN OUT</th>
<th>OUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Formal parameter initialized to:</td>
<td>Actual parameter value</td>
<td>Actual parameter value</td>
<td>Actual parameter value</td>
</tr>
<tr>
<td>Formal parameter modifiable by the called program?</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Actual parameter contains: (after normal called program termination)</td>
<td>Original actual parameter value prior to the call</td>
<td>Last value of the formal parameter</td>
<td>Last value of the formal parameter</td>
</tr>
<tr>
<td>Actual parameter contains: (after a handled exception in the called program)</td>
<td>Original actual parameter value prior to the call</td>
<td>Last value of the formal parameter</td>
<td>Last value of the formal parameter</td>
</tr>
<tr>
<td>Actual parameter contains: (after an unhandled exception in the called program)</td>
<td>Original actual parameter value prior to the call</td>
<td>Original actual parameter value prior to the call</td>
<td>Original actual parameter value prior to the call</td>
</tr>
</tbody>
</table>

As shown by the table, an IN formal parameter is initialized to the actual parameter with which it is called unless it was explicitly initialized with a default value. The IN parameter may be referenced within the called program, however, the called program may not assign a new value to the IN parameter. After control returns to the calling program, the actual parameter always contains the same value as it was set to prior to the call.

The OUT formal parameter is initialized to the actual parameter with which it is called. The called program may reference and assign new values to the formal parameter. If the called program terminates without an exception, the actual parameter takes on the value last set in the formal parameter. If a handled exception occurs, the value of the actual parameter takes on the last value assigned to the formal parameter. If an unhandled exception occurs, the value of the actual parameter remains as it was prior to the call.

Like an IN parameter, an IN OUT formal parameter is initialized to the actual parameter with which it is called. Like an OUT parameter, an IN OUT formal parameter is modifiable by the called program and the last value in the formal parameter is passed to the calling program’s actual parameter if the called program terminates without an exception. If a handled exception occurs, the value of the actual parameter takes on the last value assigned to the formal parameter. If an unhandled exception occurs, the value of the actual parameter remains as it was prior to the call.
4.2.5.3 Using Default Values in Parameters

You can set a default value of a formal parameter by including the \texttt{DEFAULT} clause or using the assignment operator (\texttt{:=}) in the \texttt{CREATE PROCEDURE} or \texttt{CREATE FUNCTION} statement.

The general form of a formal parameter declaration is:

\[
\text{(name [ IN|OUT|IN OUT ] data_type [{DEFAULT | := } expr ])}
\]

\textit{name} is an identifier assigned to the parameter.

\texttt{IN|OUT|IN OUT} specifies the parameter mode.

\textit{data_type} is the data type assigned to the variable.

\textit{expr} is the default value assigned to the parameter. If you do not include a \texttt{DEFAULT} clause, the caller must provide a value for the parameter.

The default value is evaluated every time the function or procedure is invoked. For example, assigning \texttt{SYSDATE} to a parameter of type \texttt{DATE} causes the parameter to have the time of the current invocation, not the time when the procedure or function was created.

The following simple procedure demonstrates using the assignment operator to set a default value of \texttt{SYSDATE} into the parameter, \texttt{hiredate}:

```sql
CREATE OR REPLACE PROCEDURE hire_emp (p_empno NUMBER, p_ename VARCHAR2, p_hiredate DATE := SYSDATE) RETURN IS BEGIN
    INSERT INTO emp(empno, ename, hiredate)
    VALUES(p_empno, p_ename, p_hiredate);
    DBMS_OUTPUT.PUT_LINE('Hired!');
END emp_comp;
```

If the parameter declaration includes a default value, you can omit the parameter from the actual parameter list when you call the function. Calls to the sample procedure (\texttt{hire_emp}) must include two arguments: the employee number (\texttt{p_empno}) and employee name (\texttt{p_ename}). The third parameter (\texttt{p_hiredate}) defaults to the value of \texttt{SYSDATE}:

```
hire_emp 7575, Clark
```
If you do include a value for the actual parameter when you call the function, that value takes precedence over the default value:

```
hire_emp 7575, Clark, 15-FEB-2010
```

Adds a new employee with a hiredate of February 15, 2010, regardless of the current value of SYSDATE.

You can write the same function by substituting the DEFAULT keyword for the assignment operator:

```sql
CREATE OR REPLACE PROCEDURE hire_emp (    p_empno NUMBER,    p_ename VARCHAR2,    p_hiredate DATE DEFAULT SYSDATE) RETURN IS    BEGIN      INSERT INTO emp(empno, ename, hiredate) VALUES(p_empno, p_ename, p_hiredate);      DBMS_OUTPUT.PUT_LINE('Hired!');    END emp_comp;
```
4.2.6 Compilation Errors in Procedures and Functions

When the Advanced Server parsers compile a procedure or function, they confirm that both the CREATE statement and the program body (that portion of the program that follows the AS keyword) conforms to the grammar rules for SPL and SQL constructs. By default, the server will terminate the compilation process if a parser detects an error. Note that the parsers detect syntax errors in expressions, but not semantic errors (i.e. an expression referencing a non-existent column, table, or function, or a value of incorrect type).

spl.max_error_count instructs the server to stop parsing if it encounters the specified number of errors in SPL code, or when it encounters an error in SQL code. The default value of spl.max_error_count is 10; the maximum value is 1000. Setting spl.max_error_count to a value of 1 instructs the server to stop parsing when it encounters the first error in either SPL or SQL code.

You can use the SET command to specify a value for spl.max_error_count for your current session. The syntax is:

```
SET spl.max_error_count = number_of_errors
```

Where number_of_errors specifies the number of SPL errors that may occur before the server halts the compilation process. For example:

```
SET spl.max_error_count = 6
```

The example instructs the server to continue past the first five SPL errors it encounters. When the server encounters the sixth error it will stop validating, and print six detailed error messages, and one error summary.

To save time when developing new code, or when importing existing code from another source, you may want to set the spl.max_error_count configuration parameter to a relatively high number of errors.

Please note that if you instruct the server to continue parsing in spite of errors in the SPL code in a program body, and the parser encounters an error in a segment of SQL code, there may still be errors in any SPL or SQL code that follows the erroneous SQL code. For example, the following function results in two errors:

```
CREATE FUNCTION computeBonus(baseSalary number) RETURN number AS
BEGIN

    bonus := baseSalary * 1.10;
    total := bonus + 100;
```
RETURN bonus;
END;

ERROR: "bonus" is not a known variable
LINE 4: bonus := baseSalary * 1.10;

ERROR: "total" is not a known variable
LINE 5: total := bonus + 100;

ERROR: compilation of SPL function/procedure "computebonus" failed due to 2 errors

The following example adds a SELECT statement to the previous example. The error in the SELECT statement masks the other errors that follow:

CREATE FUNCTION computeBonus(employeeName number) RETURN number AS
BEGIN
    SELECT salary INTO baseSalary FROM emp
    WHERE ename = employeeName;

    bonus := baseSalary * 1.10;
    total := bonus + 100;

    RETURN bonus;
END;

ERROR: "basesalary" is not a known variable
LINE 3: SELECT salary INTO baseSalary FROM emp WHERE ename = emp...
4.2.7 Program Security

Security over what user may execute an SPL program and what database objects an SPL program may access for any given user executing the program is controlled by the following:

- Privilege to execute a program.
- Privileges granted on the database objects (including other SPL programs) which a program attempts to access.
- Whether the program is defined with definer’s rights or invoker’s rights.

These aspects are discussed in the following sections.

4.2.7.1 EXECUTE Privilege

An SPL program (function, procedure, or package) can begin execution only if any of the following are true:

- The current user is a superuser, or
- The current user has been granted EXECUTE privilege on the SPL program, or
- The current user inherits EXECUTE privilege on the SPL program by virtue of being a member of a group which does have such privilege, or
- EXECUTE privilege has been granted to the PUBLIC group.

Whenever an SPL program is created in Postgres Plus Advanced Server, EXECUTE privilege is automatically granted to the PUBLIC group by default, therefore, any user can immediately execute the program.

This default privilege can be removed by using the REVOKE EXECUTE command. See the REVOKE command for details. The following is an example:

```
REVOKE EXECUTE ON PROCEDURE list_emp FROM PUBLIC;
```

Explicit EXECUTE privilege on the program can then be granted to individual users or groups.

```
GRANT EXECUTE ON PROCEDURE list_emp TO john;
```

Now, user, john, can execute the list_emp program; other users who do not meet any of the conditions listed at the beginning of this section cannot.

Once a program begins execution, the next aspect of security is what privilege checks occur if the program attempts to perform an action on any database object including:
- Reading or modifying table or view data.
- Creating, modifying, or deleting a database object such as a table, view, index, or sequence.
- Obtaining the current or next value from a sequence.
- Calling another program (function, procedure, or package).

Each such action can be protected by privileges on the database object either allowed or disallowed for the user.

Note that it is possible for a database to have more than one object of the same type with the same name, but each such object belonging to a different schema in the database. If this is the case, which object is being referenced by an SPL program? This is the topic of the next section.

### 4.2.7.2 Database Object Name Resolution

A database object inside an SPL program may either be referenced by its qualified name or by an unqualified name. A qualified name is in the form of `schema.name` where `schema` is the name of the schema under which the database object with identifier, `name`, exists. An unqualified name does not have the “`schema.`” portion. When a reference is made to a qualified name, there is absolutely no ambiguity as to exactly which database object is intended – it either does or does not exist in the specified schema.

Locating an object with an unqualified name, however, requires the use of the current user’s search path. When a user becomes the current user of a session, a default search path is always associated with that user. The search path consists of a list of schemas which are searched in left-to-right order for locating an unqualified database object reference. The object is considered non-existent if it can’t be found in any of the schemas in the search path. The default search path can be displayed in PSQL using the `SHOW search_path` command.

```
SHOW search_path;
search_path
----------------------
$use,public,sys, dbo
(1 row)
```

$`user` in the above search path is a generic placeholder that refers to the current user so if the current user of the above session is `enterprisedb`, an unqualified database object would be searched for in the following schemas in this order – first, `enterprisedb`, then `public`, then `sys`, and finally, `dbo`.

Once an unqualified name has been resolved in the search path, it can be determined if the current user has the appropriate privilege to perform the desired action on that specific object.
Note: The concept of the search path is not Oracle compatible. For an unqualified reference, Oracle simply looks in the schema of the current user for the named database object. It also important to note that in Oracle, a user and his or her schema is the same entity while in Postgres Plus Advanced Server, a user and a schema are two distinct objects.

4.2.7.3 Database Object Privileges

Once an SPL program begins execution, any attempt to access a database object from within the program results in a check to ensure the current user has the authorization to perform the intended action against the referenced object. Privileges on database objects are bestowed and removed using the Grant and Revoke commands, respectively. If the current user attempts unauthorized access on a database object, then the program will throw an exception. See Section 4.5.7 for information about exception handling.

The final topic discusses exactly who is the current user.

4.2.7.4 Definer’s vs. Invokers Rights

When an SPL program is about to begin execution, a determination is made as to what user is to be associated with this process. This user is referred to as the current user. The current user’s database object privileges are used to determine whether or not access to database objects referenced in the program will be permitted. The current, prevailing search path in effect when the program is invoked will be used to resolve any unqualified object references.

The selection of the current user is influenced by whether the SPL program was created with definer’s right or invoker’s rights. The AUTHID clause determines that selection. Appearance of the clause AUTHID DEFINER gives the program definer’s rights. This is also the default if the AUTHID clause is omitted. Use of the clause AUTHID CURRENT_USER gives the program invoker’s rights. The difference between the two is summarized as follows:

- If a program has definer’s rights, then the owner of the program becomes the current user when program execution begins. The program owner’s search path is used to resolve unqualified object references and the program owner’s database object privileges are used to determine if access to a referenced object is permitted. In a definer’s rights program, it is irrelevant as to which user actually invoked the program.
- If a program has invoker’s rights, then the current user at the time the program is called remains the current user while the program is executing (but not necessarily within called subprograms – see the following bullet points). When an invoker’s rights program is invoked, the current user is typically the user that started the session (i.e., made the database connection) although it is possible to change the current user after the session has started using the SET ROLE command. In an
invoker’s rights program, it is irrelevant as to which user actually owns the program.

From the previous definitions, the following observations can be made:

- If a definer’s rights program calls a definer’s rights program, the current user changes from the owner of the calling program to the owner of the called program during execution of the called program.
- If a definer’s rights program calls an invoker’s rights program, the owner of the calling program remains the current user during execution of both the calling and called programs.
- If an invoker’s rights program calls an invoker’s rights program, the current user of the calling program remains the current user during execution of the called program.
- If an invokers’ rights program calls a definer’s rights program, the current user switches to the owner of the definer’s rights program during execution of the called program.

The same principles apply if the called program in turn calls another program in the cases cited above.

This section on security concludes with an example using the sample application.

4.2.7.5 Security Example

In the following example, a new database will be created along with two users – hr_mgr who will own a copy of the entire sample application in schema, hr_mgr; and sales_mgr who will own a schema named, sales_mgr, that will have a copy of only the emp table containing only the employees who work in sales.

The procedure list_emp, function hire_clerk, and package emp_admin will be used in this example. All of the default privileges that are granted upon installation of the sample application will be removed and then be explicitly re-granted so as to present a more secure environment in this example.

Programs list_emp and hire_clerk will be changed from the default of definer’s rights to invoker’s rights. It will be then illustrated that when sales_mgr runs these programs, they act upon the emp table in sales_mgr’s schema since sales_mgr’s search path and privileges will be used for name resolution and authorization checking.

Programs get_dept_name and hire_emp in the emp_admin package will then be executed by sales_mgr. In this case, the dept table and emp table in hr_mgr’s schema will be accessed as hr_mgr is the owner of the emp_admin package which is using definer’s rights. Since the default search path is in effect with the $user placeholder, the schema matching the user (in this case, hr_mgr) is used to find the tables.
Step 1 – Create Database and Users

As user `enterprisedb`, create the `hr` database:

```sql
CREATE DATABASE hr;
```

Switch to the hr database and create the users:

```sql
\c hr enterprisedb
CREATE USER hr_mgr IDENTIFIED BY password;
CREATE USER sales_mgr IDENTIFIED BY password;
```

Step 2 – Create the Sample Application

Create the entire sample application, owned by `hr_mgr`, in `hr_mgr`’s schema.

```sql
\c - hr_mgr
\i C:/Program Files/PostgresPlus/9.2AS/installer/server/edb-sample.sql
BEGIN
CREATE TABLE
CREATE TABLE
CREATE TABLE
CREATE VIEW
CREATE SEQUENCE
.
.
.
CREATE PACKAGE
CREATE PACKAGE BODY
COMMIT
```

Step 3 – Create the `emp` Table in Schema `sales_mgr`

Create a subset of the `emp` table owned by `sales_mgr` in `sales_mgr`’s schema.

```sql
\c - hr_mgr
GRANT USAGE ON SCHEMA hr_mgr TO sales_mgr;
\c - sales_mgr
CREATE TABLE emp AS SELECT * FROM hr_mgr.emp WHERE job = 'SALESMAN';
```

In the above example, the `GRANT USAGE ON SCHEMA` command is given to allow `sales_mgr` access into `hr_mgr`’s schema to make a copy of `hr_mgr`’s `emp` table. This step is required in Postgres Plus Advanced Server and is not Oracle compatible since Oracle does not have the concept of a schema that is distinct from its user.

Step 4 – Remove Default Privileges

Remove all privileges to later illustrate the minimum required privileges needed.

```sql
\c - hr_mgr
REVOKE USAGE ON SCHEMA hr_mgr FROM sales_mgr;
REVOKE ALL ON dept FROM PUBLIC;
```
Step 5 – Change list_emp to Invoker’s Rights

While connected as user, hr_mgr, add the AUTHID CURRENT_USER clause to the list_emp program and resave it in Postgres Plus Advanced Server. When performing this step, be sure you are logged on as hr_mgr, otherwise the modified program may wind up in the public schema instead of in hr_mgr’s schema.

```sql
CREATE OR REPLACE PROCEDURE list_emp
AUTHID CURRENT_USER
IS
  v_empno NUMBER(4);
  v_ename VARCHAR2(10);
  CURSOR emp_cur IS
    SELECT empno, ename FROM emp ORDER BY empno;
BEGIN
  OPEN emp_cur;
  DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
  DBMS_OUTPUT.PUT_LINE('-----    -------');
  LOOP
    FETCH emp_cur INTO v_empno, v_ename;
    EXIT WHEN emp_cur%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE(v_empno || '     ' || v_ename);
  END LOOP;
  CLOSE emp_cur;
END;
```

Step 6 – Change hire_clerk to Invoker’s Rights and Qualify Call to new_empno

While connected as user, hr_mgr, add the AUTHID CURRENT_USER clause to the hire_clerk program.

Also, after the BEGIN statement, fully qualify the reference, new_empno, to hr_mgr.new_empno in order to ensure the hire_clerk function call to the new_empno function resolves to the hr_mgr schema.

When resaving the program, be sure you are logged on as hr_mgr, otherwise the modified program may wind up in the public schema instead of in hr_mgr’s schema.

```sql
CREATE OR REPLACE FUNCTION hire_clerk (
  p_ename VARCHAR2,
  p_deptno NUMBER
) RETURN NUMBER
AUTHID CURRENT_USER
IS
  v_empno NUMBER(4);
  v_ename VARCHAR2(10);
  v_job VARCHAR2(9);
  v_mgr NUMBER(4);
```
v_hiredate DATE;
v_sal NUMBER(7,2);
v_comm NUMBER(7,2);
v_deptno NUMBER(2);
BEGIN
  v_empno := hr_mgr.new_empno;
  INSERT INTO emp VALUES (v_empno, p_ename, 'CLERK', 7782,
  TRUNC(SYSDATE), 950.00, NULL, p_deptno);
  SELECT empno, ename, job, mgr, hiredate, sal, comm, deptno INTO
  v_empno, v_ename, v_job, v_mgr, v_hiredate, v_sal, v_comm, v_deptno
  FROM emp WHERE empno = v_empno;
  DBMS_OUTPUT.PUT_LINE('Department : ' || v_deptno);
  DBMS_OUTPUT.PUT_LINE('Employee No: ' || v_empno);
  DBMS_OUTPUT.PUT_LINE('Name       : ' || v_ename);
  DBMS_OUTPUT.PUT_LINE('Job        : ' || v_job);
  DBMS_OUTPUT.PUT_LINE('Manager    : ' || v_mgr);
  DBMS_OUTPUT.PUT_LINE('Hire Date  : ' || v_hiredate);
  DBMS_OUTPUT.PUT_LINE('Salary     : ' || v_sal);
  DBMS_OUTPUT.PUT_LINE('Commission : ' || v_comm);
  RETURN v_empno;
EXCEPTION
  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);
    RETURN -1;
END;

Step 7 – Grant Required Privileges

While connected as user, hr_mgr, grant the privileges needed so sales_mgr can
execute the list_emp procedure, hire_clerk function, and emp_admin package.
Note that the only data object sales_mgr has access to is the emp table in the
sales_mgr schema. sales_mgr has no privileges on any table in the hr_mgr schema.

GRANT USAGE ON SCHEMA hr_mgr TO sales_mgr;
GRANT EXECUTE ON PROCEDURE list_emp TO sales_mgr;
GRANT EXECUTE ON FUNCTION hire_clerk(VARCHAR2,NUMBER) TO sales_mgr;
GRANT EXECUTE ON FUNCTION new_empno() TO sales_mgr;
GRANT EXECUTE ON PACKAGE emp_admin TO sales_mgr;

Step 8 – Run Programs list_emp and hire_clerk

Connect as user, sales_mgr, and run the following anonymous block:

\c - sales_mgr
DECLARE
  v_empno NUMBER(4);
BEGIN
  hr_mgr.list_emp;
  DBMS_OUTPUT.PUT_LINE('*** Adding new employee ***');
  v_empno := hr_mgr.hire_clerk('JONES',40);
  DBMS_OUTPUT.PUT_LINE('*** After new employee added ***');
  hr_mgr.list_emp;
END;

EMPNO  ENAME
The table and sequence accessed by the programs of the anonymous block are illustrated in the following diagram. The gray ovals represent the schemas of sales_mgr and hr_mgr. The current user during each program execution is shown within parenthesis in bold red font.

The table and sequence accessed by the programs of the anonymous block are illustrated in the following diagram. The gray ovals represent the schemas of sales_mgr and hr_mgr. The current user during each program execution is shown within parenthesis in bold red font.

### Figure 3 - Invoker's Rights Programs

Selecting from sales_mgr’s emp table shows that the update was made in this table.

```sql
SELECT empno, ename, hiredate, sal, deptno,
     hr_mgr.emp_admin.get_dept_name(deptno) FROM sales_mgr.emp;
```

<table>
<thead>
<tr>
<th>empno</th>
<th>ename</th>
<th>hiredate</th>
<th>sal</th>
<th>deptno</th>
<th>get_dept_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>7499</td>
<td>ALLEN</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7521</td>
<td>WARD</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7654</td>
<td>MARTIN</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7844</td>
<td>TURNER</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8000</td>
<td>JONES</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The following diagram shows that the `SELECT` command references the `emp` table in the `sales_mgr` schema, but the `dept` table referenced by the `get_dept_name` function in the `emp_admin` package is from the `hr_mgr` schema since the `emp_admin` package has definer’s rights and is owned by `hr_mgr`. The default search path setting with the `$user` placeholder resolves the access by user `hr_mgr` to the `dept` table in the `hr_mgr` schema.

![Diagram showing database compatibility](image)

**Figure 4 Definer’s Rights Package**

**Step 9 – Run Program hire_emp in the emp_admin Package**

While connected as user, `sales_mgr`, run the `hire_emp` procedure in the `emp_admin` package.

```
EXEC hr_mgr.emp_admin.hire_emp(9001, 'ALICE', 'SALESMAN', 8000, TRUNC(SYSDATE), 1000, 7369, 40);
```

This diagram illustrates that the `hire_emp` procedure in the `emp_admin` definer’s rights package updates the `emp` table belonging to `hr_mgr` since the object privileges of `hr_mgr` are used, and the default search path setting with the `$user` placeholder resolves to the schema of `hr_mgr`. 
Figure 5 Definer's Rights Package

Now connect as user, hr_mgr. The following SELECT command verifies that the new employee was added to hr_mgr's emp table since the emp_admin package has definer’s rights and hr_mgr is emp_admin’s owner.

```sql
\c - hr_mgr
SELECT empno, ename, hiredate, sal, deptno,
hr_mgr.emp_admin.get_dept_name(deptno) FROM hr_mgr.emp;
```

<table>
<thead>
<tr>
<th>empno</th>
<th>ename</th>
<th>hiredate</th>
<th>sal</th>
<th>deptno</th>
<th>get_dept_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>7369</td>
<td>SMITH</td>
<td>17-DEC-80 00:00:00</td>
<td>800.00</td>
<td>20</td>
<td>RESEARCH</td>
</tr>
<tr>
<td>7499</td>
<td>ALLEN</td>
<td>20-FEB-81 00:00:00</td>
<td>1600.00</td>
<td>30</td>
<td>SALES</td>
</tr>
<tr>
<td>7521</td>
<td>WARD</td>
<td>22-FEB-81 00:00:00</td>
<td>1250.00</td>
<td>30</td>
<td>SALES</td>
</tr>
<tr>
<td>7566</td>
<td>JONES</td>
<td>02-APR-81 00:00:00</td>
<td>2975.00</td>
<td>20</td>
<td>RESEARCH</td>
</tr>
<tr>
<td>7654</td>
<td>MARTIN</td>
<td>28-SEP-81 00:00:00</td>
<td>1250.00</td>
<td>30</td>
<td>SALES</td>
</tr>
<tr>
<td>7698</td>
<td>BLAKE</td>
<td>01-MAY-81 00:00:00</td>
<td>2850.00</td>
<td>30</td>
<td>SALES</td>
</tr>
<tr>
<td>7782</td>
<td>CLARK</td>
<td>09-JUN-81 00:00:00</td>
<td>2450.00</td>
<td>10</td>
<td>ACCOUNTING</td>
</tr>
<tr>
<td>7788</td>
<td>SCOTT</td>
<td>19-APR-87 00:00:00</td>
<td>3000.00</td>
<td>20</td>
<td>RESEARCH</td>
</tr>
<tr>
<td>7839</td>
<td>KING</td>
<td>17-NOV-81 00:00:00</td>
<td>5000.00</td>
<td>10</td>
<td>ACCOUNTING</td>
</tr>
<tr>
<td>7844</td>
<td>TURNER</td>
<td>08-SEP-81 00:00:00</td>
<td>1500.00</td>
<td>30</td>
<td>SALES</td>
</tr>
<tr>
<td>7876</td>
<td>ADAMS</td>
<td>23-MAY-87 00:00:00</td>
<td>1100.00</td>
<td>20</td>
<td>RESEARCH</td>
</tr>
<tr>
<td>7900</td>
<td>JAMES</td>
<td>03-DEC-81 00:00:00</td>
<td>950.00</td>
<td>30</td>
<td>SALES</td>
</tr>
<tr>
<td>7902</td>
<td>FORD</td>
<td>03-DEC-81 00:00:00</td>
<td>3000.00</td>
<td>20</td>
<td>RESEARCH</td>
</tr>
<tr>
<td>7934</td>
<td>MILLER</td>
<td>23-JAN-82 00:00:00</td>
<td>1300.00</td>
<td>10</td>
<td>ACCOUNTING</td>
</tr>
<tr>
<td>9001</td>
<td>ALICE</td>
<td>08-NOV-07 00:00:00</td>
<td>8000.00</td>
<td>40</td>
<td>OPERATIONS</td>
</tr>
</tbody>
</table>
```

(15 rows)
4.3 Variable Declarations

SPL is a block-structured language. The first section that can appear in a block is the declaration. The declaration contains the definition of variables, cursors, and other types that can be used in SPL statements contained in the block.

4.3.1 Declaring a Variable

Generally, all variables used in a block must be declared in the declaration section of the block. A variable declaration consists of a name that is assigned to the variable and its data type. (See Section 3.2 for a discussion of data types.) Optionally, the variable can be initialized to a default value in the variable declaration.

The general syntax of a variable declaration is:

```
name type [ { := | DEFAULT } { expression | NULL } ];
```

- `name` is an identifier assigned to the variable.
- `type` is the data type assigned to the variable.
- `[ := expression ]`, if given, specifies the initial value assigned to the variable when the block is entered. If the clause is not given then the variable is initialized to the SQL NULL value.

The default value is evaluated every time the block is entered. So, for example, assigning `SYSDATE` to a variable of type `DATE` causes the variable to have the time of the current invocation, not the time when the procedure or function was precompiled.

The following procedure illustrates some variable declarations that utilize defaults consisting of string and numeric expressions.

```sql
CREATE OR REPLACE PROCEDURE dept_salary_rpt (
p_deptno NUMBER)
IS
  todays_date DATE := SYSDATE;
rpt_title VARCHAR2(60) := 'Report For Department # ' || p_deptno || ' on ' || todays_date;
  base_sal INTEGER := 35525;
  base_comm_rate NUMBER := 1.33333;
  base_annual NUMBER := ROUND(base_sal * base_comm_rate, 2);
BEGIN
  DBMS_OUTPUT.PUT_LINE(rpt_title);
  DBMS_OUTPUT.PUT_LINE('Base Annual Salary: ' || base_annual);
END;
```

The following output of the above procedure shows that default values in the variable declarations are indeed assigned to the variables.
4.3.2 Using %TYPE in Variable Declarations

Often, variables will be declared in SPL programs that will be used to hold values from tables in the database. In order to ensure compatibility between the table columns and the SPL variables, the data types of the two should be the same.

However, as quite often happens, a change might be made to the table definition. If the data type of the column is changed, the corresponding change may be required to the variable in the SPL program.

Instead of coding the specific column data type into the variable declaration the column attribute, %TYPE, can be used instead. A qualified column name in dot notation or the name of a previously declared variable must be specified as a prefix to %TYPE. The data type of the column or variable prefixed to %TYPE is assigned to the variable being declared. If the data type of the given column or variable changes, the new data type will be associated with the variable without the need to modify the declaration code.

Note: The %TYPE attribute can be used with formal parameter declarations as well.

```sql
name { { table | view }.column | variable }%TYPE;
```

*name* is the identifier assigned to the variable or formal parameter that is being declared. *column* is the name of a column in *table* or *view*. *variable* is the name of a variable that was declared prior to the variable identified by *name*.

Note: The variable does not inherit any of the column’s other attributes such as might be specified on the column with the NOT NULL clause or the DEFAULT clause.

In the following example a procedure queries the emp table using an employee number, displays the employee’s data, finds the average salary of all employees in the department to which the employee belongs, and then compares the chosen employee’s salary with the department average.

```sql
CREATE OR REPLACE PROCEDURE emp_sal_query (  
p_emppno IN NUMBER
) IS  
v_ename VARCHAR2(10);  
v_job VARCHAR2(9);  
v_hiredate DATE;  
v_sal NUMBER(7,2);  
v_deptno NUMBER(2);  
v_avgsal NUMBER(7,2);  
BEGIN  
SELECT ename, job, hiredate, sal, deptno
```
Instead of the above, the procedure could be written as follows without explicitly coding the `emp` table data types into the declaration section of the procedure.

```sql
CREATE OR REPLACE PROCEDURE emp_sal_query (p_empno IN emp.empno%TYPE)
IS
v_ename emp.ename%TYPE;
v_job emp.job%TYPE;
v_hiredate emp.hiredate%TYPE;
v_sal emp.sal%TYPE;
v_deptno emp.deptno%TYPE;
v_avgsal v_sal%TYPE;
BEGIN
SELECT ename, job, hiredate, sal, deptno
  INTO v_ename, v_job, v_hiredate, v_sal, v_deptno
  FROM emp WHERE empno = p_empno;
DBMS_OUTPUT.PUT_LINE('Employee # : ' || p_empno);
DBMS_OUTPUT.PUT_LINE('Name       : ' || v_ename);
DBMS_OUTPUT.PUT_LINE('Job        : ' || v_job);
DBMS_OUTPUT.PUT_LINE('Hire Date  : ' || v_hiredate);
DBMS_OUTPUT.PUT_LINE('Salary     : ' || v_sal);
DBMS_OUTPUT.PUT_LINE('Dept #     : ' || v_deptno);

SELECT AVG(sal) INTO v_avgsal
  FROM emp WHERE deptno = v_deptno;
IF v_sal > v_avgsal THEN
  DBMS_OUTPUT.PUT_LINE('Employee''s salary is more than the ' || 'department average of ' || v_avgsal);
ELSE
  DBMS_OUTPUT.PUT_LINE('Employee''s salary does not exceed the ' || 'department average of ' || v_avgsal);
END IF;
END;
END;

Note: `p_empno` shows an example of a formal parameter defined using `%TYPE.

`v_avgsal` illustrates the usage of `%TYPE` referring to another variable instead of a table column.
The following is sample output from executing this procedure.

```
EXEC emp_sal_query(7698);
```

Employee #: 7698
Name: BLAKE
Job: MANAGER
Hire Date: 01-MAY-81 00:00:00
Salary: 2850.00
Dept #: 30
Employee's salary is more than the department average of 1566.67

4.3.3 Using %ROWTYPE in Record Declarations

Using the %TYPE attribute provides an easy way to create a variable dependent upon a column’s data type. Using the %ROWTYPE attribute, a record can be defined that contains fields corresponding to all columns of a given table. Each field takes on the data type of its corresponding column.

**Note:** The fields in the record do not inherit any of the columns’ other attributes such as might be specified with the NOT NULL clause or the DEFAULT clause.

A record is a named, ordered collection of fields. A field is similar to a variable; it has an identifier and data type, but has the additional property of belonging to a record, and must be referenced using dot notation with the record name as its qualifier.

A record can be declared using the %ROWTYPE attribute. The %ROWTYPE attribute is prefixed by a table name. Each column in the named table defines an identically named field in the record with the same data type as the column.

```
record table%ROWTYPE;
```

record is an identifier assigned to the record. table is the name of a table whose columns are to define the fields in the record. A view may be used as well to define a record. The following example shows how the emp_sal_query procedure from the prior section can be modified to use emp%ROWTYPE to create a record named r_emp instead of declaring individual variables for the columns in emp.

```
CREATE OR REPLACE PROCEDURE emp_sal_query ( 
    p_empno         IN emp.empno%TYPE 
) 
IS 
    r_emp           emp%ROWTYPE;
    v_avgsal        emp.sal%TYPE;
BEGIN 
    SELECT ename, job, hiredate, sal, deptno INTO r_emp.ename, r_emp.job, r_emp.hiredate, r_emp.sal, r_emp.deptno
```
FROM emp WHERE empno = p_empno;
DBMS_OUTPUT.PUT_LINE('Employee # : ' || p_empno);
DBMS_OUTPUT.PUT_LINE('Name       : ' || r_emp.ename);
DBMS_OUTPUT.PUT_LINE('Job        : ' || r_emp.job);
DBMS_OUTPUT.PUT_LINE('Hire Date  : ' || r_emp.hiredate);
DBMS_OUTPUT.PUT_LINE('Salary     : ' || r_emp.sal);
DBMS_OUTPUT.PUT_LINE('Dept #     : ' || r_emp.deptno);
SELECT AVG(sal) INTO v_avgsal
  FROM emp WHERE deptno = r_emp.deptno;
IF r_emp.sal > v_avgsal THEN
  DBMS_OUTPUT.PUT_LINE('Employee''s salary is more than the ' || 'department average of ' || v_avgsal);
ELSE
  DBMS_OUTPUT.PUT_LINE('Employee''s salary does not exceed the ' || 'department average of ' || v_avgsal);
END IF;
END;

4.3.4 User-Defined Record Types and Record Variables

Records can be declared based upon a table definition using the %ROWTYPE attribute as shown in Section 4.3.3. This section describes how a new record structure can be defined that is not tied to any particular table definition.

The TYPE IS RECORD statement is used to create the definition of a record type. A record type is a definition of a record comprised of one or more identifiers and their corresponding data types. A record type cannot, by itself, be used to manipulate data.

The syntax for a TYPE IS RECORD statement is:

TYPE rec_type IS RECORD ( fields )

Where fields is a comma-separated list of one or more field definitions of the following form:

field_name data_type [NOT NULL] [{:= | DEFAULT} default_value]

Where:

rec_type

rec_type is an identifier assigned to the record type.

field_name

field_name is the identifier assigned to the field of the record type.

data_type
*data_type* specifies the data type of *field_name*.

**DEFAULT default_value**

The **DEFAULT** clause assigns a default data value for the corresponding field. The data type of the default expression must match the data type of the column. If no default is specified, then the default is **NULL**.

A **record variable** or simply put, a **record**, is an instance of a record type. A record is declared from a record type. The properties of the record such as its field names and types are inherited from the record type.

The following is the syntax for a record declaration.

```
record rectype
```

*record* is an identifier assigned to the record variable. *rectype* is the identifier of a previously defined record type. Once declared, a record can then be used to hold data.

Dot notation is used to make reference to the fields in the record.

```
record.field
```

*record* is a previously declared record variable and *field* is the identifier of a field belonging to the record type from which *record* is defined.

The **emp_sal_query** is again modified – this time using a user-defined record type and record variable.

```sql
CREATE OR REPLACE PROCEDURE emp_sal_query ( p_empno IN emp.empno%TYPE ) IS
    TYPE emp_typ IS RECORD (
        ename emp.ename%TYPE,
        job emp.job%TYPE,
        hiredate emp.hiredate%TYPE,
        sal emp.sal%TYPE,
        deptno emp.deptno%TYPE
    );
    r_emp emp_typ;
    v_avgsal emp.sal%TYPE;
BEGIN
    SELECT ename, job, hiredate, sal, deptno
    INTO r_emp.ename, r_emp.job, r_emp.hiredate, r_emp.sal, r_emp.deptno
    FROM emp WHERE empno = p_empno;
    DBMS_OUTPUT.PUT_LINE('Employee # : ' || p_empno);
    DBMS_OUTPUT.PUT_LINE('Name       : ' || r_emp.ename);
    DBMS_OUTPUT.PUT_LINE('Job        : ' || r_emp.job);
    DBMS_OUTPUT.PUT_LINE('Hire Date  : ' || r_emp.hiredate);
    DBMS_OUTPUT.PUT_LINE('Salary     : ' || r_emp.sal);
    DBMS_OUTPUT.PUT_LINE('Dept #     : ' || r_emp.deptno);
END;
```
SELECT AVG(sal) INTO v_avgsal
FROM emp WHERE deptno = r_emp.deptno;
IF r_emp.sal > v_avgsal THEN
    DBMS_OUTPUT.PUT_LINE('Employee''s salary is more than the ' || 'department average of ' || v_avgsal);
ELSE
    DBMS_OUTPUT.PUT_LINE('Employee''s salary does not exceed the ' || 'department average of ' || v_avgsal);
END IF;
END;

Note that instead of specifying data type names, the %TYPE attribute can be used for the field data types in the record type definition.

The following is the output from executing this stored procedure.

EXEC emp_sal_query(7698);
Employee # : 7698
Name       : BLAKE
Job        : MANAGER
Hire Date  : 01-MAY-81 00:00:00
Salary     : 2850.00
Dept #     : 30
Employee's salary is more than the department average of 1566.67
4.4 Basic Statements

This section begins the discussion of the programming statements that can be used in an SPL program.
4.4.1 NULL

The simplest statement is the NULL statement. This statement is an executable statement that does nothing.

    NULL;

The following is the simplest, possible valid SPL program.

```
BEGIN
  NULL;
END;
```

The NULL statement can act as a placeholder where an executable statement is required such as in a branch of an IF-THEN-ELSE statement.

For example:

```
CREATE OR REPLACE PROCEDURE divide_it (  
  p_numerator     IN  NUMBER,  
  p_denominator   IN  NUMBER,  
  p_result        OUT NUMBER  
)  
IS  
BEGIN  
  IF p_denominator = 0 THEN
    NULL;
  ELSE
    p_result := p_numerator / p_denominator;
  END IF;
END;
```


4.4.2 Assignment

The assignment statement sets a variable or a formal parameter of mode `OUT` or `IN OUT` specified on the left side of the assignment, `:=`, to the evaluated expression specified on the right side of the assignment.

```
variable := expression;
```

*variable* is an identifier for a previously declared variable, `OUT` formal parameter, or `IN OUT` formal parameter.

*expression* is an expression that produces a single value. The value produced by the expression must have a compatible data type with that of *variable*.

The following example shows the typical use of assignment statements in the executable section of the procedure.

```
CREATE OR REPLACE PROCEDURE dept_salary_rpt (p_deptno NUMBER)
IS
  todays_date DATE;
  rpt_title VARCHAR2(60);
  base_sal INTEGER;
  base_comm_rate NUMBER;
  base_annual NUMBER;
BEGIN
  todays_date := SYSDATE;
  rpt_title := 'Report For Department # ' || p_deptno || ' on ' || todays_date;
  base_sal := 35525;
  base_comm_rate := 1.33333;
  base_annual := ROUND(base_sal * base_comm_rate, 2);
  DBMS_OUTPUT.PUT_LINE(rpt_title);
  DBMS_OUTPUT.PUT_LINE('Base Annual Salary: ' || base_annual);
END;
```
4.4.3 SELECT INTO

The `SELECT INTO` statement is an SPL variation of the SQL `SELECT` command, the differences being:

- That `SELECT INTO` is designed to assign the results to variables or records where they can then be used in SPL program statements.
- The accessible result set of `SELECT INTO` is at most one row.

Other than the above, all of the clauses of the `SELECT` command such as `WHERE`, `ORDER BY`, `GROUP BY`, `HAVING`, etc. are valid for `SELECT INTO`. The following are the two variations of `SELECT INTO`.

```
SELECT select_expressions INTO target FROM ...;
```

`target` is a comma-separated list of simple variables. `select_expressions` and the remainder of the statement are the same as for the `SELECT` command. The selected values must exactly match in data type, number, and order the structure of the target or a runtime error occurs.

```
SELECT * INTO record FROM table ...;
```

`record` is a record variable that has previously been declared.

If the query returns zero rows, null values are assigned to the target(s). If the query returns multiple rows, the first row is assigned to the target(s) and the rest are discarded. (Note that "the first row" is not well-defined unless you’ve used `ORDER BY`.)

**Note:** In either cases, where no row is returned or more than one row is returned, SPL throws an exception.

**Note:** There is a variation of `SELECT INTO` using the `BULK COLLECT` clause that allows a result set of more than one row that is returned into a collection. See Section 4.12.4.1 for more information on using the `BULK COLLECT` clause with the `SELECT INTO` statement.

You can use the `WHEN NO_DATA_FOUND` clause in an `EXCEPTION` block to determine whether the assignment was successful (that is, at least one row was returned by the query).

This version of the `emp_sal_query` procedure uses the variation of `SELECT INTO` that returns the result set into a record. Also note the addition of the `EXCEPTION` block containing the `WHEN NO_DATA_FOUND` conditional expression.
CREATE OR REPLACE PROCEDURE emp_sal_query (  
    p_empno         IN emp.empno%TYPE  
) IS  
    r_emp           emp%ROWTYPE;  
    v_avgsal        emp.sal%TYPE;  
BEGIN  
    SELECT * INTO r_emp  
        FROM emp WHERE empno = p_empno;  
    DBMS_OUTPUT.PUT_LINE('Employee # : ' || p_empno);  
    DBMS_OUTPUT.PUT_LINE('Name       : ' || r_emp.ename);  
    DBMS_OUTPUT.PUT_LINE('Job        : ' || r_emp.job);  
    DBMS_OUTPUT.PUT_LINE('Hire Date  : ' || r_emp.hiredate);  
    DBMS_OUTPUT.PUT_LINE('Salary     : ' || r_emp.sal);  
    DBMS_OUTPUT.PUT_LINE('Dept #     : ' || r_emp.deptno);  
    SELECT AVG(sal) INTO v_avgsal  
        FROM emp WHERE deptno = r_emp.deptno;  
    IF r_emp.sal > v_avgsal THEN  
        DBMS_OUTPUT.PUT_LINE('Employee''s salary is more than the '  
                    || 'department average of ' || v_avgsal);  
    ELSE  
        DBMS_OUTPUT.PUT_LINE('Employee''s salary does not exceed the '  
                    || 'department average of ' || v_avgsal);  
    END IF;  
EXCEPTION  
    WHEN NO_DATA_FOUND THEN  
        DBMS_OUTPUT.PUT_LINE('Employee # ' || p_empno || ' not found');  
END;  

If the query is executed with a non-existent employee number the results appear as follows.

EXEC emp_sal_query(0);  
Employee # 0 not found

Another conditional clause of use in the EXCEPTION section with SELECT INTO is the TOO_MANY_ROWS exception. If more than one row is selected by the SELECT INTO statement an exception is thrown by SPL.

When the following block is executed, the TOO_MANY_ROWS exception is thrown since there are many employees in the specified department.

DECLARE  
    v_ename         emp.ename%TYPE;  
BEGIN  
    SELECT ename INTO v_ename FROM emp WHERE deptno = 20 ORDER BY ename;  
EXCEPTION  
    WHEN TOO_MANY_ROWS THEN  
        DBMS_OUTPUT.PUT_LINE('More than one employee found');  
        DBMS_OUTPUT.PUT_LINE('First employee returned is ' || v_ename);  
END;  

More than one employee found  
First employee returned is ADAMS

Note: See Section 4.5.7 or more information on exception handling.
4.4.4 INSERT

The INSERT command available in the SQL language can also be used in SPL programs.

An expression in the SPL language can be used wherever an expression is allowed in the SQL INSERT command. Thus, SPL variables and parameters can be used to supply values to the insert operation.

The following is an example of a procedure that performs an insert of a new employee using data passed from a calling program.

```sql
CREATE OR REPLACE PROCEDURE emp_insert (  
  p_empno         IN emp.empno%TYPE,  
  p_ename         IN emp.ename%TYPE,  
  p_job           IN emp.job%TYPE,  
  p_mgr           IN emp.mgr%TYPE,  
  p_hiredate      IN emp.hiredate%TYPE,  
  p_sal           IN emp.sal%TYPE,  
  p_comm          IN emp.comm%TYPE,  
  p_deptno        IN emp.deptno%TYPE  
)  
IS  
BEGIN  
  INSERT INTO emp VALUES (  
    p_empno,  
    p_ename,  
    p_job,  
    p_mgr,  
    p_hiredate,  
    p_sal,  
    p_comm,  
    p_deptno);  
  DBMS_OUTPUT.PUT_LINE('Added employee...');  
  DBMS_OUTPUT.PUT_LINE('Employee # : ' || p_empno);  
  DBMS_OUTPUT.PUT_LINE('Name       : ' || p_ename);  
  DBMS_OUTPUT.PUT_LINE('Job        : ' || p_job);  
  DBMS_OUTPUT.PUT_LINE('Manager    : ' || p_mgr);  
  DBMS_OUTPUT.PUT_LINE('Hire Date  : ' || p_hiredate);  
  DBMS_OUTPUT.PUT_LINE('Salary     : ' || p_sal);  
  DBMS_OUTPUT.PUT_LINE('Commission : ' || p_comm);  
  DBMS_OUTPUT.PUT_LINE('Dept #     : ' || p_deptno);  
  DBMS_OUTPUT.PUT_LINE('----------------------');  
EXCEPTION  
  WHEN OTHERS THEN  
    DBMS_OUTPUT.PUT_LINE('OTHERS exception on INSERT of employee # ' || p_empno);  
    DBMS_OUTPUT.PUT_LINE('SQLCODE : ' || SQLCODE);  
    DBMS_OUTPUT.PUT_LINE('SQLERRM : ' || SQLERRM);  
END;
```

If an exception occurs all database changes made in the procedure are automatically rolled back. In this example the EXCEPTION section with the WHEN OTHERS clause catches all exceptions. Two variables are displayed. SQLCODE is a number that identifies
the specific exception that occurred. SQLERRM is a text message explaining the error. See Section 4.5.7 for more information on exception handling.

The following shows the output when this procedure is executed.

```sql
EXEC emp_insert(9503,'PETERSON', 'ANALYST', 7902, '31-MAR-05', 5000, NULL, 40);
```

```
Added employee...
Employee # : 9503
Name : PETERSON
Job : ANALYST
Manager : 7902
Hire Date : 31-MAR-05 00:00:00
Salary : 5000
Dept # : 40

SELECT * FROM emp WHERE empno = 9503;
```

```
<table>
<thead>
<tr>
<th>empno</th>
<th>ename</th>
<th>job</th>
<th>mgr</th>
<th>hiredate</th>
<th>sal</th>
<th>comm</th>
<th>deptno</th>
</tr>
</thead>
<tbody>
<tr>
<td>9503</td>
<td>PETERSON</td>
<td>ANALYST</td>
<td>7902</td>
<td>31-MAR-05 00:00:00</td>
<td>5000.00</td>
<td></td>
<td>40</td>
</tr>
</tbody>
</table>
```

Note: The `INSERT` command can be included in a `FORALL` statement. A `FORALL` statement allows a single `INSERT` command to insert multiple rows from values supplied in one or more collections. See Section 4.12.3 for more information on the `FORALL` statement.
4.4.5 UPDATE

The UPDATE command available in the SQL language can also be used in SPL programs.

An expression in the SPL language can be used wherever an expression is allowed in the SQL UPDATE command. Thus, SPL variables and parameters can be used to supply values to the update operation.

```sql
CREATE OR REPLACE PROCEDURE emp_comp_update (p_empno IN emp.empno%TYPE,
p_sal IN emp.sal%TYPE,
p_comm IN emp.comm%TYPE)
IS
BEGIN
    UPDATE emp SET sal = p_sal, comm = p_comm WHERE empno = p_empno;
    IF SQL%FOUND THEN
        DBMS_OUTPUT.PUT_LINE('Updated Employee # : ' || p_empno);
        DBMS_OUTPUT.PUT_LINE('New Salary         : ' || p_sal);
        DBMS_OUTPUT.PUT_LINE('New Commission     : ' || p_comm);
    ELSE
        DBMS_OUTPUT.PUT_LINE('Employee # ' || p_empno || ' not found');
    END IF;
END;
```

The SQL%FOUND conditional expression returns TRUE if a row is updated, FALSE otherwise. See Section 4.4.8 for a discussion of SQL%FOUND and other similar expressions.

The following shows the update on the employee using this procedure.

```sql
EXEC emp_comp_update(9503, 6540, 1200);
```

Updated Employee # : 9503
New Salary         : 6540
New Commission     : 1200

SELECT * FROM emp WHERE empno = 9503;

<table>
<thead>
<tr>
<th>empno</th>
<th>ename</th>
<th>job</th>
<th>mgr</th>
<th>hiredate</th>
<th>sal</th>
<th>comm</th>
<th>deptno</th>
</tr>
</thead>
<tbody>
<tr>
<td>9503</td>
<td>PETERSON</td>
<td>ANALYST</td>
<td>7902</td>
<td>31-MAR-05</td>
<td>6540.00</td>
<td>1200.00</td>
<td>40</td>
</tr>
</tbody>
</table>

(1 row)

**Note:** The UPDATE command can be included in a FORALL statement. A FORALL statement allows a single UPDATE command to update multiple rows from values supplied in one or more collections. See Section 4.12.3 for more information on the FORALL statement.
4.4.6 DELETE

The DELETE command (available in the SQL language) can also be used in SPL programs.

An expression in the SPL language can be used wherever an expression is allowed in the SQL DELETE command. Thus, SPL variables and parameters can be used to supply values to the delete operation.

```sql
CREATE OR REPLACE PROCEDURE emp_delete (  
p_empno         IN emp.empno%TYPE  
)  
IS  
BEGIN  
    DELETE FROM emp WHERE empno = p_empno;  
    IF SQL%FOUND THEN  
        DBMS_OUTPUT.PUT_LINE('Deleted Employee # : ' || p_empno);  
    ELSE  
        DBMS_OUTPUT.PUT_LINE('Employee # ' || p_empno || ' not found');  
    END IF;  
END;
```

The SQL%FOUND conditional expression returns TRUE if a row is deleted, FALSE otherwise. See Section 4.4.8 for a discussion of SQL%FOUND and other similar expressions.

The following shows the deletion of an employee using this procedure.

```sql
EXEC emp_delete(9503);  
Deleted Employee # : 9503  
SELECT * FROM emp WHERE empno = 9503;  
empno | ename | job | mgr | hiredate | sal | comm | deptno  
-------+-------+-----+-----+----------+-----+-------+---------  
(0 rows)
```

**Note:** The DELETE command can be included in a FORALL statement. A FORALL statement allows a single DELETE command to delete multiple rows from values supplied in one or more collections. See Section 4.12.3 for more information on the FORALL statement.
4.4.7 Using the RETURNING INTO Clause

The INSERT, UPDATE, and DELETE commands may be appended by the optional RETURNING INTO clause. This clause allows the SPL program to capture the newly added, modified, or deleted values from the results of an INSERT, UPDATE, or DELETE command, respectively.

The following is the syntax.

    { insert | update | delete }
    RETURNING { * | expr_1 [, expr_2 ] ...}
    INTO { record | field_1 [, field_2 ] ...};

*insert* is a valid INSERT command. *update* is a valid UPDATE command. *delete* is a valid DELETE command. If * is specified, then the values from the row affected by the INSERT, UPDATE, or DELETE command are made available for assignment to the record or fields to the right of the INTO keyword. (Note that the use of * is a Postgres Plus Advanced Server extension and is not Oracle compatible.) *expr_1, expr_2...* are expressions evaluated upon the row affected by the INSERT, UPDATE, or DELETE command. The evaluated results are assigned to the record or fields to the right of the INTO keyword. *record* is the identifier of a record that must contain fields that match in number and order, and are data type compatible with the values in the RETURNING clause. *field_1, field_2,...* are variables that must match in number and order, and are data type compatible with the set of values in the RETURNING clause.

If the INSERT, UPDATE, or DELETE command returns a result set with more than one row, then an exception is thrown with SQLCODE 01422, query returned more than one row. If no rows are in the result set, then the variables following the INTO keyword are set to null.

**Note:** There is a variation of RETURNING INTO using the BULK COLLECT clause that allows a result set of more than one row that is returned into a collection.

The following example is a modification of the emp_comp_update procedure introduced in Section 4.4.5, with the addition of the RETURNING INTO clause.

```sql
CREATE OR REPLACE PROCEDURE emp_comp_update (
    p_empno         IN emp.empno%TYPE,
    p_sal           IN emp.sal%TYPE,
    p_comm
) IS
    v_empno         emp.empno%TYPE;
    v_ename         emp.ename%TYPE;
    v_job           emp.job%TYPE;
    v_sal           emp.sal%TYPE;
    v_comm          emp.comm%TYPE;
```
v_deptno emp.deptno%TYPE;
BEGIN
  UPDATE emp SET sal = p_sal, comm = p_comm WHERE empno = p_empno
  RETURNING empno, ename, job, sal, comm, deptno
  INTO v_empno, v_ename, v_job, v_sal, v_comm, v_deptno;
  IF SQL%FOUND THEN
    DBMS_OUTPUT.PUT_LINE('Updated Employee # : ' || v_empno);
    DBMS_OUTPUT.PUT_LINE('Name               : ' || v_ename);
    DBMS_OUTPUT.PUT_LINE('Job                : ' || v_job);
    DBMS_OUTPUT.PUT_LINE('Department         : ' || v_deptno);
    DBMS_OUTPUT.PUT_LINE('New Salary         : ' || v_sal);
    DBMS_OUTPUT.PUT_LINE('New Commission     : ' || v_comm);
  ELSE
    DBMS_OUTPUT.PUT_LINE('Employee # ' || p_empno || ' not found');
  END IF;
END;

The following is the output from this procedure (assuming employee 9503 created by the emp_insert procedure still exists within the table).

EXEC emp_comp_update(9503, 6540, 1200);

Updated Employee # : 9503
Name               : PETERSON
Job                : ANALYST
Department         : 40
New Salary         : 6540.00
New Commission     : 1200.00

The following example is a modification of the emp_delete procedure with the addition of the RETURNING INTO clause using record types.

CREATE OR REPLACE PROCEDURE emp_delete ( p_empno IN emp.empno%TYPE ) IS
  r_emp emp%ROWTYPE;
BEGIN
  DELETE FROM emp WHERE empno = p_empno
  RETURNING *
  INTO r_emp;
  IF SQL%FOUND THEN
    DBMS_OUTPUT.PUT_LINE('Deleted Employee # : ' || r_emp.empno);
    DBMS_OUTPUT.PUT_LINE('Name               : ' || r_emp.ename);
  END IF;
END;
The following is the output from this procedure.

EXEC emp_delete(9503);

Deleted Employee # : 9503
Name               : PETERSON
Job                : ANALYST
Manager            : 7902
Hire Date          : 31-MAR-05 00:00:00
Salary             : 6540.00
Commission         : 1200.00
Department         : 40
4.4.8 Obtaining the Result Status

There are several attributes that can be used to determine the effect of a command. SQL%FOUND is a Boolean that returns TRUE if at least one row was affected by an INSERT, UPDATE or DELETE command or a SELECT INTO command retrieved one or more rows.

The following anonymous block inserts a row and then displays the fact that the row has been inserted.

```
BEGIN
    INSERT INTO emp (empno,ename,job,sal,deptno) VALUES (9001, 'JONES', 'CLERK', 850.00, 40);
    IF SQL%FOUND THEN
        DBMS_OUTPUT.PUT_LINE('Row has been inserted');
    END IF;
END;
```

Row has been inserted

SQL%ROWCOUNT provides the number of rows affected by an INSERT, UPDATE or DELETE command. The following example updates the row that was just inserted and displays SQL%ROWCOUNT.

```
BEGIN
    UPDATE emp SET hiredate = '03-JUN-07' WHERE empno = 9001;
    DBMS_OUTPUT.PUT_LINE('# rows updated: ' || SQL%ROWCOUNT);
END;
```

# rows updated: 1

SQL%NOTFOUND is the opposite of SQL%FOUND. SQL%NOTFOUND returns TRUE if no rows were affected by an INSERT, UPDATE or DELETE command or a SELECT INTO command retrieved no rows.

```
BEGIN
    UPDATE emp SET hiredate = '03-JUN-07' WHERE empno = 9000;
    IF SQL%NOTFOUND THEN
        DBMS_OUTPUT.PUT_LINE('No rows were updated');
    END IF;
END;
```

No rows were updated
4.5 Control Structures

The programming statements in SPL that make it a full procedural complement to SQL are described in this section.
4.5.1 IF Statement

IF statements let you execute commands based on certain conditions. SPL has four forms of IF:

- IF ... THEN
- IF ... THEN ... ELSE
- IF ... THEN ... ELSE IF
- IF ... THEN ... ELSIF ... THEN ... ELSE

4.5.1.1 IF-THEN

IF boolean-expression THEN
  statements
END IF;

IF-THEN statements are the simplest form of IF. The statements between THEN and END IF will be executed if the condition is TRUE. Otherwise, they are skipped.

In the following example an IF-THEN statement is used to test and display employees who have a commission.

DECLARE
  v_empno     emp.empno%TYPE;
  v_comm      emp.comm%TYPE;
  CURSOR emp_cursor IS SELECT empno, comm FROM emp;
BEGIN
  OPEN emp_cursor;
  DBMS_OUTPUT.PUT_LINE('EMPNO    COMM');
  DBMS_OUTPUT.PUT_LINE('-----    ------');
  LOOP
    FETCH emp_cursor INTO v_empno, v_comm;
    EXIT WHEN emp_cursor%NOTFOUND;
    -- Test whether or not the employee gets a commission
    IF v_comm IS NOT NULL AND v_comm > 0 THEN
      DBMS_OUTPUT.PUT_LINE(v_empno || ' ' || TO_CHAR(v_comm,'$99999.99'));
    END IF;
  END LOOP;
  CLOSE emp_cursor;
END;

The following is the output from this program.

EMPNO  COMM
-----  ------
74999  $300.00
7521   $500.00
7654   $1400.00
4.5.1.2 IF-THEN-ELSE

IF boolean-expression THEN
  statements
ELSE
  statements
END IF;

IF-THEN-ELSE statements add to IF-THEN by letting you specify an alternative set of
statements that should be executed if the condition evaluates to false.

The previous example is modified so an IF-THEN-ELSE statement is used to display the
text Non-commission if the employee does not get a commission.

DECLARE
  v_empno         emp.empno%TYPE;
  v_comm          emp.comm%TYPE;
CURSOR emp_cursor IS SELECT empno, comm FROM emp;
BEGIN
  OPEN emp_cursor;
  DBMS_OUTPUT.PUT_LINE('EMPNO    COMM');
  DBMS_OUTPUT.PUT_LINE('-----    -------');
  LOOP
    FETCH emp_cursor INTO v_empno, v_comm;
    EXIT WHEN emp_cursor%NOTFOUND;
    --
    -- Test whether or not the employee gets a commission
    --
    IF v_comm IS NOT NULL AND v_comm > 0 THEN
      DBMS_OUTPUT.PUT_LINE(v_empno || '  ' ||
                            TO_CHAR(v_comm,'$99999.99'));
    ELSE
      DBMS_OUTPUT.PUT_LINE(v_empno || '     ' || 'Non-commission');
    END IF;
  END LOOP;
  CLOSE emp_cursor;
END;

The following is the output from this program.

EMPNO    COMM
-----    -------
7369     Non-commission
7499     $   300.00
7521     $   500.00
7566     Non-commission
7654     $  1400.00
7698     Non-commission
7782     Non-commission
7788     Non-commission
7839     Non-commission
7844     Non-commission
7876     Non-commission
7900     Non-commission
7902     Non-commission
7934     Non-commission
4.5.1.3 IF-THEN-ELSE IF

IF statements can be nested so that alternative IF statements can be invoked once it is determined whether or not the conditional of an outer IF statement is TRUE or FALSE.

In the following example the outer IF-THEN-ELSE statement tests whether or not an employee has a commission. The inner IF-THEN-ELSE statements then test whether the employee’s total compensation exceeds or is less than the company average.

DECLARE
    v_empno         emp.empno%TYPE;
    v_sal          emp.sal%TYPE;
    v_comm          emp.comm%TYPE;
    v_avg          NUMBER(7,2);
CURSOR emp_cursor IS SELECT empno, sal, comm FROM emp;
BEGIN
    -- Calculate the average yearly compensation in the company
    --
    SELECT AVG((sal + NVL(comm,0)) * 24) INTO v_avg FROM emp;
    DBMS_OUTPUT.PUT_LINE('Average Yearly Compensation: ' ||
        TO_CHAR(v_avg,'$999,999.99'));
    OPEN emp_cursor;
    DBMS_OUTPUT.PUT_LINE('EMPNO    YEARLY COMP');
    DBMS_OUTPUT.PUT_LINE('------    -----------');
    LOOP
        FETCH emp_cursor INTO v_empno, v_sal, v_comm;
        EXIT WHEN emp_cursor%NOTFOUND;
        -- Test whether or not the employee gets a commission
        --
        IF v_comm IS NOT NULL AND v_comm > 0 THEN
            -- Test if the employee's compensation with commission exceeds the average
            IF (v_sal + v_comm) * 24 > v_avg THEN
                DBMS_OUTPUT.PUT_LINE(v_empno || ' ' ||
                    TO_CHAR((v_sal + v_comm) * 24,'$999,999.99') ||
                    ' Exceeds Average');
            ELSE
                DBMS_OUTPUT.PUT_LINE(v_empno || ' ' ||
                    TO_CHAR((v_sal + v_comm) * 24,'$999,999.99') ||
                    ' Below Average');
            END IF;
        ELSE
            -- Test if the employee's compensation without commission exceeds the average
            IF v_sal * 24 > v_avg THEN
                DBMS_OUTPUT.PUT_LINE(v_empno || ' ' ||
                    TO_CHAR(v_sal * 24,'$999,999.99') ||
                    ' Exceeds Average');
            ELSE
                DBMS_OUTPUT.PUT_LINE(v_empno || ' ' ||
                    TO_CHAR(v_sal * 24,'$999,999.99') ||
                    ' Below Average');
            END IF;
        END IF;
    END LOOP;
    CLOSE emp_cursor;
Note: The logic in this program can be simplified considerably by calculating the employee’s yearly compensation using the `NVL` function within the `SELECT` command of the cursor declaration, however, the purpose of this example is to demonstrate how `IF` statements can be used.

The following is the output from this program.

```
Average Yearly Compensation: $ 53,528.57
EMPNO    YEARLY COMP
-----    -------------
7369    $ 19,200.00 Below Average
7499    $ 45,600.00 Below Average
7521    $ 42,000.00 Below Average
7566    $ 71,400.00 Exceeds Average
7654    $ 63,600.00 Exceeds Average
7698    $ 68,400.00 Exceeds Average
7728    $ 58,800.00 Exceeds Average
7788    $ 72,000.00 Exceeds Average
7839    $120,000.00 Exceeds Average
7844    $ 36,000.00 Below Average
7876    $ 26,400.00 Below Average
7900    $ 22,800.00 Below Average
7902    $ 72,000.00 Exceeds Average
7934    $ 31,200.00 Below Average
```

When you use this form, you are actually nesting an `IF` statement inside the `ELSE` part of an outer `IF` statement. Thus you need one `END IF` statement for each nested `IF` and one for the parent `IF-ELSE`.

**4.5.1.4 IF-THEN-ELSIF-ELSE**

```
IF boolean-expression THEN
  statements
[ ELSIF boolean-expression THEN
  statements
[ ELSIF boolean-expression THEN
  statements ] ...]
[ ELSE
  statements ]
END IF;
```

`IF-THEN-ELSIF-ELSE` provides a method of checking many alternatives in one statement. Formally it is equivalent to nested `IF-THEN-ELSE-IF-THEN` commands, but only one `END IF` is needed.

The following example uses an `IF-THEN-ELSIF-ELSE` statement to count the number of employees by compensation ranges of $25,000.
 DECLARE
  v_lt_25K       SMALLINT  := 0;
  v_25K_50K      SMALLINT  := 0;
  v_50K_75K      SMALLINT  := 0;
  v_75K_100K     SMALLINT  := 0;
  v_ge_100K      SMALLINT  := 0;
  CURSOR emp_cursor IS SELECT empno, (sal + NVL(comm,0)) * 24 FROM emp;
BEGIN
  OPEN emp_cursor;
  LOOP
    FETCH emp_cursor INTO v_empno, v_comp;
    EXIT WHEN emp_cursor%NOTFOUND;
    IF v_comp < 25000 THEN
      v_lt_25K := v_lt_25K + 1;
    ELSIF v_comp < 50000 THEN
      v_25K_50K := v_25K_50K + 1;
    ELSIF v_comp < 75000 THEN
      v_50K_75K := v_50K_75K + 1;
    ELSIF v_comp < 100000 THEN
      v_75K_100K := v_75K_100K + 1;
    ELSE
      v_ge_100K := v_ge_100K + 1;
    END IF;
  END LOOP;
  CLOSE emp_cursor;
  DBMS_OUTPUT.PUT_LINE('Number of employees by yearly compensation');
  DBMS_OUTPUT.PUT_LINE('Less than 25,000 : ' || v_lt_25K);
  DBMS_OUTPUT.PUT_LINE('25,000 - 49,999 : ' || v_25K_50K);
  DBMS_OUTPUT.PUT_LINE('50,000 - 74,999 : ' || v_50K_75K);
  DBMS_OUTPUT.PUT_LINE('75,000 - 99,999 : ' || v_75K_100K);
  DBMS_OUTPUT.PUT_LINE('100,000 and over : ' || v_ge_100K);
END;

The following is the output from this program.

Number of employees by yearly compensation
Less than 25,000 : 2
25,000 - 49,999 : 5
50,000 - 74,999 : 6
75,000 - 99,999 : 0
100,000 and over : 1
4.5.2 RETURN Statement

The RETURN statement terminates the current function, procedure or anonymous block and returns control to the caller.

There are two forms of the RETURN Statement. The first form of the RETURN statement is used to terminate a procedure or function that returns void. The syntax of the first form is:

RETURN;

The second form of RETURN returns a value to the caller. The syntax of the second form of the RETURN statement is:

RETURN expression;

expression must evaluate to the same data type as the return type of the function.

The following example uses the RETURN statement returns a value to the caller:

```
CREATE OR REPLACE FUNCTION emp_comp ( 
    p_sal           NUMBER, 
    p_comm          NUMBER 
) RETURN NUMBER 
IS 
BEGIN 
    RETURN (p_sal + NVL(p_comm, 0)) * 24; 
END emp_comp;
```
4.5.3 GOTO Statement

The GOTO statement causes the point of execution to jump to the statement with the specified label. The syntax of a GOTO statement is:

```
GOTO label
```

`label` is a name assigned to an executable statement. `label` must be unique within the scope of the function, procedure or anonymous block.

To label a statement, use the syntax:

```
<<label>> statement
```

`statement` is the point of execution that the program jumps to.

You can label assignment statements, any SQL statement (like `INSERT`, `UPDATE`, `CREATE`, etc.) and selected procedural language statements. The procedural language statements that can be labeled are:

- IF
- EXIT
- RETURN
- RAISE
- EXECUTE
- PERFORM
- GET DIAGNOSTICS
- OPEN
- FETCH
- MOVE
- CLOSE
- NULL
- COMMIT
- ROLLBACK
- GOTO
- CASE
- LOOP
- WHILE
- FOR

Please note that `exit` is considered a keyword, and cannot be used as the name of a label.

GOTO statements cannot transfer control into a conditional block or sub-block, but can transfer control from a conditional block or sub-block.
The following example verifies that an employee record contains a name, job description, and employee hire date; if any piece of information is missing, a GOTO statement transfers the point of execution to a statement that prints a message that the employee is not valid.

```sql
CREATE OR REPLACE PROCEDURE verify_emp (  
p_empno NUMBER
)  
IS  
v_ename emp.ename%TYPE;  
v_job emp.job%TYPE;  
v_hiredate emp.hiredate%TYPE;  
BEGIN  
  SELECT ename, job, hiredate  
    INTO v_ename, v_job, v_hiredate FROM emp  
    WHERE empno = p_empno;  
  IF v_ename IS NULL THEN  
    GOTO invalid_emp;  
  END IF;  
  IF v_job IS NULL THEN  
    GOTO invalid_emp;  
  END IF;  
  IF v_hiredate IS NULL THEN  
    GOTO invalid_emp;  
  END IF;  
  DBMS_OUTPUT.PUT_LINE('Employee ' || p_empno || ' validated without errors.');  
  RETURN;  
<<invalid_emp>> DBMS_OUTPUT.PUT_LINE('Employee ' || p_empno || ' is not a valid employee.');  
END;
```

GOTO statements have the following restrictions:

- A **GOTO** statement cannot jump to a declaration.

- A **GOTO** statement cannot transfer control to another function or procedure.

- A **label** should not be placed at the end of a block, function or procedure.
4.5.4 CASE Expression

The CASE expression returns a value that is substituted where the CASE expression is located within an expression.

There are two formats of the CASE expression - one that is called a searched CASE and the other that uses a selector.

4.5.4.1 Selector CASE Expression

The selector CASE expression attempts to match an expression called the selector to the expression specified in one or more WHEN clauses. result is an expression that is type-compatible in the context where the CASE expression is used. If a match is found, the value given in the corresponding THEN clause is returned by the CASE expression. If there are no matches, the value following ELSE is returned. If ELSE is omitted, the CASE expression returns null.

```
CASE selector-expression
    WHEN match-expression THEN
        result
    [ WHEN match-expression THEN
        result
    [ WHEN match-expression THEN
        result ] ...
    [ ELSE
        result ]
END;
```

match-expression is evaluated in the order in which it appears within the CASE expression. result is an expression that is type-compatible in the context where the CASE expression is used. When the first match-expression is encountered that equals selector-expression, result in the corresponding THEN clause is returned as the value of the CASE expression. If none of match-expression equals selector-expression then result following ELSE is returned. If no ELSE is specified, the CASE expression returns null.

The following example uses a selector CASE expression to assign the department name to a variable based upon the department number.

```sql
DECLARE
    v_empno    emp.empno%TYPE;
    v_ename    emp.ename%TYPE;
    v_deptno   emp.deptno%TYPE;
    v_dname    dept.dname%TYPE;
BEGIN
    CURSOR emp_cursor IS SELECT empno, ename, deptno FROM emp;
    OPEN emp_cursor;
```
The following is the output from this program.

<table>
<thead>
<tr>
<th>EMPNO</th>
<th>ENAME</th>
<th>DEPTNO</th>
<th>DNAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>7369</td>
<td>SMITH</td>
<td>20</td>
<td>Research</td>
</tr>
<tr>
<td>7499</td>
<td>ALLEN</td>
<td>30</td>
<td>Sales</td>
</tr>
<tr>
<td>7521</td>
<td>WARD</td>
<td>30</td>
<td>Sales</td>
</tr>
<tr>
<td>7566</td>
<td>JONES</td>
<td>20</td>
<td>Research</td>
</tr>
<tr>
<td>7654</td>
<td>MARTIN</td>
<td>30</td>
<td>Sales</td>
</tr>
<tr>
<td>7698</td>
<td>BLAKE</td>
<td>30</td>
<td>Sales</td>
</tr>
<tr>
<td>7782</td>
<td>CLARK</td>
<td>10</td>
<td>Accounting</td>
</tr>
<tr>
<td>7788</td>
<td>SCOTT</td>
<td>20</td>
<td>Research</td>
</tr>
<tr>
<td>7839</td>
<td>KING</td>
<td>10</td>
<td>Accounting</td>
</tr>
<tr>
<td>7844</td>
<td>TURNER</td>
<td>30</td>
<td>Sales</td>
</tr>
<tr>
<td>7876</td>
<td>ADAMS</td>
<td>20</td>
<td>Research</td>
</tr>
<tr>
<td>7900</td>
<td>JAMES</td>
<td>30</td>
<td>Sales</td>
</tr>
<tr>
<td>7902</td>
<td>FORD</td>
<td>20</td>
<td>Research</td>
</tr>
<tr>
<td>7934</td>
<td>MILLER</td>
<td>10</td>
<td>Accounting</td>
</tr>
</tbody>
</table>

### 4.5.4.2 Searched CASE Expression

A searched CASE expression uses one or more Boolean expressions to determine the resulting value to return.

```
CASE WHEN boolean-expression THEN
    result
[ WHEN boolean-expression THEN
    result
[ WHEN boolean-expression THEN
    result ] ...]
[ ELSE
    result ]
END;
```

*boolean-expression* is evaluated in the order in which it appears within the CASE expression. *result* is an expression that is type-compatible in the context where the CASE expression is used. When the first *boolean-expression* is encountered that
evaluates to \texttt{TRUE}, \texttt{result} in the corresponding \texttt{THEN} clause is returned as the value of the \texttt{CASE} expression. If none of \texttt{boolean-expression} evaluates to true then \texttt{result} following \texttt{ELSE} is returned. If no \texttt{ELSE} is specified, the \texttt{CASE} expression returns null.

The following example uses a searched \texttt{CASE} expression to assign the department name to a variable based upon the department number.

DECLARE
  v_empno         emp.empno%TYPE;
  v_ename         emp.ename%TYPE;
  v_deptno        emp.deptno%TYPE;
  v_dname         dept.dname%TYPE;
CURSOR emp_cursor IS SELECT empno, ename, deptno FROM emp;
BEGIN
  OPEN emp_cursor;
  DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME     DEPTNO    DNAME');
  DBMS_OUTPUT.PUT_LINE('----- ------ ------ ----------');
  LOOP
    FETCH emp_cursor INTO v_empno, v_ename, v_deptno;
    EXIT WHEN emp_cursor%NOTFOUND;
    v_dname :=
      CASE
        WHEN v_deptno = 10 THEN 'Accounting'
        WHEN v_deptno = 20 THEN 'Research'
        WHEN v_deptno = 30 THEN 'Sales'
        WHEN v_deptno = 40 THEN 'Operations'
        ELSE 'unknown'
      END;
    DBMS_OUTPUT.PUT_LINE(v_empno || '     ' || RPAD(v_ename, 10) ||
                         '  ' || v_deptno || '      ' || v_dname);
  END LOOP;
  CLOSE emp_cursor;
END;

The following is the output from this program.

<table>
<thead>
<tr>
<th>EMPNO</th>
<th>ENAME</th>
<th>DEPTNO</th>
<th>DNAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>7369</td>
<td>SMITH</td>
<td>20</td>
<td>Research</td>
</tr>
<tr>
<td>7499</td>
<td>ALLEN</td>
<td>30</td>
<td>Sales</td>
</tr>
<tr>
<td>7521</td>
<td>WARD</td>
<td>30</td>
<td>Sales</td>
</tr>
<tr>
<td>7566</td>
<td>JONES</td>
<td>20</td>
<td>Research</td>
</tr>
<tr>
<td>7654</td>
<td>MARTIN</td>
<td>30</td>
<td>Sales</td>
</tr>
<tr>
<td>7698</td>
<td>BLAKE</td>
<td>30</td>
<td>Sales</td>
</tr>
<tr>
<td>7782</td>
<td>CLARK</td>
<td>10</td>
<td>Accounting</td>
</tr>
<tr>
<td>7788</td>
<td>SCOTT</td>
<td>20</td>
<td>Research</td>
</tr>
<tr>
<td>7839</td>
<td>KING</td>
<td>10</td>
<td>Accounting</td>
</tr>
<tr>
<td>7844</td>
<td>TURNER</td>
<td>30</td>
<td>Sales</td>
</tr>
<tr>
<td>7876</td>
<td>ADAMS</td>
<td>20</td>
<td>Research</td>
</tr>
<tr>
<td>7900</td>
<td>JAMES</td>
<td>30</td>
<td>Sales</td>
</tr>
<tr>
<td>7902</td>
<td>FORD</td>
<td>20</td>
<td>Research</td>
</tr>
<tr>
<td>7934</td>
<td>MILLER</td>
<td>10</td>
<td>Accounting</td>
</tr>
</tbody>
</table>
4.5.5 CASE Statement

The CASE statement executes a set of one or more statements when a specified search condition is TRUE. The CASE statement is a stand-alone statement in itself while the previously discussed CASE expression must appear as part of an expression.

There are two formats of the CASE statement - one that is called a searched CASE and the other that uses a selector.

4.5.5.1 Selector CASE Statement

The selector CASE statement attempts to match an expression called the selector to the expression specified in one or more WHEN clauses. When a match is found one or more corresponding statements are executed.

```
CASE selector-expression
  WHEN match-expression THEN
    statements
  [ WHEN match-expression THEN
    statements
  [ WHEN match-expression THEN
    statements ] ...]
  [ ELSE
    statements ]
END CASE;
```

selector-expression returns a value type-compatible with each match-expression. match-expression is evaluated in the order in which it appears within the CASE statement. statements are one or more SPL statements, each terminated by a semi-colon. When the value of selector-expression equals the first match-expression, the statement(s) in the corresponding THEN clause are executed and control continues following the END CASE keywords. If there are no matches, the statement(s) following ELSE are executed. If there are no matches and there is no ELSE clause, an exception is thrown.

The following example uses a selector CASE statement to assign a department name and location to a variable based upon the department number.

```sql
DECLARE
  v_empno         emp.empno%TYPE;
  v_ename         emp.ename%TYPE;
  v_deptno        emp.deptno%TYPE;
  v_dname         dept.dname%TYPE;
  v_loc           dept.loc%TYPE;
CURSOR emp_cursor IS SELECT empno, ename, deptno FROM emp;
BEGIN
```
OPEN emp_cursor;
DBMS_OUTPUT.PUT_LINE('EMPNO ENAME DEPTNO DNAME LOC' || '-----' || '--------' || '----------' || '------');
LOOPLOOP
FETCH emp_cursor INTO v_empno, v_ename, v_deptno;
EXIT WHEN emp_cursor%NOTFOUND;
CASE v_deptno
    WHEN 10 THEN  v_dname := 'Accounting';
                  v_loc   := 'New York';
WHEN 20 THEN  v_dname := 'Research';
                  v_loc   := 'Dallas';
WHEN 30 THEN  v_dname := 'Sales';
                  v_loc   := 'Chicago';
WHEN 40 THEN  v_dname := 'Operations';
                  v_loc   := 'Boston';
ELSE v_dname := 'unknown';
                  v_loc   := '';END CASE;DBMS_OUTPUT.PUT_LINE(v_empno || '     ' || RPAD(v_ename, 10) || '  ' || v_deptno || '     ' || RPAD(v_dname, 14) || ' ' || v_loc);
END LOOP;
CLOSE emp_cursor;END;

The following is the output from this program.

<table>
<thead>
<tr>
<th>EMPNO</th>
<th>ENAME</th>
<th>DEPTNO</th>
<th>DNAME</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>7369</td>
<td>SMITH</td>
<td>20</td>
<td>Research</td>
<td>Dallas</td>
</tr>
<tr>
<td>7499</td>
<td>ALLEN</td>
<td>30</td>
<td>Sales</td>
<td>Chicago</td>
</tr>
<tr>
<td>7521</td>
<td>WARD</td>
<td>30</td>
<td>Sales</td>
<td>Chicago</td>
</tr>
<tr>
<td>7566</td>
<td>JONES</td>
<td>20</td>
<td>Research</td>
<td>Dallas</td>
</tr>
<tr>
<td>7654</td>
<td>MARTIN</td>
<td>30</td>
<td>Sales</td>
<td>Chicago</td>
</tr>
<tr>
<td>7698</td>
<td>BLAKE</td>
<td>30</td>
<td>Sales</td>
<td>Chicago</td>
</tr>
<tr>
<td>7782</td>
<td>CLARK</td>
<td>10</td>
<td>Accounting</td>
<td>New York</td>
</tr>
<tr>
<td>7788</td>
<td>SCOTT</td>
<td>20</td>
<td>Research</td>
<td>Dallas</td>
</tr>
<tr>
<td>7839</td>
<td>KING</td>
<td>10</td>
<td>Accounting</td>
<td>New York</td>
</tr>
<tr>
<td>7844</td>
<td>TURNER</td>
<td>30</td>
<td>Sales</td>
<td>Chicago</td>
</tr>
<tr>
<td>7876</td>
<td>ADAMS</td>
<td>20</td>
<td>Research</td>
<td>Dallas</td>
</tr>
<tr>
<td>7900</td>
<td>JAMES</td>
<td>30</td>
<td>Sales</td>
<td>Chicago</td>
</tr>
<tr>
<td>7902</td>
<td>FORD</td>
<td>20</td>
<td>Research</td>
<td>Dallas</td>
</tr>
<tr>
<td>7934</td>
<td>MILLER</td>
<td>10</td>
<td>Accounting</td>
<td>New York</td>
</tr>
</tbody>
</table>

4.5.5.2 Searched CASE statement

A searched CASE statement uses one or more Boolean expressions to determine the resulting set of statements to execute.

    CASE WHEN boolean-expression THEN
    statements
    [ WHEN boolean-expression THEN
    statements
    [ WHEN boolean-expression THEN
    statements ] ...]
    [ ELSE

Copyright © 2007 - 2015 EnterpriseDB Corporation. All rights reserved.
statements ]
END CASE;

boolean-expression is evaluated in the order in which it appears within the CASE statement. When the first boolean-expression is encountered that evaluates to TRUE, the statement(s) in the corresponding THEN clause are executed and control continues following the END CASE keywords. If none of boolean-expression evaluates to TRUE, the statement(s) following ELSE are executed. If none of boolean-expression evaluates to TRUE and there is no ELSE clause, an exception is thrown.

The following example uses a searched CASE statement to assign a department name and location to a variable based upon the department number.

```sql
DECLARE
  v_empno         emp.empno%TYPE;
  v_ename         emp.ename%TYPE;
  v_deptno        emp.deptno%TYPE;
  v_dname         dept.dname%TYPE;
  v_loc           dept.loc%TYPE;
CURSOR emp_cursor IS SELECT empno, ename, deptno FROM emp;
BEGIN
  OPEN emp_cursor;
  DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME     DEPTNO    DNAME     LOC');
  DBMS_OUTPUT.PUT_LINE('-----    -------    ------    --------    -----');
  LOOP
    FETCH emp_cursor INTO v_empno, v_ename, v_deptno;
    EXIT WHEN emp_cursor%NOTFOUND;
    CASE
      WHEN v_deptno = 10 THEN v_dname := 'Accounting';
        v_loc := 'New York';
      WHEN v_deptno = 20 THEN v_dname := 'Research';
        v_loc := 'Dallas';
      WHEN v_deptno = 30 THEN v_dname := 'Sales';
        v_loc := 'Chicago';
      WHEN v_deptno = 40 THEN v_dname := 'Operations';
        v_loc := 'Boston';
      ELSE v_dname := 'unknown';
        v_loc := '';
    END CASE;
    DBMS_OUTPUT.PUT_LINE(v_empno || '     ' || RPAD(v_ename, 10) || '  ' || v_deptno || '      ' || RPAD(v_dname, 14) || ' ' || v_loc);
  END LOOP;
  CLOSE emp_cursor;
END;
```

The following is the output from this program.

<table>
<thead>
<tr>
<th>EMPNO</th>
<th>ENAME</th>
<th>DEPTNO</th>
<th>DNAME</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>7369</td>
<td>SMITH</td>
<td>20</td>
<td>Research</td>
<td>Dallas</td>
</tr>
<tr>
<td>7499</td>
<td>ALLEN</td>
<td>30</td>
<td>Sales</td>
<td>Chicago</td>
</tr>
<tr>
<td>7521</td>
<td>WARD</td>
<td>30</td>
<td>Sales</td>
<td>Chicago</td>
</tr>
<tr>
<td>7566</td>
<td>JONES</td>
<td>20</td>
<td>Research</td>
<td>Dallas</td>
</tr>
<tr>
<td>7654</td>
<td>MARTIN</td>
<td>30</td>
<td>Sales</td>
<td>Chicago</td>
</tr>
<tr>
<td>ID</td>
<td>Name</td>
<td>Age</td>
<td>Department</td>
<td>Location</td>
</tr>
<tr>
<td>-----</td>
<td>-------</td>
<td>-----</td>
<td>------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>7698</td>
<td>BLAKE</td>
<td>30</td>
<td>Sales</td>
<td>Chicago</td>
</tr>
<tr>
<td>7782</td>
<td>CLARK</td>
<td>10</td>
<td>Accounting</td>
<td>New York</td>
</tr>
<tr>
<td>7788</td>
<td>SCOTT</td>
<td>20</td>
<td>Research</td>
<td>Dallas</td>
</tr>
<tr>
<td>7839</td>
<td>KING</td>
<td>10</td>
<td>Accounting</td>
<td>New York</td>
</tr>
<tr>
<td>7844</td>
<td>TURNER</td>
<td>30</td>
<td>Sales</td>
<td>Chicago</td>
</tr>
<tr>
<td>7876</td>
<td>ADAMS</td>
<td>20</td>
<td>Research</td>
<td>Dallas</td>
</tr>
<tr>
<td>7900</td>
<td>JAMES</td>
<td>30</td>
<td>Sales</td>
<td>Chicago</td>
</tr>
<tr>
<td>7902</td>
<td>FORD</td>
<td>20</td>
<td>Research</td>
<td>Dallas</td>
</tr>
<tr>
<td>7934</td>
<td>MILLER</td>
<td>10</td>
<td>Accounting</td>
<td>New York</td>
</tr>
</tbody>
</table>
4.5.6 Loops

With the LOOP, EXIT, CONTINUE, WHILE, and FOR statements, you can arrange for your SPL program to repeat a series of commands.

4.5.6.1 LOOP

```
LOOP
  statements
END LOOP;
```

LOOP defines an unconditional loop that is repeated indefinitely until terminated by an EXIT or RETURN statement.

4.5.6.2 EXIT

```
EXIT [ WHEN expression ];
```

The innermost loop is terminated and the statement following END LOOP is executed next.

If WHEN is present, loop exit occurs only if the specified condition is TRUE, otherwise control passes to the statement after EXIT.

EXIT can be used to cause early exit from all types of loops; it is not limited to use with unconditional loops.

The following is a simple example of a loop that iterates ten times and then uses the EXIT statement to terminate.

```
DECLARE
  v_counter       NUMBER(2);
BEGIN
  v_counter := 1;
  LOOP
    EXIT WHEN v_counter > 10;
    DBMS_OUTPUT.PUT_LINE('Iteration # ' || v_counter);
    v_counter := v_counter + 1;
  END LOOP;
END;
```

The following is the output from this program.

```
Iteration # 1
Iteration # 2
Iteration # 3
Iteration # 4
Iteration # 5
Iteration # 6
Iteration # 7
```
4.5.6.3 CONTINUE

The CONTINUE statement provides a way to proceed with the next iteration of a loop while skipping intervening statements.

When the CONTINUE statement is encountered, the next iteration of the innermost loop is begun, skipping all statements following the CONTINUE statement until the end of the loop. That is, control is passed back to the loop control expression, if any, and the body of the loop is re-evaluated.

If the WHEN clause is used, then the next iteration of the loop is begun only if the specified expression in the WHEN clause evaluates to TRUE. Otherwise, control is passed to the next statement following the CONTINUE statement.

The CONTINUE statement may not be used outside of a loop.

The following is a variation of the previous example that uses the CONTINUE statement to skip the display of the odd numbers.

```sql
DECLARE
    v_counter       NUMBER(2);
BEGIN
    v_counter := 0;
    LOOP
        v_counter := v_counter + 1;
        EXIT WHEN v_counter > 10;
        CONTINUE WHEN MOD(v_counter,2) = 1;
        DBMS_OUTPUT.PUT_LINE('Iteration # ' || v_counter);
    END LOOP;
END;
```

The following is the output from above program.

```
Iteration # 2
Iteration # 4
Iteration # 6
Iteration # 10
```

4.5.6.4 WHILE

```
WHILE expression LOOP
    statements
END LOOP;
```

The WHILE statement repeats a sequence of statements so long as the condition expression evaluates to TRUE. The condition is checked just before each entry to the loop body.
The following example contains the same logic as in the previous example except the `WHILE` statement is used to take the place of the `EXIT` statement to determine when to exit the loop.

**Note:** The conditional expression used to determine when to exit the loop must be altered. The `EXIT` statement terminates the loop when its conditional expression is true. The `WHILE` statement terminates (or never begins the loop) when its conditional expression is false.

```sql
DECLARE
  v_counter       NUMBER(2);
BEGIN
  v_counter := 1;
  WHILE v_counter <= 10 LOOP
    DBMS_OUTPUT.PUT_LINE('Iteration # ' || v_counter);
    v_counter := v_counter + 1;
  END LOOP;
END;
```

The same result is generated by this example as in the prior example.

```
Iteration # 1
Iteration # 2
Iteration # 3
Iteration # 4
Iteration # 5
Iteration # 6
Iteration # 7
Iteration # 8
Iteration # 9
Iteration # 10
```

## 4.5.6.5 FOR (integer variant)

```sql
FOR name IN [REVERSE] expression .. expression LOOP
  statements
END LOOP;
```

This form of `FOR` creates a loop that iterates over a range of integer values. The variable `name` is automatically defined as type `INTEGER` and exists only inside the loop. The two expressions giving the loop range are evaluated once when entering the loop. The iteration step is +1 and `name` begins with the value of `expression` to the left of `..` and terminates once `name` exceeds the value of `expression` to the right of `..` Thus the two expressions take on the following roles: `start-value .. end-value`.

The optional `REVERSE` clause specifies that the loop should iterate in reverse order. The first time through the loop, `name` is set to the value of the right-most `expression`; the loop terminates when the `name` is less than the left-most `expression`.

The following example simplifies the `WHILE` loop example even further by using a `FOR` loop that iterates from 1 to 10.
BEGIN
  FOR i IN 1 .. 10 LOOP
    DBMS_OUTPUT.PUT_LINE('Iteration # ' || i);
  END LOOP;
END;

Here is the output using the **FOR** statement.

Iteration # 1
Iteration # 2
Iteration # 3
Iteration # 4
Iteration # 5
Iteration # 6
Iteration # 7
Iteration # 8
Iteration # 9
Iteration # 10

If the start value is greater than the end value the loop body is not executed at all. No error is raised as shown by the following example.

BEGIN
  FOR i IN 10 .. 1 LOOP
    DBMS_OUTPUT.PUT_LINE('Iteration # ' || i);
  END LOOP;
END;

There is no output from this example as the loop body is never executed.

**Note**: SPL also supports **CURSOR FOR** loops (see Section 4.8.7).
4.5.7 Exception Handling

By default, any error occurring in an SPL program aborts execution of the program. You can trap errors and recover from them by using a BEGIN block with an EXCEPTION section. The syntax is an extension of the normal syntax for a BEGIN block:

```
[ DECLARE
    declarations ]
BEGIN
    statements
EXCEPTION
    WHEN condition [ OR condition ]... THEN
        handler_statements
    [ WHEN condition [ OR condition ]... THEN
        handler_statements ]...
END;
```

If no error occurs, this form of block simply executes all the statements, and then control passes to the next statement after END. If an error occurs within the statements, further processing of the statements is abandoned, and control passes to the EXCEPTION list. The list is searched for the first condition matching the error that occurred. If a match is found, the corresponding handler_statements are executed, and then control passes to the next statement after END. If no match is found, the error propagates out as though the EXCEPTION clause were not there at all. The error can be caught by an enclosing block with EXCEPTION; if there is no enclosing block, it aborts processing of the subprogram.

The special condition name OTHERS matches every error type. Condition names are not case-sensitive.

If a new error occurs within the selected handler_statements, it cannot be caught by this EXCEPTION clause, but is propagated out. A surrounding EXCEPTION clause could catch it.

The following table lists the condition names that may be used:

Table 4-4-2 Exception Condition Names

<table>
<thead>
<tr>
<th>Condition Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CASE_NOT_FOUND</td>
<td>The application has encountered a situation where none of the cases in a CASE statement evaluates to TRUE and there is no ELSE condition.</td>
</tr>
<tr>
<td>COLLECTION_IS_NULL</td>
<td>The application has attempted to invoke a collection method on a null collection such as an uninitialized nested table.</td>
</tr>
<tr>
<td>CURSOR_ALREADY_OPEN</td>
<td>The application has attempted to open a cursor that is already open.</td>
</tr>
<tr>
<td>Condition Name</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>DUP_VAL_ON_INDEX</td>
<td>The application has attempted to store a duplicate value that currently exists within a constrained column.</td>
</tr>
<tr>
<td>INVALID_CURSOR</td>
<td>The application has attempted to access an unopened cursor.</td>
</tr>
<tr>
<td>INVALID_NUMBER</td>
<td>The application has attempted to convert a string to a numeric literal that cannot be converted (applicable only to SQL statements).</td>
</tr>
<tr>
<td>NO_DATA_FOUND</td>
<td>No rows satisfy the selection criteria.</td>
</tr>
<tr>
<td>OTHERS</td>
<td>The application has encountered an exception that hasn’t been caught by a prior condition in the exception section.</td>
</tr>
<tr>
<td>SUBSCRIPT_BEYOND_COUNT</td>
<td>The application has attempted to reference a subscript of a nested table or varray beyond its initialized or extended size.</td>
</tr>
<tr>
<td>SUBSCRIPT_OUTSIDE_LIMIT</td>
<td>The application has attempted to reference a subscript or extend a varray beyond its maximum size limit.</td>
</tr>
<tr>
<td>TOO_MANY_ROWS</td>
<td>The application has encountered more than one row that satisfies the selection criteria (where only one row is allowed to be returned).</td>
</tr>
<tr>
<td>VALUE_ERROR</td>
<td>The application has tried to convert a string to a numeric literal that cannot be converted (applicable only to procedural statements).</td>
</tr>
<tr>
<td>ZERO_DIVIDE</td>
<td>The application has tried to divide by zero.</td>
</tr>
<tr>
<td>User-defined Exception</td>
<td>See Section 4.5.8</td>
</tr>
</tbody>
</table>
4.5.8 User-defined Exceptions

Any number of errors (referred to in PL/SQL as exceptions) can occur during program execution. When an exception is thrown, normal execution of the program stops, and control of the program transfers to the error-handling portion of the program. An exception may be a pre-defined error that is generated by the server, or may be a logical error that raises a user-defined exception.

User-defined exceptions are never raised by the server; they are raised explicitly by a RAISE statement.

A user-defined exception is raised when a developer-defined logical rule is broken; a common example of a logical rule being broken occurs when a check is presented against an account with insufficient funds. An attempt to cash a check against an account with insufficient funds will provoke a user-defined exception.

You can define exceptions in functions, procedures, packages or anonymous blocks. While you cannot declare the same exception twice in the same block, you can declare the same exception in two different blocks.

Before implementing a user-defined exception, you must declare the exception in the declaration section of a function, procedure, package or anonymous block. You can then raise the exception using the RAISE statement:

```
DECLARE
    exception_name EXCEPTION;
BEGIN
    ...
    RAISE exception_name;
    ...
END;
```

```
exception_name
```

is the name of the exception.

Unhandled exceptions propagate back through the call stack. If the exception remains unhandled, the exception is eventually reported to the client application.

User-defined exceptions declared in a block are considered to be local to that block, and global to any nested blocks within the block. To reference an exception that resides in an outer block, you must assign a label to the outer block; then, preface the name of the exception with the block name:

```
block_name.exception_name
```
Conversely, outer blocks cannot reference exceptions declared in nested blocks.

The scope of a declaration is limited to the block in which it is declared unless it is created in a package, and when referenced, qualified by the package name. For example, to raise an exception named `out_of_stock` that resides in a package named `inventory_control` a program must raise an error named:

```
inventory_control.out_of_stock
```

The following example demonstrates declaring a user-defined exception in a package. The user-defined exception does not require a package-qualifier when it is raised in `check_balance`, since it resides in the same package as the exception:

```
CREATE OR REPLACE PACKAGE ar AS
  overdrawn EXCEPTION;
  PROCEDURE check_balance(p_balance NUMBER, p_amount NUMBER);
END;

CREATE OR REPLACE PACKAGE BODY ar AS
  PROCEDURE check_balance(p_balance NUMBER, p_amount NUMBER)
  IS
    BEGIN
      IF (p_amount > p_balance) THEN
        RAISE overdrawn;
      END IF;
  END;

The following procedure (purchase) calls the check_balance procedure. If p_amount is greater than p_balance, check_balance raises an exception; purchase catches the ar.overdrawn exception. purchase must refer to the exception with a package-qualified name (ar.overdrawn) because purchase is not defined within the ar package.

```
CREATE PROCEDURE purchase(customerID INT, amount NUMERIC)
AS
  BEGIN
    ar.check_balance(getcustomerbalance(customerid), amount);
    record_purchase(customerid, amount);
    exception
      WHEN ar.overdrawn THEN
        raise_credit_limit(customerid, amount*1.5);
  END;
```

When ar.check_balance raises an exception, execution jumps to the exception handler defined in purchase:

```
exception
  WHEN ar.overdrawn THEN
    raise_credit_limit(customerid, amount*1.5);
```
The exception handler raises the customer’s credit limit and ends. When the exception handler ends, execution resumes with the statement that follows `ar.check_balance`.
4.5.9 PRAGMA EXCEPTION_INIT

PRAGMA EXCEPTION_INIT associates a user-defined error code with an exception. A PRAGMA EXCEPTION_INIT declaration may be included in any block, sub-block or package. You can only assign an error code to an exception (using PRAGMA EXCEPTION_INIT) after declaring the exception. The format of a PRAGMA EXCEPTION_INIT declaration is:

```
PRAGMA EXCEPTION_INIT(exception_name, {exception_number | exception_code})
```

Where:

- `exception_name` is the name of the associated exception.
- `exception_number` is a user-defined error code associated with the pragma. If you specify an unmapped `exception_number`, the server will return a warning.
- `exception_code` is the name of a pre-defined exception. For a complete list of valid exceptions, see the Postgres core documentation, available at the EnterpriseDB website at:

```
```

The previous section (User-defined Exceptions) included an example that demonstrates declaring a user-defined exception in a package. The following example uses the same basic structure, but adds a PRAGMA EXCEPTION_INIT declaration:

```
CREATE OR REPLACE PACKAGE ar AS
  overdrawn EXCEPTION;
  PROCEDURE check_balance(p_balance NUMBER, p_amount NUMBER);
END;

CREATE OR REPLACE PACKAGE BODY ar AS
  PROCEDURE check_balance(p_balance NUMBER, p_amount NUMBER)
  IS
    BEGIN
      IF (p_amount > p_balance) THEN
        RAISE overdrawn;
      END IF;
    END;
END;
```

The following procedure (purchase) calls the check_balance procedure. If `p_amount` is greater than `p_balance`, check_balance raises an exception; purchase catches the `ar.overdrawn` exception.
CREATE PROCEDURE purchase(customerID int, amount NUMERIC) AS
BEGIN
    ar.check_balance(getcustomerbalance(customerid), amount);
    record_purchase(customerid, amount);
EXCEPTION
    WHEN ar.overdrawn THEN
        DBMS_OUTPUT.PUT_LINE ('This account is overdrawn.');</n        DBMS_OUTPUT.PUT_LINE ('SQLCode :'||SQLCODE||' '||SQLERRM);
END;

When ar.check_balance raises an exception, execution jumps to the exception handler defined in purchase.

EXCEPTION
    WHEN ar.overdrawn THEN
        DBMS_OUTPUT.PUT_LINE ('This account is overdrawn.');</n        DBMS_OUTPUT.PUT_LINE ('SQLCode :'||SQLCODE||' '||SQLERRM);

The exception handler returns an error message, followed by SQLCODE information:

This account is overdrawn.
SQLCODE: -20100 User-Defined Exception

The following example demonstrates using a pre-defined exception. The code creates a more meaningful name for the no_data_found exception; if the given customer does not exist, the code catches the exception, calls DBMS_OUTPUT.PUT_LINE to report the error, and then re-raises the original exception:

CREATE OR REPLACE PACKAGE ar AS
    overdrawn EXCEPTION;
    PRAGMA EXCEPTION_INIT (unknown_customer, no_data_found);
    PROCEDURE check_balance(p_customer_id NUMBER);
END;

CREATE OR REPLACE PACKAGE BODY ar AS
    PROCEDURE check_balance(p_customer_id NUMBER) IS
        DECLARE
            v_balance NUMBER;
        BEGIN
            SELECT balance INTO v_balance FROM customer
            WHERE cust_id = p_customer_id;
            EXCEPTION WHEN unknown_customer THEN
                DBMS_OUTPUT.PUT_LINE('invalid customer id');
                RAISE;
            END;
        END;
4.5.10 RAISE_APPLICATION_ERROR

The procedure, RAISE_APPLICATION_ERROR, allows a developer to intentionally abort processing within an SPL program from which it is called by causing an exception. The exception is handled in the same manner as described in Section 4.5.7. In addition, the RAISE_APPLICATION_ERROR procedure makes a user-defined code and error message available to the program which can then be used to identify the exception.

RAISE_APPLICATION_ERROR(error_number, message);

Where:

error_number is an integer value or expression that is returned in a variable named SQLCODE when the procedure is executed. error_number must be a value between -20000 and -20999.

message is a string literal or expression that is returned in a variable named SQLERRM.

For additional information on the SQLCODE and SQLERRM variables, see Section 4.13, Errors and Messages.

The following example uses the RAISE_APPLICATION_ERROR procedure to display a different code and message depending upon the information missing from an employee.

```sql
CREATE OR REPLACE PROCEDURE verify_emp (p_empno NUMBER) IS
  v_ename emp.ename%TYPE;
  v_job emp.job%TYPE;
  v_mgr emp.mgr%TYPE;
  v_hiredate emp.hiredate%TYPE;
BEGIN
  SELECT ename, job, mgr, hiredate
  INTO v_ename, v_job, v_mgr, v_hiredate FROM emp
  WHERE empno = p_empno;
  IF v_ename IS NULL THEN
    RAISE_APPLICATION_ERROR(-20010, 'No name for ' || p_empno);
  END IF;
  IF v_job IS NULL THEN
    RAISE_APPLICATION_ERROR(-20020, 'No job for ' || p_empno);
  END IF;
  IF v_mgr IS NULL THEN
    RAISE_APPLICATION_ERROR(-20030, 'No manager for ' || p_empno);
  END IF;
  IF v_hiredate IS NULL THEN
    RAISE_APPLICATION_ERROR(-20040, 'No hire date for ' || p_empno);
  END IF;
  DBMS_OUTPUT.PUT_LINE('Employee ' || p_empno || ' validated without errors');
EXCEPTION```

Copyright © 2007 - 2015 EnterpriseDB Corporation. All rights reserved.
The following shows the output in a case where the manager number is missing from an employee record.

```sql
EXEC verify_emp(7839);
SQLCODE: -20030
SQLERRM: EDB-20030: No manager for 7839
```
4.6 Transaction Control

There may be circumstances where it is desired that all updates to a database are to occur successfully, or none are to occur at all if any error occurs. A set of database updates that are to all occur successfully as a single unit, or are not to occur at all, is said to be a transaction.

A common example in banking is a funds transfer between two accounts. The two parts of the transaction are the withdrawal of funds from one account, and the deposit of the funds in another account. Both parts of this transaction must occur otherwise the bank’s books will be out of balance. The deposit and withdrawal are one transaction.

An SPL application can be created that uses an Oracle compatible style of transaction control if the following conditions are met:

- The `edb_stmt_level_tx` parameter must be set to `TRUE`. This prevents the action of unconditionally rolling back all database updates within the `BEGIN/END` block if any exception occurs. See Section 1.3.3 for more information on the `edb_stmt_level_tx` parameter.
- The application must not be running in autocommit mode. If autocommit mode is on, each successful database update is immediately committed and cannot be undone. The manner in which autocommit mode is turned on or off is application dependent.

A transaction begins when the first SQL command is encountered in the SPL program. All subsequent SQL commands are included as part of that transaction. The transaction ends when one of the following occurs:

- An unhandled exception occurs in which case the effects of all database updates made during the transaction are rolled back and the transaction is aborted.
- A `COMMIT` command is encountered in which case the effect of all database updates made during the transaction become permanent.
- A `ROLLBACK` command is encountered in which case the effects of all database updates made during the transaction are rolled back and the transaction is aborted. If a new SQL command is encountered, a new transaction begins.
- Control returns to the calling application (such as Java, PSQL, etc.) in which case the action of the application determines whether the transaction is committed or rolled back.

Note: Unlike Oracle, DDL commands such as `CREATE TABLE` do not implicitly occur within their own transaction. Therefore, DDL commands do not automatically cause an immediate database commit as in Oracle, and DDL commands may be rolled back just like DML commands.
A transaction may span one or more BEGIN/END blocks, or a single BEGIN/END block may contain one or more transactions.

The following sections discuss the COMMIT and ROLLBACK commands in more detail.

### 4.6.1 COMMIT

The COMMIT command makes all database updates made during the current transaction permanent, and ends the current transaction.

```
COMMIT [ WORK ];
```

The COMMIT command may be used within anonymous blocks, stored procedures, or functions. Within an SPL program, it may appear in the executable section and/or the exception section.

In the following example, the third INSERT command in the anonymous block results in an error. The effect of the first two INSERT commands are retained as shown by the first SELECT command. Even after issuing a ROLLBACK command, the two rows remain in the table as shown by the second SELECT command verifying that they were indeed committed.

**Note:** The `edb_stmt_level_tx` configuration parameter shown in the example below can be set for the entire database using the `ALTER DATABASE` command, or it can be set for the entire database server by changing it in the `postgresql.conf` file.

```sql
\set AUTOCOMMIT off
SET edb_stmt_level_tx TO on;
BEGIN
    INSERT INTO dept VALUES (50, 'FINANCE', 'DALLAS');
    INSERT INTO dept VALUES (60, 'MARKETING', 'CHICAGO');
    COMMIT;
    INSERT INTO dept VALUES (70, 'HUMAN RESOURCES', 'CHICAGO');
EXCEPTION
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('SQLERRM: ' || SQLERRM);
        DBMS_OUTPUT.PUT_LINE('SQLCODE: ' || SQLCODE);
END;
```

```
SQLERRM: value too long for type character varying(14)  SQLCODE: 22001
```

```
SELECT * FROM dept;
```

<table>
<thead>
<tr>
<th>deptno</th>
<th>dname</th>
<th>loc</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>ACCOUNTING</td>
<td>NEW YORK</td>
</tr>
<tr>
<td>20</td>
<td>RESEARCH</td>
<td>DALLAS</td>
</tr>
<tr>
<td>30</td>
<td>SALES</td>
<td>CHICAGO</td>
</tr>
<tr>
<td>40</td>
<td>OPERATIONS</td>
<td>BOSTON</td>
</tr>
<tr>
<td>50</td>
<td>FINANCE</td>
<td>DALLAS</td>
</tr>
<tr>
<td>60</td>
<td>MARKETING</td>
<td>CHICAGO</td>
</tr>
</tbody>
</table>
4.6.2 ROLLBACK

The ROLLBACK command undoes all database updates made during the current transaction, and ends the current transaction.

    ROLLBACK [ WORK ];

The ROLLBACK command may be used within anonymous blocks, stored procedures, or functions. Within an SPL program, it may appear in the executable section and/or the exception section.

In the following example, the exception section contains a ROLLBACK command. Even though the first two INSERT commands are executed successfully, the third results in an exception that results in the rollback of all the INSERT commands in the anonymous block.

```sql
\set AUTOCOMMIT off
SET edb_stmt_level_tx TO on;
BEGIN
    INSERT INTO dept VALUES (50, 'FINANCE', 'DALLAS');
    INSERT INTO dept VALUES (60, 'MARKETING', 'CHICAGO');
    INSERT INTO dept VALUES (70, 'HUMAN RESOURCES', 'CHICAGO');
EXCEPTION
    WHEN OTHERS THEN
        ROLLBACK;
        DBMS_OUTPUT.PUT_LINE('SQLERRM: ' || SQLERRM);
        DBMS_OUTPUT.PUT_LINE('SQLCODE: ' || SQLCODE);
END;
SELECT * FROM dept;
```
The following is a more complex example using both `COMMIT` and `ROLLBACK`. First, the following stored procedure is created which inserts a new employee.

```sql
\set AUTOCOMMIT off
SET edb_stmt_level_tx TO on;

CREATE OR REPLACE PROCEDURE emp_insert (
  p_empno         IN emp.empno%TYPE,
  p_ename         IN emp.ename%TYPE,
  p_job           IN emp.job%TYPE,
  p_mgr           IN emp.mgr%TYPE,
  p_hiredate      IN emp.hiredate%TYPE,
  p_sal           IN emp.sal%TYPE,
  p_comm          IN emp.comm%TYPE,
  p_deptno        IN emp.deptno%TYPE
) IS
BEGIN
  INSERT INTO emp VALUES (
    p_empno,
    p_ename,
    p_job,
    p_mgr,
    p_hiredate,
    p_sal,
    p_comm,
    p_deptno);
  DBMS_OUTPUT.PUT_LINE('Added employee...');
  DBMS_OUTPUT.PUT_LINE('Employee # : ' || p_empno);  
  DBMS_OUTPUT.PUT_LINE('Name       : ' || p_ename);
  DBMS_OUTPUT.PUT_LINE('Job        : ' || p_job);
  DBMS_OUTPUT.PUT_LINE('Manager    : ' || p_mgr);
  DBMS_OUTPUT.PUT_LINE('Hire Date  : ' || p_hiredate);
  DBMS_OUTPUT.PUT_LINE('Salary     : ' || p_sal);
  DBMS_OUTPUT.PUT_LINE('Commission : ' || p_comm);
  DBMS_OUTPUT.PUT_LINE('Dept #     : ' || p_deptno);
  DBMS_OUTPUT.PUT_LINE('----------------------');
END;
```

Note that this procedure has no exception section so any error that may occur is propagated up to the calling program.

The following anonymous block is run. Note the use of the `COMMIT` command after all calls to the `emp_insert` procedure and the `ROLLBACK` command in the exception section.

```sql
BEGIN
  emp_insert(9601,'FARRELL','ANALYST',7902,'03-MAR-08',5000,NULL,40);
  emp_insert(9602,'TYLER','ANALYST',7900,'25-JAN-08',4800,NULL,40);
  COMMIT;
EXCEPTION
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('SQLERRM: ' || SQLERRM);
    DBMS_OUTPUT.PUT_LINE('An error occurred - roll back inserts');
```

Copyright © 2007 - 2015 EnterpriseDB Corporation. All rights reserved.
ROLLBACK;
END;

Added employee...
Employee # : 9601
Name       : FARRELL
Job        : ANALYST
Manager    : 7902
Hire Date  : 03-MAR-08 00:00:00
Salary     : 5000
Commission :
Dept #     : 40
--------------

Added employee...
Employee # : 9602
Name       : TYLER
Job        : ANALYST
Manager    : 7900
Hire Date  : 25-JAN-08 00:00:00
Salary     : 4800
Commission :
Dept #     : 40
--------------

The following SELECT command shows that employees Farrell and Tyler were successfully added.

```
SELECT * FROM emp WHERE empno > 9600;
```

```
empno | ename  |   job   | mgr  |      hiredate      |   sal   | comm | deptno
-------|--------|--------|------|-------------------|--------|------|-------
 9601 | FARRELL | ANALYST | 7902 | 03-MAR-08 00:00:00 | 5000.00 |      |     40
 9602 | TYLER   | ANALYST | 7900 | 25-JAN-08 00:00:00 | 4800.00 |      |     40
```

(2 rows)

Now, execute the following anonymous block:

```
BEGIN
    emp_insert(9603,'HARRISON','SALESMAN',7902,'13-DEC-07',5000,3000,20);
    emp_insert(9604,'JARVIS','SALESMAN',7902,'05-MAY-08',4800,4100,11);
    COMMIT;
EXCEPTION
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('SQLERRM: ' || SQLERRM);
        DBMS_OUTPUT.PUT_LINE('An error occurred - roll back inserts');
        ROLLBACK;
END;
```

Added employee...
Employee # : 9603
Name       : HARRISON
Job        : SALESMAN
Manager    : 7902
Hire Date  : 13-DEC-07 00:00:00
Salary     : 5000
Commission : 3000
Dept #     : 20
--------------

SQLERRM: insert or update on table "emp" violates foreign key constraint "emp_ref_dept_fk"
An error occurred - roll back inserts
A SELECT command run against the table yields the following:

```sql
SELECT * FROM emp WHERE empno > 9600;
```

<table>
<thead>
<tr>
<th>empno</th>
<th>ename</th>
<th>job</th>
<th>mgr</th>
<th>hiredate</th>
<th>sal</th>
<th>comm</th>
<th>deptno</th>
</tr>
</thead>
<tbody>
<tr>
<td>9601</td>
<td>FARRELL</td>
<td>ANALYST</td>
<td>7902</td>
<td>03-MAR-08 00:00:00</td>
<td>5000.00</td>
<td></td>
<td>40</td>
</tr>
<tr>
<td>9602</td>
<td>TYLER</td>
<td>ANALYST</td>
<td>7900</td>
<td>25-JAN-08 00:00:00</td>
<td>4800.00</td>
<td></td>
<td>40</td>
</tr>
</tbody>
</table>

(2 rows)

The ROLLBACK command in the exception section successfully undoes the insert of employee Harrison. Also note that employees Farrell and Tyler are still in the table as their inserts were made permanent by the COMMIT command in the first anonymous block.
4.7 Dynamic SQL

Dynamic SQL is a technique that provides the ability to execute SQL commands that are not known until the commands are about to be executed. Up to this point, the SQL commands that have been illustrated in SPL programs have been static SQL - the full command (with the exception of variables) must be known and coded into the program before the program, itself, can begin to execute. Thus using dynamic SQL, the executed SQL can change during program runtime.

In addition, dynamic SQL is the only method by which data definition commands, such as CREATE TABLE, can be executed from within an SPL program.

Note, however, that the runtime performance of dynamic SQL will be slower than static SQL.

The EXECUTE IMMEDIATE command is used to run SQL commands dynamically.

```
EXECUTE IMMEDIATE 'sql_expression';
[ INTO { variable [, ...] | record } ]
[ USING expression [, ...] ]
```

**sql_expression** is a string expression containing the SQL command to be dynamically executed. **variable** receives the output of the result set, typically from a SELECT command, created as a result of executing the SQL command in **sql_expression**. The number, order, and type of variables must match the number, order, and be type-compatible with the fields of the result set. Alternatively, a record can be specified as long as the record’s fields match the number, order, and are type-compatible with the result set. When using the INTO clause, exactly one row must be returned in the result set, otherwise an exception occurs. When using the USING clause the value of **expression** is passed to a placeholder. Placeholders appear embedded within the SQL command in **sql_expression** where variables may be used. Placeholders are denoted by an identifier with a colon (:) prefix - :**name**. The number, order, and resultant data types of the evaluated expressions must match the number, order and be type-compatible with the placeholders in **sql_expression**. Note that placeholders are not declared anywhere in the SPL program – they only appear in **sql_expression**.

The following example shows basic dynamic SQL commands as string literals.

```
DECLARE
  v_sql          VARCHAR2(50);
BEGIN
  EXECUTE IMMEDIATE 'CREATE TABLE job (jobno NUMBER(3),' ||
                  ' jname VARCHAR2(9))';
  v_sql := 'INSERT INTO job VALUES (100, ''ANALYST'')';
  EXECUTE IMMEDIATE v_sql;
  v_sql := 'INSERT INTO job VALUES (200, ''CLERK'')';
  EXECUTE IMMEDIATE v_sql;
```

Copyright © 2007 - 2015 EnterpriseDB Corporation. All rights reserved.
The following example illustrates the **USING** clause to pass values to placeholders in the SQL string.

```sql
DECLARE
    v_sql          VARCHAR2(50) := 'INSERT INTO job VALUES ' ||
                    '(:p_jobno, :p_jname)';
    v_jobno        job.jobno%TYPE;
    v_jname        job.jname%TYPE;
BEGIN
    v_jobno := 300;
    v_jname := 'MANAGER';
    EXECUTE IMMEDIATE v_sql USING v_jobno, v_jname;
    v_jobno := 400;
    v_jname := 'SALESMA';
    EXECUTE IMMEDIATE v_sql USING v_jobno, v_jname;
    v_jobno := 500;
    v_jname := 'PRESIDENT';
    EXECUTE IMMEDIATE v_sql USING v_jobno, v_jname;
END;
```

The following example shows both the **INTO** and **USING** clauses. Note the last execution of the **SELECT** command returns the results into a record instead of individual variables.

```sql
DECLARE
    v_sql          VARCHAR2(60);
    v_jobno        job.jobno%TYPE;
    v_jname        job.jname%TYPE;
    r_job          job%ROWTYPE;
BEGIN
    DBMS_OUTPUT.PUT_LINE('JOBNO    JNAME');
    DBMS_OUTPUT.PUT_LINE('-----    ------');
    v_sql := 'SELECT jobno, jname FROM job WHERE jobno = :p_jobno';
    EXECUTE IMMEDIATE v_sql INTO v_jobno, v_jname USING 100;
    DBMS_OUTPUT.PUT_LINE(v_jobno || '    ' || v_jname);
    EXECUTE IMMEDIATE v_sql INTO v_jobno, v_jname USING 200;
    DBMS_OUTPUT.PUT_LINE(v_jobno || '    ' || v_jname);
    EXECUTE IMMEDIATE v_sql INTO v_jobno, v_jname USING 300;
    DBMS_OUTPUT.PUT_LINE(v_jobno || '    ' || v_jname);
    EXECUTE IMMEDIATE v_sql INTO v_jobno, v_jname USING 400;
    DBMS_OUTPUT.PUT_LINE(v_jobno || '    ' || v_jname);
    EXECUTE IMMEDIATE v_sql INTO r_job USING 500;
    DBMS_OUTPUT.PUT_LINE(r_job.jobno || '    ' || r_job.jname);
END;
```

The following is the output from the previous anonymous block:

<table>
<thead>
<tr>
<th>JOBNO</th>
<th>JNAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>ANALYST</td>
</tr>
<tr>
<td>200</td>
<td>CLERK</td>
</tr>
<tr>
<td>300</td>
<td>MANAGER</td>
</tr>
<tr>
<td>400</td>
<td>SALESMAN</td>
</tr>
<tr>
<td>500</td>
<td>PRESIDENT</td>
</tr>
</tbody>
</table>
You can use the `BULK COLLECT` clause to assemble the result set from an `EXECUTE IMMEDIATE` statement into a named collection. See Section 4.12.4, `EXECUTE IMMEDIATE BULK COLLECT` for information about using the `BULK COLLECT` clause.
4.8 Static Cursors

Rather than executing a whole query at once, it is possible to set up a cursor that encapsulates the query, and then read the query result set one row at a time. This allows the creation of SPL program logic that retrieves a row from the result set, does some processing on the data in that row, and then retrieves the next row and repeats the process.

Cursors are most often used in the context of a FOR or WHILE loop. A conditional test should be included in the SPL logic that detects when the end of the result set has been reached so the program can exit the loop.

4.8.1 Declaring a Cursor

In order to use a cursor, it must first be declared in the declaration section of the SPL program. A cursor declaration appears as follows:

```
CURSOR name IS query;
```

`name` is an identifier that will be used to reference the cursor and its result set later in the program. `query` is a SQL SELECT command that determines the result set retrievable by the cursor.

Note: An extension of this syntax allows the use of parameters. This is discussed in more detail in Section 4.8.8.

The following are some examples of cursor declarations:

```
CREATE OR REPLACE PROCEDURE cursor_example
IS
  CURSOR emp_cur_1 IS SELECT * FROM emp;
  CURSOR emp_cur_2 IS SELECT empno, ename FROM emp;
  CURSOR emp_cur_3 IS SELECT empno, ename FROM emp WHERE deptno = 10
                  ORDER BY empno;
BEGIN
  ...
END;
```

4.8.2 Opening a Cursor

Before a cursor can be used to retrieve rows, it must first be opened. This is accomplished with the OPEN statement.

```
OPEN name;
```

`name` is the identifier of a cursor that has been previously declared in the declaration section of the SPL program. The OPEN statement must not be executed on a cursor that has already been, and still is open.
The following shows an OPEN statement with its corresponding cursor declaration.

```sql
CREATE OR REPLACE PROCEDURE cursor_example
IS
    CURSOR emp_cur_3 IS SELECT empno, ename FROM emp WHERE deptno = 10
        ORDER BY empno;
BEGIN
    OPEN emp_cur_3;
    ...
END;
```

### 4.8.3 Fetching Rows From a Cursor

Once a cursor has been opened, rows can be retrieved from the cursor’s result set by using the FETCH statement.

```sql
FETCH name INTO { record | variable [, variable_2 ]... };
```

- **name** is the identifier of a previously opened cursor. **record** is the identifier of a previously defined record (for example, using `table%ROWTYPE`). **variable, variable_2...** are SPL variables that will receive the field data from the fetched row.
- The fields in **record or variable, variable_2...** must match in number and order, the fields returned in the **SELECT** list of the query given in the cursor declaration. The data types of the fields in the **SELECT** list must match, or be implicitly convertible to the data types of the fields in **record or the data types of variable, variable_2...**

**Note:** There is a variation of FETCH INTO using the BULK COLLECT clause that can return multiple rows at a time into a collection. See Section 4.12.4.2 for more information on using the BULK COLLECT clause with the FETCH INTO statement.

The following shows the FETCH statement.

```sql
CREATE OR REPLACE PROCEDURE cursor_example
IS
    v_empno         NUMBER(4);
    v_ename         VARCHAR2(10);
    CURSOR emp_cur_3 IS SELECT empno, ename FROM emp WHERE deptno = 10
        ORDER BY empno;
BEGIN
    OPEN emp_cur_3;
    FETCH emp_cur_3 INTO v_empno, v_ename;
    ...
END;
```

Instead of explicitly declaring the data type of a target variable, **%TYPE** can be used instead. In this way, if the data type of the database column is changed, the target variable declaration in the SPL program does not have to be changed. **%TYPE** will automatically pick up the new data type of the specified column.

```sql
CREATE OR REPLACE PROCEDURE cursor_example
IS
    v_empno         emp.empno%TYPE;
```
If all the columns in a table are retrieved in the order defined in the table, \%ROWTYPE can be used to define a record into which the FETCH statement will place the retrieved data. Each field within the record can then be accessed using dot notation.

```sql
CREATE OR REPLACE PROCEDURE cursor_example IS
  v_emp_rec emp%ROWTYPE;
  CURSOR emp_cur_1 IS SELECT * FROM emp;
BEGIN
  OPEN emp_cur_1;
  FETCH emp_cur_1 INTO v_emp_rec;
  DBMS_OUTPUT.PUT_LINE('Employee Number: ' || v_emp_rec.empno);
  DBMS_OUTPUT.PUT_LINE('Employee Name  : ' || v_emp_rec.ename);
  ... 
END;
```

### 4.8.4 Closing a Cursor

Once all the desired rows have been retrieved from the cursor result set, the cursor must be closed. Once closed, the result set is no longer accessible. The CLOSE statement appears as follows:

```
CLOSE name;
```

`name` is the identifier of a cursor that is currently open. Once a cursor is closed, it must not be closed again. However, once the cursor is closed, the OPEN statement can be issued again on the closed cursor and the query result set will be rebuilt after which the FETCH statement can then be used to retrieve the rows of the new result set.

The following example illustrates the use of the CLOSE statement:

```sql
CREATE OR REPLACE PROCEDURE cursor_example IS
  v_emp_rec emp%ROWTYPE;
  CURSOR emp_cur_1 IS SELECT * FROM emp;
BEGIN
  OPEN emp_cur_1;
  FETCH emp_cur_1 INTO v_emp_rec;
  DBMS_OUTPUT.PUT_LINE('Employee Number: ' || v_emp_rec.empno);
  DBMS_OUTPUT.PUT_LINE('Employee Name  : ' || v_emp_rec.ename);
  CLOSE emp_cur_1;
END;
```
This procedure produces the following output when invoked. Employee number 7369, SMITH is the first row of the result set.

```sql
EXEC cursor_example;

Employee Number: 7369
Employee Name  : SMITH
```

### 4.8.5 Using %ROWTYPE With Cursors

Using the `%ROWTYPE` attribute, a record can be defined that contains fields corresponding to all columns fetched from a cursor or cursor variable. Each field takes on the data type of its corresponding column. The `%ROWTYPE` attribute is prefixed by a cursor name or cursor variable name.

```sql
record cursor%ROWTYPE;
```

*record* is an identifier assigned to the record. *cursor* is an explicitly declared cursor within the current scope.

The following example shows how you can use a cursor with `%ROWTYPE` to get information about which employee works in which department.

```sql
CREATE OR REPLACE PROCEDURE emp_info
IS
    CURSOR empcur IS SELECT ename, deptno FROM emp;
    myvar           empcur%ROWTYPE;
BEGIN
    OPEN empcur;
    LOOP
        FETCH empcur INTO myvar;
        EXIT WHEN empcur%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE( myvar.ename || ' works in department ' || myvar.deptno );
    END LOOP;
    CLOSE empcur;
END;
```

The following is the output from this procedure.

```sql
EXEC emp_info;

SMITH works in department 20
ALLEN works in department 30
WARD works in department 30
JONES works in department 20
MARTIN works in department 30
BLAKE works in department 30
CLARK works in department 10
SCOTT works in department 20
KING works in department 10
TURNER works in department 30
ADAMS works in department 20
JAMES works in department 30
FORD works in department 20
```
4.8.6 Cursor Attributes

Each cursor has a set of attributes associated with it that allows the program to test the state of the cursor. These attributes are %ISOPEN, %FOUND, %NOTFOUND, and %ROWCOUNT. These attributes are described in the following sections.

4.8.6.1 %ISOPEN

The %ISOPEN attribute is used to test whether or not a cursor is open.

```
cursor_name%ISOPEN
```

`cursor_name` is the name of the cursor for which a BOOLEAN data type of TRUE will be returned if the cursor is open, FALSE otherwise.

The following is an example of using %ISOPEN.

```
CREATE OR REPLACE PROCEDURE cursor_example
IS
  ...
  CURSOR emp_cur_1 IS SELECT * FROM emp;
  ...
BEGIN
  ...
  IF emp_cur_1%ISOPEN THEN
    NULL;
  ELSE
    OPEN emp_cur_1;
  END IF;
  FETCH emp_cur_1 INTO ...
  ...
END;
```

4.8.6.2 %FOUND

The %FOUND attribute is used to test whether or not a row is retrieved from the result set of the specified cursor after a FETCH on the cursor.

```
cursor_name%FOUND
```

`cursor_name` is the name of the cursor for which a BOOLEAN data type of TRUE will be returned if a row is retrieved from the result set of the cursor after a FETCH.

After the last row of the result set has been FETCHed the next FETCH results in %FOUND returning FALSE. FALSE is also returned after the first FETCH if there are no rows in the result set to begin with.
Referencing %FOUND on a cursor before it is opened or after it is closed results in an INVALID_CURSOR exception being thrown.

%FOUND returns null if it is referenced when the cursor is open, but before the first FETCH.

The following example uses %FOUND.

```sql
CREATE OR REPLACE PROCEDURE cursor_example IS
  v_emp_rec emp%ROWTYPE;
  CURSOR emp_cur_1 IS SELECT * FROM emp;
BEGIN
  OPEN emp_cur_1;
  DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
  DBMS_OUTPUT.PUT_LINE('-----    ------');
  FETCH emp_cur_1 INTO v_emp_rec;
  WHILE emp_cur_1%FOUND LOOP
    DBMS_OUTPUT.PUT_LINE(v_emp_rec.empno || '     ' || v_emp_rec.ename);
    FETCH emp_cur_1 INTO v_emp_rec;
  END LOOP;
  CLOSE emp_cur_1;
END;
```

When the previous procedure is invoked, the output appears as follows:

```
EXEC cursor_example;
EMPNO    ENAME
-----    ------
7369      SMITH
7499      ALLEN
7521      WARD
7566      JONES
7654      MARTIN
7698      BLAKE
7782      CLARK
7788      SCOTT
7839      KING
7844      TURNER
7876      ADAMS
7900      JAMES
7902      FORD
7934      MILLER
```

4.8.6.3 %NOTFOUND

The %NOTFOUND attribute is the logical opposite of %FOUND.

```sql
cursor_name%NOTFOUND
```

cursor_name is the name of the cursor for which a BOOLEAN data type of FALSE will be returned if a row is retrieved from the result set of the cursor after a FETCH.
After the last row of the result set has been FETCHed the next FETCH results in %NOTFOUND returning TRUE. TRUE is also returned after the first FETCH if there are no rows in the result set to begin with.

Referencing %NOTFOUND on a cursor before it is opened or after it is closed, results in an INVALID_CURSOR exception being thrown.

%NOTFOUND returns null if it is referenced when the cursor is open, but before the first FETCH.

The following example uses %NOTFOUND.

```
CREATE OR REPLACE PROCEDURE cursor_example
IS
    v_emp_rec       emp%ROWTYPE;
    CURSOR emp_cur_1 IS SELECT * FROM emp;
BEGIN
    OPEN emp_cur_1;
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
    DBMS_OUTPUT.PUT_LINE('-----     ------');
    LOOP
        FETCH emp_cur_1 INTO v_emp_rec;
        EXIT WHEN emp_cur_1%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(v_emp_rec.empno || '     ' || v_emp_rec.ename);
    END LOOP;
    CLOSE emp_cur_1;
END;
```

Similar to the prior example, this procedure produces the same output when invoked.

```
EXEC cursor_example;

EMPNO    ENAME
-----     ------
7369      SMITH
7499      ALLEN
7521      WARD
7566      JONES
7654      MARTIN
7698      BLAKE
7782      CLARK
7788      SCOTT
7839      KING
7844      TURNER
7876      ADAMS
7900      JAMES
7902      FORD
7934      MILLER
```

### 4.8.6.4 %ROWCOUNT

The %ROWCOUNT attribute returns an integer showing the number of rows FETCHed so far from the specified cursor.
cursor_name%ROWCOUNT

cursor_name is the name of the cursor for which %ROWCOUNT returns the number of rows retrieved thus far. After the last row has been retrieved, %ROWCOUNT remains set to the total number of rows returned until the cursor is closed at which point %ROWCOUNT will throw an INVALID_CURSOR exception if referenced.

Referencing %ROWCOUNT on a cursor before it is opened or after it is closed, results in an INVALID_CURSOR exception being thrown.

%ROWCOUNT returns 0 if it is referenced when the cursor is open, but before the first FETCH. %ROWCOUNT also returns 0 after the first FETCH when there are no rows in the result set to begin with.

The following example uses %ROWCOUNT.

```sql
CREATE OR REPLACE PROCEDURE cursor_example
IS
    v_emp_rec       emp%ROWTYPE;
    CURSOR emp_cur_1 IS SELECT * FROM emp;
BEGIN
    OPEN emp_cur_1;
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
    DBMS_OUTPUT.PUT_LINE('-----    ------');
    LOOP
        FETCH emp_cur_1 INTO v_emp_rec;
        EXIT WHEN emp_cur_1%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(v_emp_rec.empno || '     ' || v_emp_rec.ename);
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('**********************');
    DBMS_OUTPUT.PUT_LINE(emp_cur_1%ROWCOUNT || ' rows were retrieved');
    CLOSE emp_cur_1;
END;
```

This procedure prints the total number of rows retrieved at the end of the employee list as follows:

```
EXEC cursor_example;

EMPNO    ENAME
-----    ------
7369     SMITH
7499     ALLEN
7521     WARD
7566     JONES
7654     MARTIN
7698     BLAKE
7782     CLARK
7788     SCOTT
7839     KING
7844     TURNER
7876     ADAMS
7900     JAMES
7902     FORD
7934     MILLER
```
4.8.6.5 Summary of Cursor States and Attributes

The following table summarizes the possible cursor states and the values returned by the cursor attributes.

<table>
<thead>
<tr>
<th>Before OPEN</th>
<th>After OPEN &amp; Before 1st FETCH</th>
<th>After 1st Successful FETCH</th>
<th>After n-th Successful FETCH (last row)</th>
<th>After n+1st FETCH (after last row)</th>
<th>After CLOSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>False</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>INVALID_CURSOR Exception</td>
<td>Null</td>
<td>False</td>
<td>False</td>
<td>True</td>
<td>INVALID_CURSOR Exception</td>
</tr>
<tr>
<td>INVALID_CURSOR Exception</td>
<td>INVALID_CURSOR Exception</td>
<td>INVALID_CURSOR Exception</td>
<td>INVALID_CURSOR Exception</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

4.8.7 Cursor FOR Loop

In the cursor examples presented so far, the programming logic required to process the result set of a cursor included a statement to open the cursor, a loop construct to retrieve each row of the result set, a test for the end of the result set, and finally a statement to close the cursor. The cursor FOR loop is a loop construct that eliminates the need to individually code the statements just listed.

The cursor FOR loop opens a previously declared cursor, fetches all rows in the cursor result set, and then closes the cursor.

The syntax for creating a cursor FOR loop is as follows.

```sql
FOR record IN cursor
LOOP
    statements
END LOOP;
```

*record* is an identifier assigned to an implicitly declared record with definition, *cursor%ROWTYPE*, *cursor* is the name of a previously declared cursor. *statements* are one or more SPL statements. There must be at least one statement.

The following example shows the example from Section 4.8.6.3, modified to use a cursor FOR loop.
CREATE OR REPLACE PROCEDURE cursor_example
IS
  CURSOR emp_cur_1 IS SELECT * FROM emp;
BEGIN
  DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
  DBMS_OUTPUT.PUT_LINE('-----    ------');
  FOR v_emp_rec IN emp_cur_1 LOOP
    DBMS_OUTPUT.PUT_LINE(v_emp_rec.empno || '     ' || v_emp_rec.ename);
  END LOOP;
END;

The same results are achieved as shown in the output below.

EXEC cursor_example;

EMPNO    ENAME
-----    ------
7369     SMITH
7499     ALLEN
7521     WARD
7566     JONES
7654     MARTIN
7698     BLAKE
7782     CLARK
7788     SCOTT
7839     KING
7844     TURNER
7876     ADAMS
7900     JAMES
7902     FORD
7934     MILLER

4.8.8 Parameterized Cursors

A user can also declare a static cursor that accepts parameters, and can pass values for those parameters when opening that cursor. In the following example we have created a parameterized cursor which will display the name and salary of all employees from the emp table that have a salary less than a specified value which is passed as a parameter.

DECLARE
  my_record emp%ROWTYPE;
  CURSOR c1 (max_wage NUMBER) IS
    SELECT * FROM emp WHERE sal < max_wage;
BEGIN
  OPEN c1(2000);
  LOOP
    FETCH c1 INTO my_record;
    EXIT WHEN c1%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE('Name = ' || my_record.ename || ', salary = ' || my_record.sal);
  END LOOP;
  CLOSE c1;
END;
So for example if we pass the value 2000 as \texttt{max\_wage}, then we will only be shown the name and salary of all employees that have a salary less than 2000. The result of the above query is the following:

\begin{verbatim}
Name = SMITH, salary = 800.00
Name = ALLEN, salary = 1600.00
Name = WARD, salary = 1250.00
Name = MARTIN, salary = 1250.00
Name = TURNER, salary = 1500.00
Name = ADAMS, salary = 1100.00
Name = JAMES, salary = 950.00
Name = MILLER, salary = 1300.00
\end{verbatim}
4.9 REF CURSORs and Cursor Variables

This section discusses another type of cursor that provides far greater flexibility than the previously discussed static cursors.

4.9.1 REF CURSOR Overview

A cursor variable is a cursor that actually contains a pointer to a query result set. The result set is determined by the execution of the `OPEN FOR` statement using the cursor variable.

A cursor variable is not tied to a single particular query like a static cursor. The same cursor variable may be opened a number of times with `OPEN FOR` statements containing different queries. Each time, a new result set is created from that query and made available via the cursor variable.

REF CURSOR types may be passed as parameters to or from stored procedures and functions. The return type of a function may also be a REF CURSOR type. This provides the capability to modularize the operations on a cursor into separate programs by passing a cursor variable between programs.

4.9.2 Declaring a Cursor Variable

SPL supports the declaration of a cursor variable using both the `SYS_REFCURSOR` built-in data type as well as creating a type of REF CURSOR and then declaring a variable of that type. `SYS_REFCURSOR` is a REF CURSOR type that allows any result set to be associated with it. This is known as a weakly-typed REF CURSOR.

Only the declaration of `SYS_REFCURSOR` and user-defined REF CURSOR variables are different. The remaining usage like opening the cursor, selecting into the cursor and closing the cursor is the same across both the cursor types. For the rest of this chapter our examples will primarily be making use of the `SYS_REFCURSOR` cursors. All you need to change in the examples to make them work for user defined REF CURSORS is the declaration section.

Note: Strongly-typed REF CURSORS require the result set to conform to a declared number and order of fields with compatible data types and can also optionally return a result set.

4.9.2.1 Declaring a SYS_REFCURSOR Cursor Variable

The following is the syntax for declaring a SYS_REFCURSOR cursor variable:

```sql
name SYS_REFCURSOR;
```
name is an identifier assigned to the cursor variable.

The following is an example of a SYS_REFCURSOR variable declaration.

```
DECLARE
    emp_refcur      SYS_REFCURSOR;
...
```

### 4.9.2.2 Declaring a User Defined REF CURSOR Type Variable

You must perform two distinct declaration steps in order to use a user defined REF CURSOR variable:

- Create a referenced cursor TYPE
- Declare the actual cursor variable based on that TYPE

The syntax for creating a user defined REF CURSOR type is as follows:

```
TYPE cursor_type_name IS REF CURSOR [RETURN return_type];
```

The following is an example of a cursor variable declaration.

```
DECLARE
    TYPE emp_cur_type IS REF CURSOR RETURN emp%ROWTYPE;
    my_rec emp_cur_type;
    ...
```

### 4.9.3 Opening a Cursor Variable

Once a cursor variable is declared, it must be opened with an associated SELECT command. The OPEN FOR statement specifies the SELECT command to be used to create the result set.

```
OPEN name FOR query;
```

name is the identifier of a previously declared cursor variable. query is a SELECT command that determines the result set when the statement is executed. The value of the cursor variable after the OPEN FOR statement is executed identifies the result set.

In the following example, the result set is a list of employee numbers and names from a selected department. Note that a variable or parameter can be used in the SELECT command anywhere an expression can normally appear. In this case a parameter is used in the equality test for department number.

```
CREATE OR REPLACE PROCEDURE emp_by_dept (p_deptno emp.deptno%TYPE
```
4.9.4 Fetching Rows From a Cursor Variable

After a cursor variable is opened, rows may be retrieved from the result set using the `FETCH` statement. See Section 4.8.3 for details on using the `FETCH` statement to retrieve rows from a result set.

In the example below, a `FETCH` statement has been added to the previous example so now the result set is returned into two variables and then displayed. Note that the cursor attributes used to determine cursor state of static cursors can also be used with cursor variables. See Section 4.8.6 for details on cursor attributes.

```sql
CREATE OR REPLACE PROCEDURE emp_by_dept (p_deptno emp.deptno%TYPE) IS
  emp_refcur SYS_REFCURSOR;
  v_empno emp.empno%TYPE;
  v_ename emp.ename%TYPE;
BEGIN
  OPEN emp_refcur FOR SELECT empno, ename FROM emp WHERE deptno = p_deptno;
  DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
  DBMS_OUTPUT.PUT_LINE('-----    -------');
  LOOP
    FETCH emp_refcur INTO v_empno, v_ename;
    EXIT WHEN emp_refcur%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE(v_empno || '     ' || v_ename);
  END LOOP;
END LOOP;
```

4.9.5 Closing a Cursor Variable

Use the `CLOSE` statement described in Section 4.8.4 to release the result set.

**Note:** Unlike static cursors, a cursor variable does not have to be closed before it can be re-opened again. The result set from the previous open will be lost.

The example is completed with the addition of the `CLOSE` statement.

```sql
CREATE OR REPLACE PROCEDURE emp_by_dept (p_deptno emp.deptno%TYPE) IS
  emp_refcur SYS_REFCURSOR;
  v_empno emp.empno%TYPE;
  v_ename emp.ename%TYPE;
BEGIN
  OPEN emp_refcur FOR SELECT empno, ename FROM emp WHERE deptno = p_deptno;
  DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
  DBMS_OUTPUT.PUT_LINE('-----    -------');
  LOOP
    FETCH emp_refcur INTO v_empno, v_ename;
    EXIT WHEN emp_refcur%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE(v_empno || '     ' || v_ename);
  END LOOP;
  CLOSE emp_refcur;
  DBMS_OUTPUT.PUT_LINE('--------    --------');
END LOOP;
```
BEGIN
    OPEN emp_refcur FOR SELECT empno, ename FROM emp WHERE deptno = p_deptno;
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
    DBMS_OUTPUT.PUT_LINE('-----    ------');
    LOOP
        FETCH emp_refcur INTO v_empno, v_ename;
        EXIT WHEN emp_refcur%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(v_empno || '     ' || v_ename);
    END LOOP;
    CLOSE emp_refcur;
END;

The following is the output when this procedure is executed.

EXEC emp_by_dept(20)
EMPNO    ENAME
-----    ------
7369     SMITH
7566     JONES
7788     SCOTT
7876     ADAMS
7902     FORD

4.9.6 Usage Restrictions

The following are restrictions on cursor variable usage.

- Comparison operators cannot be used to test cursor variables for equality, inequality, null, or not null
- Null cannot be assigned to a cursor variable
- The value of a cursor variable cannot be stored in a database column
- Static cursors and cursor variables are not interchangeable. For example, a static cursor cannot be used in an OPEN FOR statement.

In addition the following table shows the permitted parameter modes for a cursor variable used as a procedure or function parameter depending upon the operations on the cursor variable within the procedure or function.

<table>
<thead>
<tr>
<th>Operation</th>
<th>IN</th>
<th>IN OUT</th>
<th>OUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>OPEN</td>
<td>No</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FETCH</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>CLOSE</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

So for example, if a procedure performs all three operations, OPEN FOR, FETCH, and CLOSE on a cursor variable declared as the procedure’s formal parameter, then that parameter must be declared with IN OUT mode.
4.9.7 Examples

The following examples demonstrate cursor variable usage.

4.9.7.1 Returning a REF CURSOR From a Function

In the following example the cursor variable is opened with a query that selects employees with a given job. Note that the cursor variable is specified in this function’s RETURN statement so the result set is made available to the caller of the function.

```
CREATE OR REPLACE FUNCTION emp_by_job (p_job VARCHAR2) RETURN SYS_REFCURSOR IS
    emp_refcur SYS_REFCURSOR;
BEGIN
    OPEN emp_refcur FOR SELECT empno, ename FROM emp WHERE job = p_job;
    RETURN emp_refcur;
END;
```

This function is invoked in the following anonymous block by assigning the function’s return value to a cursor variable declared in the anonymous block’s declaration section. The result set is fetched using this cursor variable and then it is closed.

```
DECLARE
    v_empno emp.empno%TYPE;
    v_ename emp.ename%TYPE;
    v_job emp.job%TYPE := 'SALESMAN';
    v_emp_refcur SYS_REFCURSOR;
BEGIN
    DBMS_OUTPUT.PUT_LINE('EMPLOYEES WITH JOB ' || v_job);
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
    DBMS_OUTPUT.PUT_LINE('-----    ------');
    v_emp_refcur := emp_by_job(v_job);
    LOOP
        FETCH v_emp_refcur INTO v_empno, v_ename;
        EXIT WHEN v_emp_refcur%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(v_empno || '     ' || v_ename);
    END LOOP;
    CLOSE v_emp_refcur;
END;
```

The following is the output when the anonymous block is executed.

```
EMPLOYEES WITH JOB SALESMAN
EMPNO    ENAME
-----    ------
7499     ALLEN
7521     WARD
7654     MARTIN
7844     TURNER
```
4.9.7.2 Modularizing Cursor Operations

The following example illustrates how the various operations on cursor variables can be modularized into separate programs.

The following procedure opens the given cursor variable with a `SELECT` command that retrieves all rows.

```sql
CREATE OR REPLACE PROCEDURE open_all_emp (p_emp_refcur IN OUT SYS_REFCURSOR) IS BEGIN OPEN p_emp_refcur FOR SELECT empno, ename FROM emp; END;
```

This variation opens the given cursor variable with a `SELECT` command that retrieves all rows, but of a given department.

```sql
CREATE OR REPLACE PROCEDURE open_emp_by_dept (p_emp_refcur IN OUT SYS_REFCURSOR, p_deptno emp.deptno%TYPE) IS BEGIN OPEN p_emp_refcur FOR SELECT empno, ename FROM emp WHERE deptno = p_deptno; END;
```

This third variation opens the given cursor variable with a `SELECT` command that retrieves all rows, but from a different table. Also note that the function’s return value is the opened cursor variable.

```sql
CREATE OR REPLACE FUNCTION open_dept (p_dept_refcur IN OUT SYS_REFCURSOR) RETURN SYS_REFCURSOR IS v_dept_refcur SYS_REFCURSOR; BEGIN v_dept_refcur := p_dept_refcur; OPEN v_dept_refcur FOR SELECT deptno, dname FROM dept; RETURN v_dept_refcur; END;
```

This procedure fetches and displays a cursor variable result set consisting of employee number and name.

```sql
CREATE OR REPLACE PROCEDURE fetch_emp (p_emp_refcur IN OUT SYS_REFCURSOR) IS v_empno emp.empno%TYPE; v_ename emp.ename%TYPE; BEGIN DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
This procedure fetches and displays a cursor variable result set consisting of department number and name.

```sql
CREATE OR REPLACE PROCEDURE fetch_dept (  
    p_dept_refcur   IN SYS_REFCURSOR  
)  
IS  
  v_deptno        dept.deptno%TYPE;  
  v_dname         dept.dname%TYPE;  
BEGIN  
  DBMS_OUTPUT.PUT_LINE('DEPT   DNAME');  
  DBMS_OUTPUT.PUT_LINE('-----   --------');  
  LOOP  
    FETCH p_dept_refcur INTO v_deptno, v_dname;  
    EXIT WHEN p_dept_refcur%NOTFOUND;  
    DBMS_OUTPUT.PUT_LINE(v_deptno || '     ' || v_dname);  
  END LOOP;  
END;
```

This procedure closes the given cursor variable.

```sql
CREATE OR REPLACE PROCEDURE close_refcur (  
    p_refcur        IN OUT SYS_REFCURSOR  
)  
IS  
BEGIN  
  CLOSE p_refcur;  
END;
```

The following anonymous block executes all the previously described programs.

```sql
DECLARE  
  gen_refcur SYS_REFCURSOR;  
BEGIN  
  DBMS_OUTPUT.PUT_LINE('ALL EMPLOYEES');  
  open_all_emp(gen_refcur);  
  fetch_emp(gen_refcur);  
  DBMS_OUTPUT.PUT_LINE('****************');  
  DBMS_OUTPUT.PUT_LINE('EMPLOYEES IN DEPT #10');  
  open_emp_by_dept(gen_refcur, 10);  
  fetch_emp(gen_refcur);  
  DBMS_OUTPUT.PUT_LINE('*****************');  
  DBMS_OUTPUT.PUT_LINE('DEPARTMENTS');  
  fetch_dept(open_dept(gen_refcur));  
  DBMS_OUTPUT.PUT_LINE('**************');  
  close_refcur(gen_refcur);  
END;
```
The following is the output from the anonymous block.

```
<table>
<thead>
<tr>
<th>EMPNO</th>
<th>ENAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>7369</td>
<td>SMITH</td>
</tr>
<tr>
<td>7499</td>
<td>ALLEN</td>
</tr>
<tr>
<td>7521</td>
<td>WARD</td>
</tr>
<tr>
<td>7566</td>
<td>JONES</td>
</tr>
<tr>
<td>7654</td>
<td>MARTIN</td>
</tr>
<tr>
<td>7698</td>
<td>BLAKE</td>
</tr>
<tr>
<td>7782</td>
<td>CLARK</td>
</tr>
<tr>
<td>7788</td>
<td>SCOTT</td>
</tr>
<tr>
<td>7839</td>
<td>KING</td>
</tr>
<tr>
<td>7844</td>
<td>TURNER</td>
</tr>
<tr>
<td>7876</td>
<td>ADAMS</td>
</tr>
<tr>
<td>7900</td>
<td>JAMES</td>
</tr>
<tr>
<td>7902</td>
<td>FORD</td>
</tr>
<tr>
<td>7934</td>
<td>MILLER</td>
</tr>
</tbody>
</table>
```

```
EMPLOYEES IN DEPT #10
<table>
<thead>
<tr>
<th>EMPNO</th>
<th>ENAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>7782</td>
<td>CLARK</td>
</tr>
<tr>
<td>7839</td>
<td>KING</td>
</tr>
<tr>
<td>7934</td>
<td>MILLER</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>DEPT</th>
<th>DNAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>ACCOUNTING</td>
</tr>
<tr>
<td>20</td>
<td>RESEARCH</td>
</tr>
<tr>
<td>30</td>
<td>SALES</td>
</tr>
<tr>
<td>40</td>
<td>OPERATIONS</td>
</tr>
</tbody>
</table>
```

### 4.9.8 Dynamic Queries With REF CURSORs

Postgres Plus Advanced Server also supports dynamic queries via the `OPEN FOR USING` statement. A string literal or string variable is supplied in the `OPEN FOR USING` statement to the `SELECT` command.

```sql
OPEN name FOR dynamic_string
    [ USING bind_arg [, bind_arg_2 ] ... ];
```

`name` is the identifier of a previously declared cursor variable. `dynamic_string` is a string literal or string variable containing a `SELECT` command (without the terminating semi-colon). `bind_arg, bind_arg_2...` are bind arguments that are used to pass variables to corresponding placeholders in the `SELECT` command when the cursor variable is opened. The placeholders are identifiers prefixed by a colon character.

The following is an example of a dynamic query using a string literal.

```
CREATE OR REPLACE PROCEDURE dept_query
```
IS
emp_refcur SYS_REFCURSOR;
v_empno emp.empno%TYPE;
v_ename emp.ename%TYPE;
BEGIN
OPEN emp_refcur FOR 'SELECT empno, ename FROM emp WHERE deptno = 30' ||
' AND sal >= 1500';
DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
DBMS_OUTPUT.PUT_LINE('-----     -------');
LOOP
FETCH emp_refcur INTO v_empno, v_ename;
EXIT WHEN emp_refcur%NOTFOUND;
DBMS_OUTPUT.PUT_LINE(v_empno || ' ' || v_ename);
END LOOP;
CLOSE emp_refcur;
END;

The following is the output when the procedure is executed.

EXEC dept_query;

EMPNO    ENAME
-----     ------
7499     ALLEN
7698     BLAKE
7844     TURNER

In the next example, the previous query is modified to use bind arguments to pass the
query parameters.

CREATE OR REPLACE PROCEDURE dept_query (p_deptno emp.deptno%TYPE,
p_sal emp.sal%TYPE)
IS
emp_refcur SYS_REFCURSOR;
v_empno emp.empno%TYPE;
v_ename emp.ename%TYPE;
BEGIN
OPEN emp_refcur FOR 'SELECT empno, ename FROM emp WHERE deptno = :dept' ||
' AND sal >= :sal' USING p_deptno, p_sal;
DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
DBMS_OUTPUT.PUT_LINE('-----     -------');
LOOP
FETCH emp_refcur INTO v_empno, v_ename;
EXIT WHEN emp_refcur%NOTFOUND;
DBMS_OUTPUT.PUT_LINE(v_empno || ' ' || v_ename);
END LOOP;
CLOSE emp_refcur;
END;

The following is the resulting output.

EXEC dept_query(30, 1500);

EMPNO    ENAME
-----     ------
7499     ALLEN
7698     BLAKE
Finally, a string variable is used to pass the `SELECT` providing the most flexibility.

```sql
CREATE OR REPLACE PROCEDURE dept_query(
    p_deptno        emp.deptno%TYPE,
    p_sal           emp.sal%TYPE
) IS
    emp_refcur      SYS_REFCURSOR;
    v_empno         emp.empno%TYPE;
    v_ename         emp.ename%TYPE;
    p_query_string  VARCHAR2(100);
BEGIN
    p_query_string := 'SELECT empno, ename FROM emp WHERE ' ||
                         'deptno = :dept AND sal >= :sal';
    OPEN emp_refcur FOR p_query_string USING p_deptno, p_sal;
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
    DBMS_OUTPUT.PUT_LINE('-----------------------');
    LOOP
        FETCH emp_refcur INTO v_empno, v_ename;
        EXIT WHEN emp_refcur%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(v_empno || '     ' || v_ename);
    END LOOP;
    CLOSE emp_refcur;
END;
EXEC dept_query(20, 1500);
EMPNO    ENAME
-------- -------
7566     JONES
7788     SCOTT
7902     FORD
```
4.10 Collections

A collection is a set of ordered data items with the same data type. Generally, the data item is a scalar field, but may also be a user-defined type such as a record type or an object type (see Section 8 for information about object types) as long as the structure and the data types that comprise each field of the user-defined type are the same for each element in the set. Each particular data item in the set is referenced by using subscript notation within a pair of parentheses.

Note: Multilevel collections (that is, where the data item of a collection is another collection) are not supported.

The most commonly known type of collection is an array. In Postgres Plus Advanced Server, the supported collection types are associative arrays (formerly called index-by-tables in Oracle), nested tables, and varrays.

The general steps for using a collection are the following:

- A collection of the desired type must be defined. This can be done in the declaration section of an SPL program, which results in a local type that is accessible only within that program. For nested table and varray types this can also be done using the CREATE TYPE command, which creates a persistent, standalone type that can be referenced by any SPL program in the database.
- Variables of the collection type are declared. The collection associated with the declared variable is said to be uninitialized at this point if there is no value assignment made as part of the variable declaration.
- Uninitialized collections of nested tables and varrays are null. A null collection does not yet exist. Generally, a COLLECTION IS NULL exception is thrown if a collection method is invoked on a null collection.
- Uninitialized collections of associative arrays exist, but have no elements. An existing collection with no elements is called an empty collection.
- To initialize a null collection, you must either make it an empty collection or assign a non-null value to it. Generally, a null collection is initialized by using its constructor.
- To add elements to an empty associative array, you can simply assign values to its keys. For nested tables and varrays, generally its constructor is used to assign initial values to the nested table or varray. For nested tables and varrays, the EXTEND method is then used to grow the collection beyond its initial size established by the constructor.

The specific process for each collection type is described in the following sections.
4.10.1  Associative Arrays

An **associative array** is a type of collection that associates a unique key with a value. The key does not have to be numeric, but can be character data as well.

An associative array has the following characteristics:

- An **associative array type** must be defined after which **array variables** can be declared of that array type. Data manipulation occurs using the array variable.
- When an array variable is declared, the associative array is created, but it is empty - just start assigning values to key values.
- The key can be any negative integer, positive integer, or zero if `INDEX BY BINARY_INTEGER` or `PLS_INTEGER` is specified.
- The key can be character data if `INDEX BY VARCHAR2` is specified.
- There is no pre-defined limit on the number of elements in the array - it grows dynamically as elements are added.
- The array can be sparse - there may be gaps in the assignment of values to keys.
- An attempt to reference an array element that has not been assigned a value will result in an exception.

The `TYPE IS TABLE OF ... INDEX BY` statement is used to define an associative array type.

```
TYPE assoctype IS TABLE OF { datatype | rectype | objtype }  
    INDEX BY { BINARY_INTEGER | PLS_INTEGER | VARCHAR2(n) };
```

`assoctype` is an identifier assigned to the array type. `datatype` is a scalar data type such as `VARCHAR2` or `NUMBER`. `rectype` is a previously defined record type. `objtype` is a previously defined object type. `n` is the maximum length of a character key.

In order to make use of the array, a **variable** must be declared with that array type. The following is the syntax for declaring an array variable.

```
array assoctype
```

`array` is an identifier assigned to the associative array. `assoctype` is the identifier of a previously defined array type.

An element of the array is referenced using the following syntax.

```
array(n)[.field ]
```

`array` is the identifier of a previously declared array. `n` is the key value, type-compatible with the data type given in the `INDEX BY` clause. If the array type of `array` is defined
from a record type or object type, then \[.field\] must reference an individual field within the record type or attribute within the object type from which the array type is defined. Alternatively, the entire record can be referenced by omitting \[.field\].

The following example reads the first ten employee names from the emp table, stores them in an array, then displays the results from the array.

```sql
DECLARE
    TYPE emp_arr_typ IS TABLE OF VARCHAR2(10) INDEX BY BINARY_INTEGER;
    emp_arr         emp_arr_typ;
    CURSOR emp_cur IS SELECT ename FROM emp WHERE ROWNUM <= 10;
    i               INTEGER := 0;
BEGIN
    FOR r_emp IN emp_cur LOOP
        i := i + 1;
        emp_arr(i) := r_emp.ename;
    END LOOP;
    FOR j IN 1..10 LOOP
        DBMS_OUTPUT.PUT_LINE(emp_arr(j));
    END LOOP;
END;
```

The above example produces the following output:

```
SMITH
ALLEN
WARD
JONES
MARTIN
BLAKE
CLARK
SCOTT
KING
TURNER
```

The previous example is now modified to use a record type in the array definition.

```sql
DECLARE
    TYPE emp_rec_typ IS RECORD (
        empno   NUMBER(4),
        ename   VARCHAR2(10)
    );
    TYPE emp_arr_typ IS TABLE OF emp_rec_typ INDEX BY BINARY_INTEGER;
    emp_arr         emp_arr_typ;
    CURSOR emp_cur IS SELECT empno, ename FROM emp WHERE ROWNUM <= 10;
    i               INTEGER := 0;
BEGIN
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME ');
    DBMS_OUTPUT.PUT_LINE('-----    ------- ');
    FOR r_emp IN emp_cur LOOP
        i := i + 1;
        emp_arr(i).empno := r_emp.empno;
        emp_arr(i).ename := r_emp.ename;
    END LOOP;
    FOR j IN 1..10 LOOP
        DBMS_OUTPUT.PUT_LINE(emp_arr(j).empno || '     ' || emp_arr(j).ename);
    END LOOP;
END;
```
The following is the output from this anonymous block.

<table>
<thead>
<tr>
<th>EMPNO</th>
<th>ENAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>7369</td>
<td>SMITH</td>
</tr>
<tr>
<td>7499</td>
<td>ALLEN</td>
</tr>
<tr>
<td>7521</td>
<td>WARD</td>
</tr>
<tr>
<td>7566</td>
<td>JONES</td>
</tr>
<tr>
<td>7654</td>
<td>MARTIN</td>
</tr>
<tr>
<td>7698</td>
<td>BLAKE</td>
</tr>
<tr>
<td>7782</td>
<td>CLARK</td>
</tr>
<tr>
<td>7788</td>
<td>SCOTT</td>
</tr>
<tr>
<td>7839</td>
<td>KING</td>
</tr>
<tr>
<td>7844</td>
<td>TURNER</td>
</tr>
</tbody>
</table>

The **emp%ROWTYPE** attribute could be used to define **emp_arr_typ** instead of using the **emp_rec_typ** record type as shown in the following.

```sql
DECLARE
    TYPE emp_arr_typ IS TABLE OF emp%ROWTYPE INDEX BY BINARY_INTEGER;
    emp_arr emp_arr_typ;
    CURSOR emp_cur IS SELECT empno, ename FROM emp WHERE ROWNUM <= 10;
    i INTEGER := 0;
BEGIN
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
    DBMS_OUTPUT.PUT_LINE('-----    ------');
    FOR r_emp IN emp_cur LOOP
        i := i + 1;
        emp_arr(i).empno := r_emp.empno;
        emp_arr(i).ename := r_emp.ename;
    END LOOP;
    FOR j IN 1..10 LOOP
        DBMS_OUTPUT.PUT_LINE(emp_arr(j).empno || '     ' || emp_arr(j).ename);
    END LOOP;
END;
```

The results are the same as in the prior example.

Instead of assigning each field of the record individually, a record level assignment can be made from **r_emp** to **emp_arr**.

```sql
DECLARE
    TYPE emp_rec_typ IS RECORD (
        empno NUMBER(4),
        ename VARCHAR2(10)
    );
    TYPE emp_arr_typ IS TABLE OF emp_rec_typ INDEX BY BINARY_INTEGER;
    emp_arr emp_arr_typ;
    CURSOR emp_cur IS SELECT empno, ename FROM emp WHERE ROWNUM <= 10;
    i INTEGER := 0;
BEGIN
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
    DBMS_OUTPUT.PUT_LINE('-----    ------');
    FOR r_emp IN emp_cur LOOP
        i := i + 1;
        emp_arr(i).empno := r_emp.empno;
        emp_arr(i).ename := r_emp.ename;
    END LOOP;
    FOR j IN 1..10 LOOP
        DBMS_OUTPUT.PUT_LINE(emp_arr(j).empno || '     ' || emp_arr(j).ename);
    END LOOP;
END;
```
The key of an associative array can be character data as shown in the following example.

```
DECLARE
    TYPE job_arr_typ IS TABLE OF NUMBER INDEX BY VARCHAR2(9);
    job_arr          job_arr_typ;
BEGIN
    job_arr('ANALYST')   := 100;
    job_arr('CLERK')     := 200;
    job_arr('MANAGER')   := 300;
    job_arr('SALESMAN')  := 400;
    job_arr('PRESIDENT') := 500;
    DBMS_OUTPUT.PUT_LINE('ANALYST  : ' || job_arr('ANALYST'));
    DBMS_OUTPUT.PUT_LINE('CLERK    : ' || job_arr('CLERK'));
    DBMS_OUTPUT.PUT_LINE('MANAGER  : ' || job_arr('MANAGER'));
    DBMS_OUTPUT.PUT_LINE('SALESMAN : ' || job_arr('SALESMAN'));
    DBMS_OUTPUT.PUT_LINE('PRESIDENT: ' || job_arr('PRESIDENT'));
END;
```

ANALYST  : 100
CLERK    : 200
MANAGER  : 300
SALESMAN : 400
PRESIDENT: 500
4.10.2 Nested Tables

A nested table is a type of collection that associates a positive integer with a value. A nested table has the following characteristics:

- A nested table type must be defined after which nested table variables can be declared of that nested table type. Data manipulation occurs using the nested table variable, or simply, “table” for short.
- When a nested table variable is declared, the nested table initially does not exist (it is a null collection). The null table must be initialized with a constructor. You can also initialize the table by using an assignment statement where the right-hand side of the assignment is an initialized table of the same type. Note: Initialization of a nested table is mandatory in Oracle, but optional in SPL.
- The key is a positive integer.
- The constructor establishes the number of elements in the table. The `EXTEND` method adds additional elements to the table. See Section 4.11 for information on collection methods. Note: Usage of the constructor to establish the number of elements in the table and usage of the `EXTEND` method to add additional elements to the table are mandatory in Oracle, but optional in SPL.
- The table can be sparse - there may be gaps in the assignment of values to keys.
- An attempt to reference a table element beyond its initialized or extended size will result in a `SUBSCRIPT_BEYOND_COUNT` exception.

The `TYPE IS TABLE` statement is used to define a nested table type within the declaration section of an SPL program.

```
TYPE tbltype IS TABLE OF { datatype | rectype | objtype };
```

`tbltype` is an identifier assigned to the nested table type. `datatype` is a scalar data type such as VARCHAR2 or NUMBER. `rectype` is a previously defined record type. `objtype` is a previously defined object type.

Note: You can use the CREATE TYPE command to define a nested table type that is available to all SPL programs in the database. See the CREATE TYPE command for more information.

In order to make use of the table, a variable must be declared of that nested table type. The following is the syntax for declaring a table variable.

```
table tbltype
```

`table` is an identifier assigned to the nested table. `tbltype` is the identifier of a previously defined nested table type.
A nested table is initialized using the nested table type’s constructor.

\[
tbltype ([
  \{ expr1 | NULL \} [, \{ expr2 | NULL \}] [, ... ] )
\]

\(tbltype\) is the identifier of the nested table type’s constructor, which has the same name as the nested table type. \(expr1, expr2, \ldots\) are expressions that are type-compatible with the element type of the table. If \(NULL\) is specified, the corresponding element is set to null. If the parameter list is empty, then an empty nested table is returned, which means there are no elements in the table. If the table is defined from an object type, then \(exprn\) must return an object of that object type. The object can be the return value of a function or the object type’s constructor, or the object can be an element of another nested table of the same type.

If a collection method other than \texttt{EXISTS} is applied to an uninitialized nested table, a \texttt{COLLECTION_IS_NULL} exception is thrown. See Section 4.11 for information on collection methods.

The following is an example of a constructor for a nested table:

```sql
DECLARE
  TYPE nested_typ IS TABLE OF CHAR(1);
  v_nested nested_typ := nested_typ('A','B');
```

An element of the table is referenced using the following syntax.

\(table(n)[.element ]\)

\(table\) is the identifier of a previously declared table. \(n\) is a positive integer. If the table type of \(table\) is defined from a record type or object type, then \(.[element ]\) must reference an individual field within the record type or attribute within the object type from which the nested table type is defined. Alternatively, the entire record or object can be referenced by omitting \(.[element ]\).

The following is an example of a nested table where it is known that there will be four elements.

```sql
DECLARE
  TYPE dname_tbl_typ IS TABLE OF VARCHAR2(14);
  dname_tbl dname_tbl_typ;
  CURSOR dept_cur IS SELECT dname FROM dept ORDER BY dname;
  i INTEGER := 0;
BEGIN
  dname_tbl := dname_tbl_typ(NULL, NULL, NULL, NULL);
  FOR r_dept IN dept_cur LOOP
    i := i + 1;
    dname_tbl(i) := r_dept.dname;
  END LOOP;
  DBMS_OUTPUT.PUT_LINE('DNAME');
  DBMS_OUTPUT.PUT_LINE('----------');
  FOR j IN 1..i LOOP
    DBMS_OUTPUT.PUT_LINE(dname_tbl(j));
  END LOOP;
END;
```

Copyright © 2007 - 2015 EnterpriseDB Corporation. All rights reserved.
The above example produces the following output:

DNAME
-------
ACCOUNTING
OPERATIONS
RESEARCH
SALES

The following example reads the first ten employee names from the emp table, stores them in a nested table, then displays the results from the table. The SPL code is written to assume that the number of employees to be returned is not known beforehand.

Note the creation of an empty table with the constructor emp_tbl_typ() as the first statement in the executable section of the anonymous block. The EXTEND collection method is then used to add an element to the table for each employee returned from the result set. See Section 4.11.4 for information on EXTEND.

The following is the output.

EMPNO  ENAME
------  ------
7369    SMITH
7499    ALLEN
7521    WARD
7566    JONES
7654    MARTIN
7698    BLAKE
7782    CLARK
7788    SCOTT
The following example shows how a nested table of an object type can be used. See Section 8 for information about object types and objects. First, an object type is created with attributes for the department name and location.

```
CREATE TYPE dept_obj_typ AS OBJECT (
  dname           VARCHAR2(14),
  loc             VARCHAR2(13)
);
```

The following anonymous block defines a nested table type whose element consists of the `dept_obj_typ` object type. A nested table variable is declared, initialized, and then populated from the `dept` table. Finally, the elements from the nested table are displayed.

```
DECLARE
  TYPE dept_tbl_typ IS TABLE OF dept_obj_typ;
  dept_tbl        dept_tbl_typ;
  CURSOR dept_cur IS SELECT dname, loc FROM dept ORDER BY dname;
  i               INTEGER := 0;
BEGIN
  dept_tbl := dept_tbl_typ(
    dept_obj_typ(NULL,NULL),
    dept_obj_typ(NULL,NULL),
    dept_obj_typ(NULL,NULL),
    dept_obj_typ(NULL,NULL)
  );
  FOR r_dept IN dept_cur LOOP
    i := i + 1;
    dept_tbl(i).dname := r_dept.dname;
    dept_tbl(i).loc   := r_dept.loc;
  END LOOP;
  DBMS_OUTPUT.PUT_LINE('DNAME          LOC');
  DBMS_OUTPUT.PUT_LINE('---------- ----------');
  FOR j IN 1..i LOOP
    DBMS_OUTPUT.PUT_LINE(RPAD(dept_tbl(j).dname,14) || ' ' || dept_tbl(j).loc);
  END LOOP;
END;
```

**Note:** The parameters comprising the nested table’s constructor, `dept_tbl_typ`, are calls to the object type’s constructor `dept_obj_typ`.

The following is the output from the anonymous block.

```
DNAME          LOC
---------- ----------
ACCOUNTING     NEW YORK
OPERATIONS     BOSTON
RESEARCH       DALLAS
SALES          CHICAGO
```
4.10.3 Varrays

A \textit{varray} or \textit{variable-size array} is a type of collection that associates a positive integer with a value. In many respects, it is similar to a nested table.

A varray has the following characteristics:

- A \textit{varray type} must be defined along with a maximum size limit. After the varray type is defined, \textit{varray variables} can be declared of that varray type. Data manipulation occurs using the varray variable, or simply, “varray” for short. The number of elements in the varray cannot exceed the maximum size limit established in the varray type definition.
- When a varray variable is declared, the varray initially does not exist (it is a null collection). The null varray must be initialized with a \textit{constructor}. You can also initialize the varray by using an assignment statement where the right-hand side of the assignment is an initialized varray of the same type.
- The key is a positive integer.
- The constructor establishes the number of elements in the varray, which must not exceed the maximum size limit. The \texttt{EXTEND} method can add additional elements to the varray up to the maximum size limit. See Section 4.11 for information on collection methods.
- Unlike a nested table, a varray cannot be sparse - there are no gaps in the assignment of values to keys.
- An attempt to reference a varray element beyond its initialized or extended size, but within the maximum size limit will result in a \texttt{SUBSCRIPT_BEYOND_COUNT} exception.
- An attempt to reference a varray element beyond the maximum size limit or extend a varray beyond the maximum size limit will result in a \texttt{SUBSCRIPT_OUTSIDE_LIMIT} exception.

The \texttt{TYPE IS VARRAY} statement is used to define a varray type within the declaration section of an SPL program.

```
    TYPE varraytype IS { VARRAY | VARYING ARRAY } (maxsize)
       OF { datatype | objtype };
```

\texttt{varraytype} is an identifier assigned to the varray type. \texttt{datatype} is a scalar data type such as \texttt{VARCHAR2} or \texttt{NUMBER}. \texttt{maxsize} is the maximum number of elements permitted in varrays of that type. \texttt{objtype} is a previously defined object type.

Note: You can use the \texttt{CREATE TYPE} command to define a nested table type that is available to all SPL programs in the database. See the \texttt{CREATE TYPE} command for more information.
In order to make use of the varray, a **variable** must be declared of that varray type. The following is the syntax for declaring a varray variable.

```
varray varraytype
```

*varray* is an identifier assigned to the varray. *varraytype* is the identifier of a previously defined varray type.

A varray is initialized using the varray type’s constructor.

```
varraytype ( [ { expr1 | NULL } [, { expr2 | NULL } ] ] [, ... ] )
```

*varraytype* is the identifier of the varray type’s constructor, which has the same name as the varray type. *expr1*, *expr2*, … are expressions that are type-compatible with the element type of the varray. If NULL is specified, the corresponding element is set to null. If the parameter list is empty, then an empty varray is returned, which means there are no elements in the varray. If the varray is defined from an object type, then *exprn* must return an object of that object type. The object can be the return value of a function or the return value of the object type’s constructor. The object can also be an element of another varray of the same varray type.

If a collection method other than EXISTs is applied to an uninitialized varray, a COLLECTION_IS_NULL exception is thrown. See Section 4.11 for information on collection methods.

The following is an example of a constructor for a varray:

```
DECLARE
    TYPE varray_typ IS VARRAY(2) OF CHAR(1);
    v_varray        varray_typ := varray_typ('A','B');
```

An element of the varray is referenced using the following syntax.

```
varray(n) [ . element ]
```

*varray* is the identifier of a previously declared varray. *n* is a positive integer. If the varray type of *varray* is defined from an object type, then [ . element ] must reference an attribute within the object type from which the varray type is defined. Alternatively, the entire object can be referenced by omitting [ . element ].

The following is an example of a varray where it is known that there will be four elements.

```
DECLARE
    TYPE dname_varray_typ IS VARRAY(4) OF VARCHAR2(14);
    dname_varray    dname_varray_typ;
    CURSOR dept_cur IS SELECT dname FROM dept ORDER BY dname;
```
The above example produces the following output:

```sql
i INTEGER := 0;
BEGIN
    dname_varray := dname_varray_typ(NULL, NULL, NULL, NULL);
    FOR r_dept IN dept_cur LOOP
        i := i + 1;
        dname_varray(i) := r_dept.dname;
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('DNAME');
    DBMS_OUTPUT.PUT_LINE('----------');
    FOR j IN 1..i LOOP
        DBMS_OUTPUT.PUT_LINE(dname_varray(j));
    END LOOP;
END;
```

DNAME  
----------  
ACCOUNTING  
OPERATIONS  
RESEARCH  
SALES
Collection methods are functions and procedures that provide useful information about a collection that can aid in the processing of data in the collection. The following sections discuss the collection methods supported by Advanced Server.

### 4.11.1 COUNT

COUNT is a method that returns the number of elements in a collection. The syntax for using COUNT is as follows:

```
collection.COUNT
```

`collection` is the name of a collection.

For a varray, COUNT always equals LAST.

The following example shows that an associative array can be sparsely populated (i.e., there are “gaps” in the sequence of assigned elements). COUNT includes only the elements that have been assigned a value.

```sql
DECLARE
    TYPE sparse_arr_typ IS TABLE OF NUMBER INDEX BY BINARY_INTEGER;
    sparse_arr sparse_arr_typ;
BEGIN
    sparse_arr(-100) := -100;
    sparse_arr(-10) := -10;
    sparse_arr(0) := 0;
    sparse_arr(10) := 10;
    sparse_arr(100) := 100;
    DBMS_OUTPUT.PUT_LINE('COUNT: ' || sparse_arr.COUNT);
END;
```

The following output shows that there are five populated elements included in COUNT.

```
COUNT: 5
```

### 4.11.2 DELETE

The DELETE method deletes entries from a collection. You can call the DELETE method in three different ways.

Use the first form of the DELETE method to remove all entries from a collection:

```
collection.DELETE
```

Use the second form of the DELETE method to remove the specified entry from a collection:
Use the third form of the **DELETE** method to remove the entries that are within the range specified by **first_subscript** and **last_subscript** (including the entries for the **first_subscript** and the **last_subscript**) from a collection.

```
collection.DELETE(first_subscript, last_subscript)
```

If **first_subscript** and **last_subscript** refer to non-existent elements, elements that are in the range between the specified subscripts are deleted. If **first_subscript** is greater than **last_subscript**, or if you specify a value of **NULL** for one of the arguments, **DELETE** has no effect.

Note that when you delete an entry, the subscript remains in the collection; you can reuse the subscript with an alternate entry. If you specify a subscript that does not exist in the call to the **DELETE** method, **DELETE** does not raise an exception.

The following example demonstrates using the **DELETE** method to remove the element with subscript 0 from the collection:

```
DECLARE
    TYPE sparse_arr_typ IS TABLE OF NUMBER INDEX BY BINARY_INTEGER;
    sparse_arr sparse_arr_typ;
    v_results VARCHAR2(50);
    v_sub NUMBER;
BEGIN
    sparse_arr(-100) := -100;
    sparse_arr(-10) := -10;
    sparse_arr(0) := 0;
    sparse_arr(10) := 10;
    sparse_arr(100) := 100;
    DBMS_OUTPUT.PUT_LINE('COUNT: ' || sparse_arr.COUNT);
    sparse_arr.DELETE(0);
    DBMS_OUTPUT.PUT_LINE('COUNT: ' || sparse_arr.COUNT);
    v_sub := sparse_arr.FIRST;
    WHILE v_sub IS NOT NULL LOOP
        IF sparse_arr(v_sub) IS NULL THEN
            v_results := v_results || 'NULL ';
        ELSE
            v_results := v_results || sparse_arr(v_sub) || ' ';
        END IF;
        v_sub := sparse_arr.NEXT(v_sub);
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('Results: ' || v_results);
END;
```

```
COUNT: 5
COUNT: 4
Results: -100 -10 10 100
```

**COUNT** indicates that before the **DELETE** method, there were 5 elements in the collection; after the **DELETE** method was invoked, the collection contains 4 elements.
### 4.11.3 EXISTs

The **EXISTS** method verifies that a subscript exists within a collection. **EXISTS** returns **TRUE** if the subscript exists; if the subscript does not exist, **EXISTS** returns **FALSE**. The method takes a single argument; the subscript that you are testing for. The syntax is:

```
collection.EXISTS(subscript)
```

*collection* is the name of the collection.

*subscript* is the value that you are testing for. If you specify a value of **NULL**, **EXISTS** returns **false**.

The following example verifies that subscript number 10 exists within the associative array:

```
DECLARE
    TYPE sparse_arr_typ IS TABLE OF NUMBER INDEX BY BINARY_INTEGER;
    sparse_arr sparse_arr_typ;
BEGIN
    sparse_arr(-100) := -100;
    sparse_arr(-10) := -10;
    sparse_arr(0) := 0;
    sparse_arr(10) := 10;
    sparse_arr(100) := 100;
    DBMS_OUTPUT.PUT_LINE('The index exists: ' ||
        CASE WHEN sparse_arr.exists(10) = TRUE THEN 'true' ELSE 'false' END);
END;
```

The index exists: true

Some collection methods raise an exception if you call them with a subscript that does not exist within the specified collection. Rather than raising an error, the **EXISTS** method returns a value of **FALSE**.

### 4.11.4 EXTEND

The **EXTEND** method increases the size of a collection. There are three variations of the **EXTEND** method. The first variation appends a single **NULL** element to a collection; the syntax for the first variation is:

```
collection.EXTEND
```

*collection* is the name of a collection.

The following example demonstrates using the **EXTEND** method to append a single, null element to a collection:

```
DECLARE
    TYPE sparse_arr_typ IS TABLE OF NUMBER;
```
The following example demonstrates using the **EXTEND** method to append multiple null elements to a collection:

```sql
DECLARE
    TYPE sparse_arr_typ IS TABLE OF NUMBER;
    sparse_arr sparse_arr_typ := sparse_arr_typ(-100,-10,0,10,100);
    v_results VARCHAR2(50);
BEGIN
    DBMS_OUTPUT.PUT_LINE('COUNT: ' || sparse_arr.COUNT);
    sparse_arr.EXTEND(3);
    DBMS_OUTPUT.PUT_LINE('COUNT: ' || sparse_arr.COUNT);
    FOR i IN sparse_arr.FIRST .. sparse_arr.LAST LOOP
        IF sparse_arr(i) IS NULL THEN
            v_results := v_results || 'NULL ';
        ELSE
            v_results := v_results || sparse_arr(i) || ' ';
        END IF;
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('Results: ' || v_results);
END;
COUNT: 5
COUNT: 8
Results: -100 -10 0 10 100 NULL NULL NULL
```

**COUNT** indicates that before the **EXTEND** method, there were 5 elements in the collection; after the **EXTEND** method was invoked, the collection contains 6 elements.

The second variation of the **EXTEND** method appends a specified number of elements to the end of a collection.

```
collection.EXTEND(count)
```

*collection* is the name of a collection.

*count* is the number of null elements added to the end of the collection.
COUNT indicates that before the EXTEND method, there were 5 elements in the collection; after the EXTEND method was invoked, the collection contains 8 elements.

The third variation of the EXTEND method appends a specified number of copies of a particular element to the end of a collection.

\[ collection.EXTEND(count, index\_number) \]

\[ \text{collection is the name of a collection.} \]

\[ \text{count is the number of elements added to the end of the collection.} \]

\[ \text{index\_number is the subscript of the element that is being copied to the collection.} \]

The following example demonstrates using the EXTEND method to append multiple copies of the second element to the collection:

```sql
DECLARE
    TYPE sparse_arr_typ IS TABLE OF NUMBER;
    sparse_arr sparse_arr_typ := sparse_arr_typ(-100,-10,0,10,100);
    v_results VARCHAR2(50);
BEGIN
    DBMS_OUTPUT.PUT_LINE('COUNT: ' || sparse_arr.COUNT);
    sparse_arr.EXTEND(3, 2);
    DBMS_OUTPUT.PUT_LINE('COUNT: ' || sparse_arr.COUNT);
    FOR i IN sparse_arr.FIRST .. sparse_arr.LAST LOOP
        IF sparse_arr(i) IS NULL THEN
            v_results := v_results || 'NULL ';
        ELSE
            v_results := v_results || sparse_arr(i) || ' ';
        END IF;
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('Results: ' || v_results);
END;
```

COUNT indicates that before the EXTEND method, there were 5 elements in the collection; after the EXTEND method was invoked, the collection contains 8 elements.

Note: The EXTEND method cannot be used on a null or empty collection.

4.11.5 FIRST

FIRST is a method that returns the subscript of the first element in a collection. The syntax for using FIRST is as follows:

\[ collection.FIRST \]
\textit{collection} is the name of a collection.

The following example displays the first element of the associative array.

\begin{verbatim}
DECLARE
    TYPE sparse_arr_typ IS TABLE OF NUMBER INDEX BY BINARY_INTEGER;
    sparse_arr sparse_arr_typ;
BEGIN
    sparse_arr(-100) := -100;
    sparse_arr(-10) := -10;
    sparse_arr(0) := 0;
    sparse_arr(10) := 10;
    sparse_arr(100) := 100;
    DBMS_OUTPUT.PUT_LINE('FIRST element: ' || sparse_arr(sparse_arr.FIRST));
END;

FIRST element: -100
\end{verbatim}

\subsection{4.11.6 \textbf{LAST}}

\textbf{LAST} is a method that returns the subscript of the last element in a collection. The syntax for using \textbf{LAST} is as follows:

\begin{verbatim}
        collection.LAST
\end{verbatim}

\textit{collection} is the name of a collection.

The following example displays the last element of the associative array.

\begin{verbatim}
DECLARE
    TYPE sparse_arr_typ IS TABLE OF NUMBER INDEX BY BINARY_INTEGER;
    sparse_arr sparse_arr_typ;
BEGIN
    sparse_arr(-100) := -100;
    sparse_arr(-10) := -10;
    sparse_arr(0) := 0;
    sparse_arr(10) := 10;
    sparse_arr(100) := 100;
    DBMS_OUTPUT.PUT_LINE('LAST element: ' || sparse_arr(sparse_arr.LAST));
END;

LAST element: 100
\end{verbatim}

\subsection{4.11.7 \textbf{LIMIT}}

\textbf{LIMIT} is a method that returns the maximum number of elements permitted in a collection. \textbf{LIMIT} is applicable only to varrays. The syntax for using \textbf{LIMIT} is as follows:

\begin{verbatim}
        collection.LIMIT
\end{verbatim}

\textit{collection} is the name of a collection.
For an initialized varray, LIMIT returns the maximum size limit determined by the varray type definition. If the varray is uninitialized (that is, it is a null varray), an exception is thrown.

For an associative array or an initialized nested table, LIMIT returns NULL. If the nested table is uninitialized (that is, it is a null nested table), an exception is thrown.

4.11.8 NEXT

NEXT is a method that returns the subscript that follows a specified subscript. The method takes a single argument; the subscript that you are testing for.

\[
\text{collection}.\text{NEXT(subscript)}
\]

\(\text{collection}\) is the name of the collection.

If the specified subscript is less than the first subscript in the collection, the function returns the first subscript. If the subscript does not have a successor, NEXT returns NULL. If you specify a NULL subscript, PRIOR does not return a value.

The following example demonstrates using NEXT to return the subscript that follows subscript 10 in the associative array, sparse_arr:

```
DECLARE
    TYPE sparse_arr_typ IS TABLE OF NUMBER INDEX BY BINARY_INTEGER;
    sparse_arr sparse_arr_typ;
BEGIN
    sparse_arr(-100) := -100;
    sparse_arr(-10) := -10;
    sparse_arr(0) := 0;
    sparse_arr(10) := 10;
    sparse_arr(100) := 100;
    DBMS_OUTPUT.PUT_LINE('NEXT element: ' || sparse_arr.next(10));
END;
```

NEXT element: 100

4.11.9 PRIOR

The PRIOR method returns the subscript that precedes a specified subscript in a collection. The method takes a single argument; the subscript that you are testing for. The syntax is:

\[
\text{collection}.\text{PRIOR(subscript)}
\]

\(\text{collection}\) is the name of the collection.
If the subscript specified does not have a predecessor, PRIOR returns NULL. If the specified subscript is greater than the last subscript in the collection, the method returns the last subscript. If you specify a NULL subscript, PRIOR does not return a value.

The following example returns the subscript that precedes subscript 100 in the associative array, sparse_arr:

```sql
DECLARE
    TYPE sparse_arr_typ IS TABLE OF NUMBER INDEX BY BINARY_INTEGER;
    sparse_arr sparse_arr_typ;
BEGIN
    sparse_arr(-100) := -100;
    sparse_arr(-10)  := -10;
    sparse_arr(0)    := 0;
    sparse_arr(10)   := 10;
    sparse_arr(100)  := 100;
    DBMS_OUTPUT.PUT_LINE('PRIOR element: ' || sparse_arr.prior(100));
END;
PRIOR element: 10
```

### 4.11.10 TRIM

The TRIM method removes an element or elements from the end of a collection. The syntax for the TRIM method is:

```sql
collection.TRIM((count))
```

*collection* is the name of a collection.

*count* is the number of elements removed from the end of the collection. Advanced Server will return an error if *count* is less than 0 or greater than the number of elements in the collection.

The following example demonstrates using the TRIM method to remove an element from the end of a collection:

```sql
DECLARE
    TYPE sparse_arr_typ IS TABLE OF NUMBER;
    sparse_arr sparse_arr_typ := sparse_arr_typ(-100,-10,0,10,100);
BEGIN
    DBMS_OUTPUT.PUT_LINE('COUNT: ' || sparse_arr.COUNT);
    sparse_arr.trim;
    DBMS_OUTPUT.PUT_LINE('COUNT: ' || sparse_arr.COUNT);
END;
COUNT: 5
COUNT: 4
```
COUNT indicates that before the TRIM method, there were 5 elements in the collection; after the TRIM method was invoked, the collection contains 4 elements.

You can also specify the number of elements to remove from the end of the collection with the TRIM method:

```
DECLARE
    TYPE sparse_arr_typ IS TABLE OF NUMBER;
    sparse_arr sparse_arr_typ := sparse_arr_typ(-100,-10,0,10,100);
    v_results VARCHAR2(50);
BEGIN
    DBMS_OUTPUT.PUT_LINE('COUNT: ' || sparse_arr.COUNT);
    sparse_arr.TRIM(2);
    DBMS_OUTPUT.PUT_LINE('COUNT: ' || sparse_arr.COUNT);
    FOR i IN sparse_arr.FIRST .. sparse_arr.LAST LOOP
        IF sparse_arr(i) IS NULL THEN
            v_results := v_results || 'NULL ';
        ELSE
            v_results := v_results || sparse_arr(i) || ' ';
        END IF;
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('Results: ' || v_results);
END;
COUNT: 5
COUNT: 3
Results: -100 -10 0
```

COUNT indicates that before the TRIM method, there were 5 elements in the collection; after the TRIM method was invoked, the collection contains 3 elements.
4.12 Working with Collections

Collection operators allow you to transform, query and manipulate the contents of a collection.

4.12.1 TABLE()

Use the TABLE() function to transform the members of an array into a set of rows. The signature is:

    TABLE(collection_value)

Where:

    collection_value

    collection_value is an expression that evaluates to a value of collection type.

The TABLE() function expands the nested contents of a collection into a table format. You can use the TABLE() function anywhere you use a regular table expression.

The TABLE() function returns a SETOF ANYELEMENT (a set of values of any type). For example, if the argument passed to this function is an array of dates, TABLE() will return a SETOF dates. If the argument passed to this function is an array of paths, TABLE() will return a SETOF paths.

You can use the TABLE() function to expand the contents of a collection into table form:

    postgres=# SELECT * FROM TABLE(monthly_balance(445.00, 980.20, 552.00));

    monthly_balance
    ---------------
    445.00
    980.20
    552.00
    (3 rows)

4.12.2 Using the MULTISET UNION Operator

The MULTISET UNION operator combines two collections to form a third collection. The signature is:

    coll_1 MULTISET UNION [ALL | DISTINCT] coll_2

    coll_1 and coll_2 specify the names of the collections to combine.
Include the ALL keyword to specify that duplicate elements (elements that are present in both \texttt{coll_1} and \texttt{coll_2}) should be represented in the result, once for each time they are present in the original collections. This is the default behavior of \texttt{MULTISET UNION}.

Include the DISTINCT keyword to specify that duplicate elements should be included in the result only once.

The following example demonstrates using the \texttt{MULTISET UNION} operator to combine two collections (\texttt{collection_1} and \texttt{collection_2}) into a third collection (\texttt{collection_3}):


declare
    type int_arr_typ is table of number(2);
    collection_1 int_arr_typ;
    collection_2 int_arr_typ;
    collection_3 int_arr_typ;
    v_results varchar2(50);
begin
    collection_1 := int_arr_typ(10,20,30);
    collection_2 := int_arr_typ(30,40);
    collection_3 := collection_1 multiset union all collection_2;
    dbms_output.put_line('count: ' || collection_3.count);
    for i in collection_3.first .. collection_3.last loop
        if collection_3(i) is null then
            v_results := v_results || 'null ';
        else
            v_results := v_results || collection_3(i) || ' ';
        end if;
    end loop;
    dbms_output.put_line('results: ' || v_results);
end;

count: 5
results: 10 20 30 30 40

The resulting collection includes one entry for each element in \texttt{collection_1} and \texttt{collection_2}. If the \texttt{DISTINCT} keyword is used, the results are the following:

declare
    type int_arr_typ is table of number(2);
    collection_1 int_arr_typ;
    collection_2 int_arr_typ;
    collection_3 int_arr_typ;
    v_results varchar2(50);
begin
    collection_1 := int_arr_typ(10,20,30);
    collection_2 := int_arr_typ(30,40);
    collection_3 := collection_1 multiset union distinct collection_2;
    dbms_output.put_line('count: ' || collection_3.count);
    for i in collection_3.first .. collection_3.last loop
        if collection_3(i) is null then
            v_results := v_results || 'null ';
        else
            v_results := v_results || collection_3(i) || ' '; (1)
        end if;
    end loop;
    dbms_output.put_line('results: ' || v_results);
end;

count: 5
results: 10 20 30 40
The resulting collection includes only those members with distinct values. Note in the following example that the MULTISET UNION DISTINCT operator also removes duplicate entries that are stored within the same collection:

```sql
DECLARE
    TYPE int_arr_typ IS TABLE OF NUMBER(2);
    collection_1    int_arr_typ;
    collection_2    int_arr_typ;
    collection_3    int_arr_typ;
    v_results       VARCHAR2(50);
BEGIN
    collection_1 := int_arr_typ(10,20,30,30);
    collection_2 := int_arr_typ(40,50);
    collection_3 := collection_1 MULTISET UNION DISTINCT collection_2;
    DBMS_OUTPUT.PUT_LINE('COUNT: ' || collection_3.COUNT);
    FOR i IN collection_3.FIRST .. collection_3.LAST LOOP
        IF collection_3(i) IS NULL THEN
            v_results := v_results || 'NULL ';
        ELSE
            v_results := v_results || collection_3(i) || ' ';
        END IF;
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('Results: ' || v_results);
END;
COUNT: 5
Results: 10 20 30 40 50
```

### 4.12.3 Using the FORALL Statement

Collections can be used to more efficiently process DML commands by passing all the values to be used for repetitive execution of a DELETE, INSERT, or UPDATE command in one pass to the database server rather than re-iteratively invoking the DML command with new values. The DML command to be processed in such a manner is specified with the `FORALL` statement. In addition, one or more collections are given in the DML command where different values are to be substituted each time the command is executed.

```
FORALL index IN lower_bound .. upper_bound
    { insert_stmt | update_stmt | delete_stmt };
```

`index` is the position in the collection given in the `insert_stmt`, `update_stmt`, or `delete_stmt` DML command that iterates from the integer value given as `lower_bound` up to and including `upper_bound`.

**Note:** If an exception occurs during any iteration of the `FORALL` statement, all updates that occurred since the start of the execution of the `FORALL` statement are automatically
rolled back. This behavior is not Oracle compatible. Oracle allows explicit use of the COMMIT or ROLLBACK commands to control whether or not to commit or roll back updates that occurred prior to the exception.

The FORALL statement creates a loop—each iteration of the loop increments the index variable (you typically use the index within the loop to select a member of a collection). The number of iterations is controlled by the lower_bound .. upper_bound clause. The loop is executes once for each integer between the lower_bound and upper_bound (inclusive) and the index is incremented by one for each iteration. For example:

```
FORALL i IN 2 .. 5
```

Creates a loop that executes four times—in the first iteration, the index(i) is set to the value 2; in the second iteration, the index is set to the value 3, and so on. The loop executes for the value 5 and then terminates.

The following example creates a table (emp_copy) that is an empty copy of the emp table. The example declares a type (emp_tbl) that is an array where each element in the array is of composite type, composed of the column definitions used to create the table, emp. The example also creates an index on the emp_tbl type.

```
t_emp is an associative array, of type emp_tbl. The SELECT statement uses the BULK COLLECT INTO command to populate the t_emp array. After the t_emp array is populated, the FORALL statement iterates through the values (i) in the t_emp array index and inserts a row for each record into emp_copy.
```

```
CREATE TABLE emp_copy(LIKE emp);
DECLARE
    TYPE emp_tbl IS TABLE OF emp%ROWTYPE INDEX BY BINARY_INTEGER;
    t_emp emp_tbl;
BEGIN
    SELECT * FROM emp BULK COLLECT INTO t_emp;
    FORALL i IN t_emp.FIRST .. t_emp.LAST
    INSERT INTO emp_copy VALUES t_emp(i);
END;
```

The following example uses a FORALL statement to update the salary of three employees:

```
DECLARE
    TYPE empno_tbl  IS TABLE OF emp.empno%TYPE INDEX BY BINARY_INTEGER;
    TYPE sal_tbl    IS TABLE OF emp.ename%TYPE INDEX BY BINARY_INTEGER;
    t_empno         EMPNO_TBL;
    t_sal           SAL_TBL;
BEGIN
```

Copyright © 2007 - 2015 EnterpriseDB Corporation. All rights reserved.
t_empno(1) := 9001;
t_sal(1) := 3350.00;
t_empno(2) := 9002;
t_sal(2) := 2000.00;
t_empno(3) := 9003;
t_sal(3) := 4100.00;
FORALL i IN t_empno.FIRST..t_empno.LAST
    UPDATE emp SET sal = t_sal(i) WHERE empno = t_empno(i);
END;

SELECT * FROM emp WHERE empno > 9000;

empno | ename  |   job   | mgr | hiredate |   sal   | comm | deptno
-------+--------+---------+-----+----------+--------+------+-------
   9001 | JONES  | ANALYST |     |          | 3350.00 |      |     40
   9002 | LARSEN | CLERK   |     |          | 2000.00 |      |     40
   9003 | WILSON | MANAGER |     |          | 4100.00 |      |     40
(3 rows)

The following example deletes three employees in a FORALL statement:

DECLARE
    TYPE empno_tbl  IS TABLE OF emp.empno%TYPE INDEX BY BINARY_INTEGER;
t_empno EMPNO_TBL;
BEGIN
    t_empno(1) := 9001;
t_empno(2) := 9002;
t_empno(3) := 9003;
    FORALL i IN t_empno.FIRST..t_empno.LAST
        DELETE FROM emp WHERE empno = t_empno(i);
END;

SELECT * FROM emp WHERE empno > 9000;

empno | ename | job | mgr | hiredate | sal | comm | deptno
-------+-------+-----+-----+----------+-----+------+-------
(0 rows)
4.12.4 Using the BULK COLLECT Clause

SQL commands that return a result set consisting of a large number of rows may not be operating as efficiently as possible due to the constant context switching that must occur between the database server and the client in order to transfer the entire result set. This inefficiency can be mitigated by using a collection to gather the entire result set in memory which the client can then access. The BULK COLLECT clause is used to specify the aggregation of the result set into a collection.

The BULK COLLECT clause can be used with the SELECT INTO, FETCH INTO and EXECUTE IMMEDIATE commands, and with the RETURNING INTO clause of the DELETE, INSERT, and UPDATE commands. Each of these is illustrated in the following sections.

4.12.4.1 SELECT BULK COLLECT

The BULK COLLECT clause can be used with the SELECT INTO statement as follows. (Refer to Section 4.4.3 for additional information on the SELECT INTO statement.)

```
SELECT select_expressions BULK COLLECT INTO collection [, ...] FROM ...;
```

If a single collection is specified, then collection may be a collection of a single field, or it may be a collection of a record type. If more than one collection is specified, then each collection must consist of a single field. select_expressions must match in number, order, and type-compatibility all fields in the target collections.

The following example shows the use of the BULK COLLECT clause where the target collections are associative arrays consisting of a single field.

```sql
DECLARE
    TYPE empno_tbl    IS TABLE OF emp.empno%TYPE    INDEX BY BINARY_INTEGER;
    TYPE ename_tbl    IS TABLE OF emp.ename%TYPE    INDEX BY BINARY_INTEGER;
    TYPE job_tbl      IS TABLE OF emp.job%TYPE      INDEX BY BINARY_INTEGER;
    TYPE hiredate_tbl IS TABLE OF emp.hiredate%TYPE INDEX BY BINARY_INTEGER;
    TYPE sal_tbl      IS TABLE OF emp.sal%TYPE      INDEX BY BINARY_INTEGER;
    TYPE comm_tbl     IS TABLE OF emp.comm%TYPE     INDEX BY BINARY_INTEGER;
    TYPE deptno_tbl   IS TABLE OF emp.deptno%TYPE   INDEX BY BINARY_INTEGER;
    t_empno           EMPNO_TBL;
    t_ename           ENAME_TBL;
    t_job             JOB_TBL;
    t_hiredate        HIREDATE_TBL;
    t_sal             SAL_TBL;
    t_comm            COMM_TBL;
    t_deptno          DEPTNO_TBL;
BEGIN
    SELECT empno, ename, job, hiredate, sal, comm, deptno BULK COLLECT
    INTO t_empno, t_ename, t_job, t_hiredate, t_sal, t_comm, t_deptno
```
FROM emp;
DBMS_OUTPUT.PUT_LINE('EMPNO  ENAME    JOB        HIREDATE    ' ||
'SAL        ' || 'COMM      DEPTNO');
DBMS_OUTPUT.PUT_LINE('----- -------- -------- -------- ' ||
'T---------' || '-------- ');
FOR i IN 1..t_empno.COUNT LOOP
DBMS_OUTPUT.PUT_LINE(t_empno(i) || ' ' ||
RPAD(t_ename(i),8) || ' ' ||
RPAD(t_job(i),10) || ' ' ||
TO_CHAR(t_hiredate(i),'DD-MON-YY') || ' ' ||
TO_CHAR(t_sal(i),'99,999.99') || ' ' ||
 TO_CHAR(NVL(t_comm(i),0),'99,999.99') || ' ' ||
t_deptno(i));
END LOOP;
END;
EMPNO  ENAME    JOB        HIREDATE    SAL        COMM      DEPTNO
----- -------- -------- -------- ------ -------- ------
7369    SMITH    CLERK      17-DEC-80   800.00        .00  20
7499    ALLEN    SALESMAN   20-FEB-81   1,600.00     300.00  30
7521    WARD     SALESMAN   22-FEB-81   1,250.00    500.00  30
7566    JONES    MANAGER    02-APR-81   2,975.00        .00  20
7654    MARTIN   SALESMAN   28-SEP-81   1,250.00    1,400.00  30
7698    BLAKE    MANAGER    01-MAY-81   2,850.00        .00  30
7782    CLARK    MANAGER    09-JUN-81   2,450.00        .00  10
7788    SCOTT    ANALYST    19-APR-87   3,000.00        .00  20
7839    KING     PRESIDENT  17-NOV-81   5,000.00        .00 10
7844    TURNER   SALES MAN  08-SEP-81   1,500.00        .00  30
7876    ADAMS    CLERK      23-MAY-87   1,100.00        .00  20
7900    JAMES    CLERK      03-DEC-81   950.00        .00  30
7902    FORD     ANALYST    03-DEC-81   3,000.00        .00  20
7934    MILLER   CLERK      23-JAN-82   1,300.00        .00  10

The following example produces the same result, but uses an associative array on a
record type defined with the %ROWTYPE attribute.

DECLARE
  TYPE emp_tbl IS TABLE OF emp%ROWTYPE INDEX BY BINARY_INTEGER;
  t_emp IS emp_tbl;
BEGIN
SELECT * BULK COLLECT INTO t_emp FROM emp;
DBMS_OUTPUT.PUT_LINE('EMPNO  ENAME    JOB        HIREDATE    ' ||
'SAL        ' || 'COMM      DEPTNO');
DBMS_OUTPUT.PUT_LINE('----- -------- -------- -------- ' ||
'T---------' || '-------- ');
FOR i IN 1..t_emp.COUNT LOOP
DBMS_OUTPUT.PUT_LINE(t_emp(i).empno || ' ' ||
RPAD(t_emp(i).ename,8) || ' ' ||
RPAD(t_emp(i).job,10) || ' ' ||
TO_CHAR(t_emp(i).hiredate,'DD-MON-YY') || ' ' ||
TO_CHAR(t_emp(i).sal,'99,999.99') || ' ' ||
TO_CHAR(NVL(t_emp(i).comm,0),'99,999.99') || ' ' ||
t_emp(i).deptno);
END LOOP;
END;
4.12.4.2 FETCH BULK COLLECT

The BULK COLLECT clause can be used with a FETCH statement. (See Section 4.8.3 for information on the FETCH statement.) Instead of returning a single row at a time from the result set, the FETCH BULK COLLECT will return all rows at once from the result set into the specified collection unless restricted by the LIMIT clause.

```
FETCH name BULK COLLECT INTO collection [, ...] [ LIMIT n ];
```

If a single collection is specified, then collection may be a collection of a single field, or it may be a collection of a record type. If more than one collection is specified, then each collection must consist of a single field. The expressions in the SELECT list of the cursor identified by name must match in number, order, and type-compatibility all fields in the target collections. If LIMIT n is specified, the number of rows returned into the collection on each FETCH will not exceed n.

The following example uses the FETCH BULK COLLECT statement to retrieve rows into an associative array.

```
DECLARE
    TYPE emp_tbl IS TABLE OF emp%ROWTYPE INDEX BY BINARY_INTEGER;
    t_emp           EMP_TBL;
    CURSOR emp_cur IS SELECT * FROM emp;
BEGIN
    OPEN emp_cur;
    FETCH emp_cur BULK COLLECT INTO t_emp;
    CLOSE emp_cur;
    DBMS_OUTPUT.PUT_LINE('EMPNO  ENAME    JOB        HIREDATE    ' ||
                          'SAL        ' || 'COMM      DEPTNO');
    DBMS_OUTPUT.PUT_LINE('-----  ------    --------    --------    ' ||
                          '-------    ------    ------');
    FOR i IN 1..t_emp.COUNT LOOP
        DBMS_OUTPUT.PUT_LINE(t_emp(i).empno || '   ' ||
                                RPAD(t_emp(i).ename,8) || ' ' ||
                                RPAD(t_emp(i).job,10) || ' ' ||
                                TO_CHAR(t_emp(i).hiredate,'DD-MON-YY') || ' ' ||
                                TO_CHAR(t_emp(i).sal,'99,999.99') || ' ' ||
                                TO_CHAR(NVL(t_emp(i).comm,0),'99,999.99') || ' ' ||
                                t_emp(i).deptno);
    END LOOP;
END;
```

4.12.4.3 EXECUTE IMMEDIATE BULK COLLECT

The BULK COLLECT clause can be used with a EXECUTE IMMEDIATE statement to specify a collection to receive the returned rows.

```sql
EXECUTE IMMEDIATE 'sql_expression;
BULK COLLECT INTO collection [, ...]
[USING {[bind_type] bind_argument} [, ...]]
```

`collection` specifies the name of a collection.

`bind_type` specifies the parameter mode of the `bind_argument`.

- A `bind_type` of **IN** specifies that the `bind_argument` contains a value that is passed to the `sql_expression`.
- A `bind_type` of **OUT** specifies that the `bind_argument` receives a value from the `sql_expression`.
- A `bind_type` of **IN OUT** specifies that the `bind_argument` is passed to `sql_expression`, and then stores the value returned by `sql_expression`.

`bind_argument` specifies a parameter that contains a value that is either passed to the `sql_expression` (specified with a `bind_type` of **IN**), or that receives a value from the `sql_expression` (specified with a `bind_type` of **OUT**), or both (specified with a `bind_type` of **IN OUT**).

If a single collection is specified, then `collection` may be a collection of a single field, or a collection of a record type; if more than one collection is specified, each `collection` must consist of a single field.
4.12.4.4  RETURNING BULK COLLECT

The **BULK COLLECT** clause can be added to the **RETURNING INTO** clause of a **DELETE**, **INSERT**, or **UPDATE** command. (See Section 4.4.7 for information on the **RETURNING INTO** clause.)

```
{ insert | update | delete }
RETURNING { * | expr_1 [, expr_2 ] ...}
BULK COLLECT INTO collection [, ...];
```

**insert, update, and delete** are the **INSERT**, **UPDATE**, and **DELETE** commands as described in Sections 0, 0, and 0, respectively. If a single collection is specified, then **collection** may be a collection of a single field, or it may be a collection of a record type. If more than one collection is specified, then each **collection** must consist of a single field. The expressions following the **RETURNING** keyword must match in number, order, and type-compatibility all fields in the target collections. If * is specified, then all columns in the affected table are returned. (Note that the use of * is a Postgres Plus Advanced Server extension and is not Oracle compatible.)

The **clerkemp** table created by copying the **emp** table is used in the remaining examples in this section as shown below.

```
CREATE TABLE clerkemp AS SELECT * FROM emp WHERE job = 'CLERK';
SELECT * FROM clerkemp;
empno | ename  |  job  | mgr  |      hiredate      |   sal   | comm | deptno
-------+--------+-------+------|--------------------+--------+------|--------
7369  | SMITH  | CLERK | 7902 | 17-DEC-80 00:00:00 |  800.00 |      |     20
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
7934  | MILLER | CLERK | 7782 | 23-JAN-82 00:00:00 | 1300.00 |      |     10
```

The following example increases everyone’s salary by 1.5, stores the employees’ numbers, names, and new salaries in three associative arrays, and finally, displays the contents of these arrays.

```
DECLARE
  TYPE empno_tbl IS TABLE OF emp.empno%TYPE INDEX BY BINARY_INTEGER;
  TYPE ename_tbl IS TABLE OF emp.ename%TYPE INDEX BY BINARY_INTEGER;
  TYPE sal_tbl   IS TABLE OF emp.sal%TYPE   INDEX BY BINARY_INTEGER;
  t_empno         EMPNO_TBL;
  t_ename         ENAME_TBL;
  t_sal           SAL_TBL;
BEGIN
  UPDATE clerkemp SET sal = sal * 1.5 RETURNING empno, ename, sal
  BULK COLLECT INTO t_empno, t_ename, t_sal;
  DBMS_OUTPUT.PUT_LINE('EMPNO  ENAME      SAL');
  DBMS_OUTPUT.PUT_LINE('----- ------- --------
FOR i IN 1..t_empno.COUNT LOOP
  DBMS_OUTPUT.PUT_LINE(t_empno(i) || ' ' || RPAD(t_ename(i),8) || ' ' || t_sal(i));
END LOOP;
```

The following example increases everyone’s salary by 1.5, stores the employees’ numbers, names, and new salaries in three associative arrays, and finally, displays the contents of these arrays.
The following example performs the same functionality as the previous example, but uses a single collection defined with a record type to store the employees’ numbers, names, and new salaries.

```
DECLARE
    TYPE emp_rec IS RECORD (
        empno   emp.empno%TYPE,
        ename   emp.ename%TYPE,
        sal     emp.sal%TYPE
    );
    TYPE emp_tbl IS TABLE OF emp_rec INDEX BY BINARY_INTEGER;
    t_emp    EMP_TBL;
BEGIN
    UPDATE clerkemp SET sal = sal * 1.5 RETURNING empno, ename, sal
    BULK COLLECT INTO t_emp;
    DBMS_OUTPUT.PUT_LINE('EMPNO  ENAME      SAL        ');
    FOR i IN 1..t_emp.COUNT LOOP
        DBMS_OUTPUT.PUT_LINE(t_emp(i).empno || '   ' ||
            RPAD(t_emp(i).ename,8) || ' ' ||
            TO_CHAR(t_emp(i).sal,'99,999.99')));
    END LOOP;
END;
```

```
EMPNO  E       S
------  -------- --------
7369   SMITH    1,200.00
7876   ADAMS    1,650.00
7900   JAMES    1,425.00
7934   MILLER   1,950.00
```

The following example deletes all rows from the clerkemp table, and returns information on the deleted rows into an associative array, which is then displayed.

```
DECLARE
    TYPE emp_rec IS RECORD (
        empno   emp.empno%TYPE,
        ename   emp.ename%TYPE,
        job     emp.job%TYPE,
        hiredate emp.hiredate%TYPE,
        sal     emp.sal%TYPE,
        comm    emp.comm%TYPE,
        deptno  emp.deptno%TYPE
    );
    TYPE emp_tbl IS TABLE OF emp_rec INDEX BY BINARY_INTEGER;
    r_emp   EMP_TBL;
BEGIN
    DELETE FROM clerkemp RETURNING empno, ename, job, hiredate, sal,
        comm, deptno BULK COLLECT INTO r_emp;
```

The following example deletes all rows from the clerkemp table, and returns information on the deleted rows into an associative array, which is then displayed.
4.13 Errors and Messages

Use the `DBMS_OUTPUT.PUT_LINE` statement to report messages.

```sql
DBMS_OUTPUT.PUT_LINE ( message );
```

`message` is any expression evaluating to a string.

This example displays the message on the user's output display:

```sql
DBMS_OUTPUT.PUT_LINE('My name is John');
```

The special variables `SQLCODE` and `SQLERRM` contain a numeric code and a text message, respectively, that describe the outcome of the last SQL command issued. If any other error occurs in the program such as division by zero, these variables contain information pertaining to the error.
5 Triggers

This chapter describes triggers in Postgres Plus Advanced Server. As with procedures and functions, triggers are written in the SPL language.

5.1 Overview

A trigger is a named SPL code block that is associated with a table and stored in the database. When a specified event occurs on the associated table, the SPL code block is executed. The trigger is said to be fired when the code block is executed.

The event that causes a trigger to fire can be any combination of an insert, update, or deletion carried out on the table, either directly or indirectly. If the table is the object of a SQL INSERT, UPDATE, or DELETE command the trigger is directly fired assuming that the corresponding insert, update, or deletion event is defined as a triggering event. The events that fire the trigger are defined in the CREATE TRIGGER command.

A trigger can be fired indirectly if a triggering event occurs on the table as a result of an event initiated on another table. For example, if a trigger is defined on a table containing a foreign key defined with the ON DELETE CASCADE clause and a row in the parent table is deleted, all children of the parent would be deleted as well. If deletion is a triggering event on the child table, deletion of the children will cause the trigger to fire.

5.2 Types of Triggers

Postgres Plus Advanced Server supports both row-level and statement-level triggers. A row-level trigger fires once for each row that is affected by a triggering event. For example, if deletion is defined as a triggering event on a table and a single DELETE command is issued that deletes five rows from the table, then the trigger will fire five times, once for each row.

In contrast, a statement-level trigger fires once per triggering statement regardless of the number of rows affected by the triggering event. In the prior example of a single DELETE command deleting five rows, a statement-level trigger would fire only once.

The sequence of actions can be defined regarding whether the trigger code block is executed before or after the triggering statement, itself, in the case of statement-level triggers; or before or after each row is affected by the triggering statement in the case of row-level triggers.

In a before row-level trigger, the trigger code block is executed before the triggering action is carried out on each affected row. In a before statement-level trigger, the trigger code block is executed before the action of the triggering statement is carried out.
In an *after* row-level trigger, the trigger code block is executed after the triggering action is carried out on each affected row. In an *after* statement-level trigger, the trigger code block is executed after the action of the triggering statement is carried out.

### 5.3 Creating Triggers

The `CREATE TRIGGER` command defines and names a trigger that will be stored in the database.

#### Name

`CREATE TRIGGER` -- define a new trigger

#### Synopsis

```
CREATE [ OR REPLACE ] TRIGGER name
{ BEFORE | AFTER | INSTEAD OF}
{ INSERT | UPDATE | DELETE }
[ OR { INSERT | UPDATE | DELETE } ] [, ...]
ON table
[ FOR EACH ROW ]
[ WHEN condition ]
[ DECLARE
  declaration; [, ...] ]
BEGIN
  statement; [, ...]
[ EXCEPTION
  { WHEN exception [ OR exception ] [...] THEN
    statement; [, ...] } [, ...]
  ]
END
```

#### Description

`CREATE TRIGGER` defines a new trigger. `CREATE OR REPLACE TRIGGER` will either create a new trigger, or replace an existing definition.

The name of the new trigger must not match any existing trigger defined on the same table unless the intent is to update the definition of an existing trigger, in which case use `CREATE OR REPLACE TRIGGER`.

The trigger is created in the same schema as the table on which the triggering event is defined.

#### Parameters
name

The name of the trigger to create.

BEFORE | AFTER

Determines whether the trigger is fired before or after the triggering event.

INSERT | UPDATE | DELETE

Defines the triggering event.

table

The name of the table on which the triggering event occurs.

condition

(condition is a Boolean expression that determines if the trigger will actually be executed; if condition evaluates to TRUE, the trigger will fire.

If the trigger definition includes the FOR EACH ROW keywords, the WHEN clause can refer to columns of the old and/or new row values by writing OLD.column_name or NEW.column_name respectively. INSERT triggers cannot refer to OLD and DELETE triggers cannot refer to NEW.

If the trigger includes the INSTEAD OF keywords, it may not include a WHEN clause.

WHEN clauses cannot contain subqueries.

FOR EACH ROW

Determines whether the trigger should be fired once for every row affected by the triggering event, or just once per SQL statement. If specified, the trigger is fired once for every affected row (row-level trigger), otherwise the trigger is a statement-level trigger.

declaration

A variable, type, or REF CURSOR declaration.

statement

An SPL program statement. Note that a DECLARE - BEGIN - END block is considered an SPL statement unto itself. Thus, the trigger body may contain nested blocks.
exception

An exception condition name such as NO_DATA_FOUND, OTHERS, etc.

5.4 Trigger Variables

In the trigger code block, several special variables are available for use.

NEW

NEW is a pseudo-record name that refers to the new table row for insert and update operations in row-level triggers. This variable is not applicable in statement-level triggers and in delete operations of row-level triggers.

Its usage is: :NEW.column where column is the name of a column in the table on which the trigger is defined.

The initial content of :NEW.column is the value in the named column of the new row to be inserted or of the new row that is to replace the old one when used in a before row-level trigger. When used in an after row-level trigger, this value has already been stored in the table since the action has already occurred on the affected row.

In the trigger code block, :NEW.column can be used like any other variable. If a value is assigned to :NEW.column, in the code block of a before row-level trigger, the assigned value will be used in the new inserted or updated row.

OLD

OLD is a pseudo-record name that refers to the old table row for update and delete operations in row-level triggers. This variable is not applicable in statement-level triggers and in insert operations of row-level triggers.

Its usage is: :OLD.column where column is the name of a column in the table on which the trigger is defined.

The initial content of :OLD.column is the value in the named column of the row to be deleted or of the old row that is to be replaced by the new one when used in a before row-level trigger. When used in an after row-level trigger, this value is no longer stored in the table since the action has already occurred on the affected row.

In the trigger code block, :OLD.column can be used like any other variable. Assigning a value to :OLD.column, has no effect on the action of the trigger.
INSERTING

INSERTING is a conditional expression that returns TRUE if an insert operation fired the trigger, otherwise it returns FALSE.

UPDATING

UPDATING is a conditional expression that returns TRUE if an update operation fired the trigger, otherwise it returns FALSE.

DELETING

DELETING is a conditional expression that returns TRUE if a delete operation fired the trigger, otherwise it returns FALSE.

5.5 Transactions and Exceptions

A trigger is always executed as part of the same transaction within which the triggering statement is executing. When no exceptions occur within the trigger code block, the effects of any DML commands within the trigger are committed if and only if the transaction containing the triggering statement is committed. Therefore, if the transaction is rolled back, the effects of any DML commands within the trigger are also rolled back.

If an exception does occur within the trigger code block, but it is caught and handled in an exception section, the effects of any DML commands within the trigger are still rolled back nonetheless. The triggering statement itself, however, is not rolled back unless the application forces a roll back of the encapsulating transaction.

If an unhandled exception occurs within the trigger code block, the transaction that encapsulates the trigger is aborted and rolled back. Therefore the effects of any DML commands within the trigger and the triggering statement, itself are all rolled back.
5.6 Trigger Examples

The following sections illustrate an example of each type of trigger.

5.6.1 Before Statement-Level Trigger

The following is an example of a simple before statement-level trigger that displays a message prior to an insert operation on the emp table.

```sql
CREATE OR REPLACE TRIGGER emp_alert_trig
  BEFORE INSERT ON emp
BEGIN
  DBMS_OUTPUT.PUT_LINE('New employees are about to be added');
END;
```

The following INSERT is constructed so that several new rows are inserted upon a single execution of the command. For each row that has an employee id between 7900 and 7999, a new row is inserted with an employee id incremented by 1000. The following are the results of executing the command when three new rows are inserted.

```
INSERT INTO emp (empno, ename, deptno) SELECT empno + 1000, ename, 40
  FROM emp WHERE empno BETWEEN 7900 AND 7999;
```

```
New employees are about to be added

SELECT empno, ename, deptno FROM emp WHERE empno BETWEEN 8900 AND 8999;

EMPNO  ENAME          DEPTNO
----------  ----------  ----------
 8900  JAMES          40
 8902  FORD           40
 8934  MILLER         40
```

The message, New employees are about to be added, is displayed once by the firing of the trigger even though the result is the addition of three new rows.

5.6.2 After Statement-Level Trigger

The following is an example of an after statement-level trigger. Whenever an insert, update, or delete operation occurs on the emp table, a row is added to the empauditlog table recording the date, user, and action.

```sql
CREATE TABLE empauditlog (audit_date DATE,
  audit_user VARCHAR2(20),
  audit_desc VARCHAR2(20));
CREATE OR REPLACE TRIGGER emp_audit_trig
  AFTER INSERT OR UPDATE OR DELETE ON emp
BEGIN
  v_action VARCHAR2(20);
  IF INSERTING THEN
    v_action := 'INSERT';
  ELSE IF UPDATING THEN
    v_action := 'UPDATE';
  ELSE IF DELETING THEN
    v_action := 'DELETE';
  END IF;
  INSERT INTO empauditlog (audit_date, audit_user, audit_desc)
  VALUES (SYSDATE, :NEW.EMPNO, v_action);
END;
```

The message, New employees are about to be added, is displayed once by the firing of the trigger even though the result is the addition of three new rows.
v_action := 'Added employee(s)'
ELSIF UPDATING THEN
v_action := 'Updated employee(s)'
ELSIF DELETING THEN
v_action := 'Deleted employee(s)'
END IF;
INSERT INTO empauditlog VALUES (SYSDATE, USER, v_action);
END;

In the following sequence of commands, two rows are inserted into the emp table using two INSERT commands. The sal and comm columns of both rows are updated with one UPDATE command. Finally, both rows are deleted with one DELETE command.

INSERT INTO emp VALUES (9001,'SMITH','ANALYST',7782,SYSDATE,NULL,NULL,10);
INSERT INTO emp VALUES (9002,'JONES','CLERK',7782,SYSDATE,NULL,NULL,10);
UPDATE emp SET sal = 4000.00, comm = 1200.00 WHERE empno IN (9001, 9002);
DELETE FROM emp WHERE empno IN (9001, 9002);
SELECT TO_CHAR(AUDIT_DATE,'DD-MON-YY HH24:MI:SS') AS "AUDIT DATE",
       audit_user, audit_desc FROM empauditlog ORDER BY 1 ASC;
AUDIT DATE         AUDIT_USER           AUDIT_DESC
------------------ ------------------ ------------------
31-MAR-05 14:59:48 SYSTEM               Added employee(s)
31-MAR-05 15:00:07 SYSTEM               Added employee(s)
31-MAR-05 15:00:19 SYSTEM               Updated employee(s)
31-MAR-05 15:00:34 SYSTEM               Deleted employee(s)

The contents of the empauditlog table show how many times the trigger was fired — once each for the two inserts, once for the update (even though two rows were changed) and once for the deletion (even though two rows were deleted).

### 5.6.3 Before Row-Level Trigger

The following example is a before row-level trigger that calculates the commission of every new employee belonging to department 30 that is inserted into the emp table.

CREATE OR REPLACE TRIGGER emp_comm_trig
    BEFORE INSERT ON emp
    FOR EACH ROW
BEGIN
    IF :NEW.deptno = 30 THEN
        :NEW.comm := :NEW.sal * .4;
    END IF;
END;

The listing following the addition of the two employees shows that the trigger computed their commissions and inserted it as part of the new employee rows.

INSERT INTO emp VALUES (9005,'ROBERS','SALESMAN',7782,SYSDATE,3000.00,NULL,30);
INSERT INTO emp VALUES (9006,'ALLEN','SALESMAN',7782,SYSDATE,4500.00,NULL,30);
5.6.4 After Row-Level Trigger

The following example is an after row-level trigger. When a new employee row is inserted, the trigger adds a new row to the jobhist table for that employee. When an existing employee is updated, the trigger sets the enddate column of the latest jobhist row (assumed to be the one with a null enddate) to the current date and inserts a new jobhist row with the employee’s new information.

Finally, trigger adds a row to the empchglog table with a description of the action.

```sql
CREATE TABLE empchglog (
    chg_date DATE,
    chg_desc VARCHAR2(30)
);
CREATE OR REPLACE TRIGGER emp_chg_trig
    AFTER INSERT OR UPDATE OR DELETE ON emp
    FOR EACH ROW
DECLARE
    v_empno emp.empno%TYPE;
    v_deptno emp.deptno%TYPE;
    v_dname dept.dname%TYPE;
    v_action VARCHAR2(7);
    v_chgdesc jobhist.chgdesc%TYPE;
BEGIN
    IF INSERTING THEN
        v_action := 'Added';
        v_empno := :NEW.empno;
        v_deptno := :NEW.deptno;
        INSERT INTO jobhist VALUES (:NEW.empno, SYSDATE, NULL,
    ELSIF UPDATING THEN
        v_action := 'Updated';
        v_empno := :NEW.empno;
        v_deptno := :NEW.deptno;
        v_chgdesc := '';
        IF NVL(:OLD.ename, '-null-') != NVL(:NEW.ename, '-null-') THEN
            v_chgdesc := v_chgdesc || 'name, '
        END IF;
        IF NVL(:OLD.job, '-null-') != NVL(:NEW.job, '-null-') THEN
            v_chgdesc := v_chgdesc || 'job, '
        END IF;
        IF NVL(:OLD.sal, -1) != NVL(:NEW.sal, -1) THEN
            v_chgdesc := v_chgdesc || 'salary, '
        END IF;
        IF NVL(:OLD.comm, -1) != NVL(:NEW.comm, -1) THEN
            v_chgdesc := v_chgdesc || 'commission, '
        END IF;
        IF NVL(:OLD.deptno, -1) != NVL(:NEW.deptno, -1) THEN
            v_chgdesc := v_chgdesc || 'department, '
        END IF;
    END IF;
END;
```
v_chgdesc := 'Changed ' || RTRIM(v_chgdesc, ', ');
UPDATE jobhist SET enddate = SYSDATE WHERE empno = :OLD.empno
AND enddate IS NULL;
INSERT INTO jobhist VALUES (:NEW.empno, SYSDATE, NULL,
:NEW.job, :NEW.sal, :NEW.comm, :NEW.deptno, v_chgdesc);
ELSIF DELETING THEN
v_action := 'Deleted';
v_empno := :OLD.empno;
v_deptno := :OLD.deptno;
END IF;
INSERT INTO empchglog VALUES (SYSDATE,
  v_action || ' employee # ' || v_empno);
END;

In the first sequence of commands shown below, two employees are added using two
separate INSERT commands and then both are updated using a single UPDATE command.
The contents of the jobhist table shows the action of the trigger for each affected row -
two new hire entries for the two new employees and two changed commission records for
the updated commissions on the two employees. The empchglog table also shows the
trigger was fired a total of four times, once for each action on the two rows.

```
INSERT INTO emp VALUES (9003,'PETERS','ANALYST',7782,SYSDATE,5000.00,NULL,40);
INSERT INTO emp VALUES (9004,'AIKENS','ANALYST',7782,SYSDATE,4500.00,NULL,40);
UPDATE emp SET comm = sal * 1.1 WHERE empno IN (9003, 9004);
SELECT * FROM jobhist WHERE empno IN (9003, 9004);

EMPNO STARTDATE ENDDATE JOB TEAL SAL COMM DEPTNO CHGDESC
---------- --------- -------- ------ ------- ------ ------- ---------
9003 31-MAR-05 31-MAR-05 ANALYST      5000                40 New Hire
9004 31-MAR-05 31-MAR-05 ANALYST      4500                40 New Hire
9003 31-MAR-05 31-MAR-05 ANALYST      5000               5500 40 Changed
commission
9004 31-MAR-05 31-MAR-05 ANALYST      4500               4950 40 Changed
commission
```

Finally, both employees are deleted with a single DELETE command. The empchglog
table now shows the trigger was fired twice, once for each deleted employee.

```
DELETE FROM emp WHERE empno IN (9003, 9004);
SELECT * FROM empchglog;

CHG_DATE CHG_DESC
---------- -------------------
31-MAR-05 Added employee # 9003
31-MAR-05 Added employee # 9004
31-MAR-05 Updated employee # 9003
31-MAR-05 Updated employee # 9004
```
<table>
<thead>
<tr>
<th>Date</th>
<th>Action</th>
<th>Employee ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-MAR-05</td>
<td>Updated</td>
<td>9004</td>
</tr>
<tr>
<td>31-MAR-05</td>
<td>Deleted</td>
<td>9003</td>
</tr>
<tr>
<td>31-MAR-05</td>
<td>Deleted</td>
<td>9004</td>
</tr>
</tbody>
</table>
6 Packages

This chapter discusses the concept of packages in Postgres Plus Advanced Server. A package is a named collection of functions, procedures, variables, cursors, user-defined record types, and records that are referenced using a common qualifier – the package identifier. Packages have the following characteristics:

- Packages provide a convenient means of organizing the functions and procedures that perform a related purpose. Permission to use the package functions and procedures is dependent upon one privilege granted to the entire package. All of the package programs must be referenced with a common name.
- Certain functions, procedures, variables, types, etc. in the package can be declared as public. Public entities are visible and can be referenced by other programs that are given `EXECUTE` privilege on the package. For public functions and procedures, only their signatures are visible - the program names, parameters if any, and return types of functions). The SPL code of these function and procedures is not accessible to others, therefore applications that utilize a package are dependent only upon the information available in the signature – not in the procedural logic, itself.
- Other functions, procedures, variables, types, etc. in the package can be declared as private. Private entities can be referenced and using by function and procedures within the package, but not by other external applications. Private entities are for use only by programs within the package.
- Function and procedure names can be overloaded within a package. One or more functions/procedures can be defined with the same name, but with different signatures. This provides the capability to create identically named programs that perform the same job, but on different types of input.

6.1 Package Components

Packages consist of two main components:

- The package specification: This is the public interface, (these are the elements which can be referenced outside the package). We declare all database objects that are to be a part of our package within the specification.
- The package body: This contains the actual implementation of all the database objects declared within the package specification.

The package body implements the specifications in the package specification. It contains implementation details and private declarations which are invisible to the application. You can debug, enhance or replace a package body without changing the specifications. Similarly, you can change the body without recompiling the calling programs because the implementation details are invisible to the application.
6.1.1 Package Specification Syntax

The following is the syntax of the package specification:

```
CREATE [ OR REPLACE ] PACKAGE package_name
    [ AUTHID { DEFINER | CURRENT_USER } ]
    [ AS ]
    [ declaration; ] ...
    [ { PROCEDURE proc_name
        [ (parm1 IN | IN OUT | OUT ] datatype1
        [, parm2 [IN | IN OUT | OUT ] datatype2 ] ... ]);
    [PRAGMA RESTRICT_REFERENCES (proc_name, WNDS [, WNPS]
        [, RNDS] [, RNPS] [, TRUST] ) ];

    | FUNCTION func_name
    [ (parm1 IN | IN OUT | OUT ] datatype1
    [, parm2 [IN | IN OUT | OUT ] datatype2 ] ... ]
    RETURN return_type; } ...
    [PRAGMA RESTRICT_REFERENCES (func_name, WNDS [, WNPS]
        [, RNDS] [, RNPS] [, TRUST] ) ];

END [ package_name ];
```

`package_name` is an identifier assigned to the package. If the AUTHID clause is omitted or DEFINER is specified, the rights of the package owner are used to determine access privileges to database objects. If CURRENT_USER is specified, the rights of the current user executing a program in the package are used to determine access privileges.

`declaration` is an identifier of a public variable. A public variable can be accessed from outside of the package using the syntax `package_name.variable`. There can be none, one, or more public variables. Public variable definitions must come before procedure or function declarations.

`declaration` can be any of the following:

- Variable Declaration
- Record Declaration (see Section 4.3.4)
- Collection Declaration (see Section 4.10)
- REF CURSOR and Cursor Variable Declaration
- TYPE Definitions for Records, Collections, and REF CURSORS
- Object Variable Declaration (see Section 8.4)

`proc_name` is an identifier of a public procedure. Public procedures can be invoked from outside of the package using the syntax `package_name.proc_name[(...)]`. If specified, `parm1, parm2,...` are the formal parameters of the procedure. `datatype1, datatype2,...` are the data types of `parm1, parm2,...` respectively. IN, IN OUT, and OUT are the possible parameter modes for each formal parameter. If none are specified, the default is IN.
func_name is an identifier of a public function. Public functions can be invoked from outside of the package using the syntax package_name.func_name[ (... )]. If specified, parm1, parm2,... are the formal parameters of the function. datatype1, datatype2,... are the data types of parm1, parm2,... respectively. IN, IN OUT, and OUT are the possible parameter modes for each formal parameter. If none are specified, the default is IN. return_type is the data type of the value the function returns. IN parameters can also be initialized with a default value which is used in place of any IN parameter you miss.

PRAGMA RESTRICT_REFERENCES is supported for syntax compatibility only; the pragma is parsed and ignored.

### 6.1.2 Package Body Syntax

The following is the syntax for the package body:

```
CREATE [ OR REPLACE ] PACKAGE BODY package_name
{ IS | AS }
[ { private_declaration; } ... ]
{ { PROCEDURE proc_name
   { (parm1 [IN | IN OUT | OUT ] datatype1
     [, parm2 [IN | IN OUT | OUT ] datatype2 ] ... ) ]
   IS | AS }
   proc_declaration; } ... ]
BEGIN
   statement; ...
[ EXCEPTION
   WHEN ... THEN
   statement; ...]
END;

| FUNCTION func_name
{ (parm1 [IN | IN OUT | OUT ] datatype1
[, parm2 [IN | IN OUT | OUT ] datatype2 ] ... ) ]
RETURN return_type
{IS | AS }
[ func_declaration; ]...
BEGIN
   statement; ...
[ EXCEPTION
   WHEN ... THEN
   statement; ...]
END; }

| BEGIN
   init_statement; ...
END [ package_name ];
```

package_name is the name of the package for which this is the package body. There must be an existing package specification with the same name.
private_declaration is an identifier of a private variable that can be accessed by any procedure or function within the package. There can be none, one, or more private variables. private_declaration can be any of the following:

- Variable Declaration
- Record Declaration (see Section 4.3.4)
- Collection Declaration (see Section 4.10)
- REF CURSOR and Cursor Variable Declaration
- TYPE Definitions for Records, Collections, and REF CURSORS
- Object Variable Declaration (see Section 8.4)

If proc_name is the same as the identifier of a public procedure declared in the package specification and the signature of proc_name (i.e., formal parameter names (parm1, parm2,...), data types (datatype1, datatype2,...), parameter modes, order of formal parameters, and number of formal parameters) exactly matches the signature of the public procedure’s declaration, then proc_name defines the body of this public procedure.

If the conditions described in the prior paragraph are not true, then proc_name defines a private procedure.

parm1, parm2,... are the formal parameters of the procedure. datatype1, datatype2,... are the data types of parm1, parm2,... respectively. IN, IN OUT, and OUT are the possible parameter modes for each formal parameter. If none are specified, the default is IN. IN parameters can also be initialized with a default value which is used in place of any IN parameter you miss.

proc_variable is an identifier of a variable that can be accessed only from within procedure, proc_name. There can be none, one, or more variables. datatype is the data type of proc_variable. statement is an SPL program statement.

If func_name is the same as the identifier of a public function declared in the package specification and the signature of func_name (i.e., formal parameter names (parm1, parm2,...), data types (datatype1, datatype2,...), parameter modes, order of formal parameters, and number of formal parameters) exactly matches the signature of the public function’s declaration, then func_name defines the body of this public function.

If the conditions described in the prior paragraph are not true, then func_name defines a private function.

parm1, parm2,... are the formal parameters of the function. datatype1, datatype2,... are the data types of parm1, parm2,... respectively. IN, IN OUT, and OUT are the possible parameter modes for each formal parameter. If none are specified, the default is IN. return_type is the data type of the value returned by the function.
\textit{func\_variable} is an identifier of a variable that can be accessed only from within function, \textit{func\_name}. There can be none, one, or more variables. \textit{datatype} is the data type of \textit{func\_variable}. \textit{statement} is an SPL program statement.

\textit{init\_statement} is a statement in the initialization section of the package body. The initialization section, if specified, must contain at least one statement. The statements in the initialization section are executed once per user’s session when the package is first referenced.
6.2 Creating Packages

We will now try to create packages and store them in our database. One thing to remember here is that packages are not executable piece of code. Rather they are a repository of code that is used. When you use a package, you actually execute or make reference to an element in a package. This information is contained in the package specification.

6.2.1 Creating the Package Specification

The package specification contains definition of all the elements in the package that can be referenced from outside it. These are called the public elements of the package and act as the package interface. Following is a package specification.

```sql
CREATE OR REPLACE PACKAGE emp_admin
IS

FUNCTION get_dept_name (p_deptno NUMBER DEFAULT 10)
RETURN VARCHAR2;

FUNCTION update_emp_sal (p_empno NUMBER, p_raise NUMBER)
RETURN NUMBER;

PROCEDURE hire_emp (p_empno NUMBER,
  p_ename VARCHAR2,
  p_job VARCHAR2,
  p_sal NUMBER,
  p_hiredate DATE DEFAULT sysdate,
  p_comm NUMBER DEFAULT 0,
  p_mgr NUMBER,
  p_deptno NUMBER DEFAULT 10);

PROCEDURE fire_emp (p_empno NUMBER);

END emp_admin;
```

Here we have created the `emp_admin` package specification. This package specification consists of two functions and two stored procedures. We can also add the `OR REPLACE` clause to the `CREATE PACKAGE` statement for convenience.

6.2.2 Creating the Package Body

The body of the package contains the actual implementation behind the package specification. For the above `emp_admin` package specification, we shall now create a
package body which will implement the specifications. The body will contain the
implementation of the functions and stored procedures in the specification.

-- Package body for the 'emp_admin' package.

CREATE OR REPLACE PACKAGE BODY emp_admin
IS
  -- Function that queries the 'dept' table based on the department
  -- number and returns the corresponding department name.
  FUNCTION get_dept_name (p_deptno IN NUMBER DEFAULT 10)
  RETURN VARCHAR2
  IS
    v_dname VARCHAR2(14);
    BEGIN
      SELECT dname INTO v_dname FROM dept WHERE deptno = p_deptno;
      RETURN v_dname;
    EXCEPTION
      WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.PUT_LINE('Invalid department number ' || p_deptno);
        RETURN '';
    END;
  -- Function that updates an employee's salary based on the
  -- employee number and salary increment/decrement passed
  -- as IN parameters. Upon successful completion the function
  -- returns the new updated salary.
  FUNCTION update_emp_sal (p_empno IN NUMBER,
                           p_raise IN NUMBER)
  RETURN NUMBER
  IS
    v_sal NUMBER := 0;
    BEGIN
      SELECT sal INTO v_sal FROM emp WHERE empno = p_empno;
      v_sal := v_sal + p_raise;
      UPDATE emp SET sal = v_sal WHERE empno = p_empno;
      RETURN v_sal;
    EXCEPTION
      WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.PUT_LINE('Employee ' || p_empno || ' not found');
        RETURN -1;
      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);
        RETURN -1;
    END;
  -- Procedure that inserts a new employee record into the 'emp' table.
  PROCEDURE hire_emp (p_empno NUMBER,
                      p_name VARCHAR2,
                      p_job VARCHAR2,
INSERT INTO emp(empno, ename, job, sal, hiredate, comm, mgr, deptno)
VALUES(p_empno, p_ename, p_job, p_sal, p_hiredate, p_comm, p_mgr, p_deptno);
END;

-- Procedure that deletes an employee record from the 'emp' table based
-- on the employee number.

PROCEDURE fire_emp (p_empno NUMBER)
AS
BEGIN
DELETE FROM emp WHERE empno = p_empno;
END;
END;

6.3 Referencing a Package

To reference the types, items and subprograms that are declared within a package
specification, we use the dot notation. For example:

package_name.type_name
package_name.item_name
package_name.subprogram_name

To invoke a function from the emp_admin package specification, we will execute the
following SQL command.

SELECT emp_admin.get_dept_name(10) FROM DUAL;

Here we are invoking the get_dept_name function declared within the package
emp_admin. We are passing the department number as an argument to the function,
which will return the name of the department. Here the value returned should be
ACCOUNTING, which corresponds to department number 10.
6.4 Using Packages With User Defined Types

The following example incorporates the various user-defined types discussed in earlier chapters within the context of a package.

The package specification of `emp_rpt` shows the declaration of a record type, `emprec_typ`, and a weakly-typed `REF CURSOR`, `emp_refcur`, as publicly accessible along with two functions and two procedures. Function, `open_emp_by_dept`, returns the `REF CURSOR` type, `EMP_REFCUR`. Procedures, `fetch_emp` and `close_refcur`, both declare a weakly-typed `REF CURSOR` as a formal parameter.

```sql
CREATE OR REPLACE PACKAGE emp_rpt
IS
    TYPE emprec_typ IS RECORD (
        empno       NUMBER(4),
        ename       VARCHAR(10)
    );
    TYPE emp_refcur IS REF CURSOR;
    FUNCTION get_dept_name (p_deptno IN NUMBER) RETURN VARCHAR2;
    FUNCTION open_emp_by_dept (p_deptno IN emp.deptno%TYPE) RETURN EMP_REFCUR;
    PROCEDURE fetch_emp (p_refcur IN OUT SYS_REFCURSOR);
    PROCEDURE close_refcur (p_refcur IN OUT SYS_REFCURSOR);
END emp_rpt;
```

The package body shows the declaration of several private variables - a static cursor, `dept_cur`, a table type, `depttab_typ`, a table variable, `t_dept`, an integer variable, `t_dept_max`, and a record variable, `r_emp`.

```sql
CREATE OR REPLACE PACKAGE BODY emp_rpt
IS
    CURSOR dept_cur IS SELECT * FROM dept;
    TYPE depttab_typ IS TABLE OF dept%ROWTYPE INDEX BY BINARY_INTEGER;
    t_dept          DEPTTAB_TYP;
    t_dept_max      INTEGER := 1;
    r_emp           EMPREC_TYP;
    FUNCTION get_dept_name (p_deptno IN NUMBER) RETURN VARCHAR2 IS
        FOR i IN 1..t_dept_max LOOP
            IF p_deptno = t_dept(i).deptno THEN
                RETURN t_dept(i).dname;
            END IF;
        END LOOP;
    END get_dept_name;
END emp_rpt;
```
This package contains an initialization section that loads the private table variable, t_dept, using the private static cursor, dept_cur. t_dept serves as a department name lookup table in function, get_dept_name.

Function, open_emp_by_dept returns a REF CURSOR variable for a result set of employee numbers and names for a given department. This REF CURSOR variable can then be passed to procedure, fetch_emp, to retrieve and list the individual rows of the result set. Finally, procedure, close_refcur, can be used to close the REF CURSOR variable associated with this result set.

The following anonymous block runs the package function and procedures. In the anonymous block's declaration section, note the declaration of cursor variable, v_emp_cur, using the package’s public REF CURSOR type, EMP_REFCUR. v_emp_cur
contains the pointer to the result set that is passed between the package function and procedures.

```
DECLARE
    v_deptno        dept.deptno%TYPE DEFAULT 30;
    v_emp_cur       emp_rpt.EMP_REFCUR;
BEGIN
    v_emp_cur := emp_rpt.open_emp_by_dept(v_deptno);
    DBMS_OUTPUT.PUT_LINE('EMPLOYEES IN DEPT #' || v_deptno || ':
        ' || emp_rpt.get_dept_name(v_deptno));
    emp_rpt.fetch_emp(v_emp_cur);
    DBMS_OUTPUT.PUT_LINE('**********************
        ' || v_emp_cur%ROWCOUNT || ' rows were retrieved');
    emp_rpt.close_refcur(v_emp_cur);
END;
```

The following is the result of this anonymous block.

```
EMPLOYEES IN DEPT #30: SALES
EMPNO    ENAME
-----    ------
7499     ALLEN
7521     WARD
7654     MARTIN
7698     BLAKE
7844     TURNER
7900     JAMES
**********************
6 rows were retrieved
```

The following anonymous block illustrates another means of achieving the same result. Instead of using the package procedures, `fetch_emp` and `close_refcur`, the logic of these programs is coded directly into the anonymous block. In the anonymous block’s declaration section, note the addition of record variable, `r_emp`, declared using the package’s public record type, `EMPREC_TYP`.

```
DECLARE
    v_deptno        dept.deptno%TYPE DEFAULT 30;
    v_emp_cur       emp_rpt.EMP_REFCUR;
    r_emp           emp_rpt.EMPREC_TYP;
BEGIN
    v_emp_cur := emp_rpt.open_emp_by_dept(v_deptno);
    DBMS_OUTPUT.PUT_LINE('EMPLOYEES IN DEPT #' || v_deptno || ':
        ' || emp_rpt.get_dept_name(v_deptno));
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
    DBMS_OUTPUT.PUT_LINE('-----    ------');
    LOOP
        FETCH v_emp_cur INTO r_emp;
        EXIT WHEN v_emp_cur%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(r_emp.empno || '     ' ||
                             r_emp.ename);
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('**********************
        ' || v_emp_cur%ROWCOUNT || ' rows were retrieved');
    CLOSE v_emp_cur;
END;
```
The following is the result of this anonymous block.

<table>
<thead>
<tr>
<th>EMPNO</th>
<th>ENAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>7499</td>
<td>ALLEN</td>
</tr>
<tr>
<td>7521</td>
<td>WARD</td>
</tr>
<tr>
<td>7654</td>
<td>MARTIN</td>
</tr>
<tr>
<td>7698</td>
<td>BLAKE</td>
</tr>
<tr>
<td>7844</td>
<td>TURNER</td>
</tr>
<tr>
<td>7900</td>
<td>JAMES</td>
</tr>
</tbody>
</table>

6 rows were retrieved

6.5 Dropping a Package

The syntax for deleting an entire package or just the package body is as follows:

```
DROP PACKAGE [ BODY ] package_name;
```

If the keyword, BODY, is omitted, both the package specification and the package body are deleted - i.e., the entire package is dropped. If the keyword, BODY, is specified, then only the package body is dropped. The package specification remains intact. `package_name` is the identifier of the package to be dropped.

Following statement will destroy only the package body of `emp_admin`:

```
DROP PACKAGE BODY emp_admin;
```

The following statement will drop the entire `emp_admin` package:

```
DROP PACKAGE emp_admin;
```
7 Built-In Packages

This chapter describes the built-in packages that are provided with Postgres Plus 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;

7.1 DBMS_ALERT

The DBMS_ALERT package provides the capability to register for, send, and receive alerts.

The procedures and functions available in the DBMS_ALERT package are listed in the following table.

Table 7-7-1 DBMS_ALERT Functions/Procedures

<table>
<thead>
<tr>
<th>Function/Procedure</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>REGISTER(name)</td>
<td>n/a</td>
<td>Register to be able to receive alerts named, name.</td>
</tr>
<tr>
<td>REMOVE(name)</td>
<td>n/a</td>
<td>Remove registration for the alert named, name.</td>
</tr>
<tr>
<td>REMOVEALL</td>
<td>n/a</td>
<td>Remove registration for all alerts.</td>
</tr>
<tr>
<td>SIGNAL(name, message)</td>
<td>n/a</td>
<td>Signals the alert named, name, with message.</td>
</tr>
<tr>
<td>WAITANY(name OUT, message OUT, status OUT, timeout)</td>
<td>n/a</td>
<td>Wait for any registered alert to occur.</td>
</tr>
<tr>
<td>WAITONE(name, message OUT, status OUT, timeout)</td>
<td>n/a</td>
<td>Wait for the specified alert, name, to occur.</td>
</tr>
</tbody>
</table>

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/9.2AS/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:

```plaintext
custom_variable_classes = 'dbms_alert, ...'
```

After editing the postgresql.conf file parameters, you must restart the server for the changes to take effect.

### 7.1.1 REGISTER

The REGISTER procedure enables the current session to be notified of the specified alert.

```plaintext
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.

```sql
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...
7.1.2 REMOVE

The REMOVE procedure unregisters the session for the named alert.

REMOVE(name VARCHAR2)

Parameters

name

Name of the alert to be unregistered.

7.1.3 REMOVEALL

The REMOVEALL procedure unregisters the session for all alerts.

REMOVEALL

7.1.4 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
```
7.1.5 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`:

```sql
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:

```sql
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
```

### 7.1.6 WAITONE

The `WAITONE` procedure waits for the specified registered alert to occur.

```sql
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`.

```sql
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:

```sql
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:

```sql
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
```

7.1.7 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 TRIGGER dept_alert_trig
   AFTER INSERT OR UPDATE OR DELETE ON dept
DECLARE
   v_action VARCHAR2(25);
BEGIN
   IF INSERTING THEN
      v_action := ' added department(s) ';
   ELSIF UPDATING THEN
      v_action := ' updated department(s) ';
   ELSIF DELETING THEN
      v_action := ' deleted department(s) ';
   END IF;
   DBMS_ALERT.SIGNAL('dept_alert',USER || v_action || 'on ' || SYSDATE);
END;

CREATE OR REPLACE TRIGGER emp_alert_trig
   AFTER INSERT OR UPDATE OR DELETE ON emp
DECLARE
   v_action VARCHAR2(25);
BEGIN
   IF INSERTING THEN
      v_action := ' added employee(s) ';
   ELSIF UPDATING THEN
      v_action := ' updated employee(s) ';
   ELSIF DELETING THEN
      v_action := ' deleted employee(s) ';
   END IF;
   DBMS_ALERT.SIGNAL('emp_alert',USER || v_action || 'on ' || SYSDATE);
END;

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...

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);
```

The following change is made by user, john:

```
INSERT INTO dept VALUES (60, 'HR', 'LOS ANGELES');
```

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 25-OCT-07 16:41:01
Alert status : 0

Alert name : emp_alert
Alert msg  : mary added employee(s) on 25-OCT-07 16:41:02
Alert status : 0

Alert name : dept_alert
Alert msg  : john added department(s) on 25-OCT-07 16:41:22
Alert status : 0

Alert status : 1
```
7.2 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 Postgres Plus Advanced Server installer installs pgAgent, but you must start the pgAgent service manually prior to using DBMS_JOB. If you attempt to use this package to schedule a job after uninstalling pgAgent, DBMS_JOB will throw an error. DBMS_JOB verifies that pgAgent is installed, but does not verify that the service is running.

Table 7-2 DBMS_JOB Functions/Procedures

<table>
<thead>
<tr>
<th>Function/Procedure</th>
<th>Function or Procedure</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BROKEN(job, broken [, next_date ])</td>
<td>Procedure</td>
<td>n/a</td>
<td>Specify that a given job is either broken or not broken.</td>
</tr>
<tr>
<td>CHANGE(job, what, next_date, interval, instance, force)</td>
<td>Procedure</td>
<td>n/a</td>
<td>Change the job’s parameters.</td>
</tr>
<tr>
<td>INTERVAL(job, interval)</td>
<td>Procedure</td>
<td>n/a</td>
<td>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.</td>
</tr>
<tr>
<td>NEXT_DATE(job, next_date)</td>
<td>Procedure</td>
<td>n/a</td>
<td>Set the next date/time the job is to be run.</td>
</tr>
<tr>
<td>REMOVE(job)</td>
<td>Procedure</td>
<td>n/a</td>
<td>Delete the job definition from the database.</td>
</tr>
<tr>
<td>RUN(job)</td>
<td>Procedure</td>
<td>n/a</td>
<td>Forces execution of a job even if it is marked broken.</td>
</tr>
<tr>
<td>SUBMIT(job OUT, what [, next_date [, interval [, no_parse ]]]))</td>
<td>Procedure</td>
<td>n/a</td>
<td>Creates a job and stores its definition in the database.</td>
</tr>
<tr>
<td>WHAT(job, what)</td>
<td>Procedure</td>
<td>n/a</td>
<td>Change the stored procedure run by a job.</td>
</tr>
</tbody>
</table>

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;

7.2.1 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;
7.2.2 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.

\[
\text{CHANGE}(\text{job INTEGER, what VARCHAR2, next_date DATE, interval VARCHAR2, instance INTEGER, force BOOLEAN})
\]

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;
```
7.2.3 INTERVAL

The INTERVAL procedure sets the frequency of how often a job is to be run.

\[
\text{INTERVAL}(\text{job BINARY\_INTEGER, interval VARCHAR2})
\]

**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:

```sql
BEGIN
  DBMS_JOB.INTERVAL(104,'SYSDATE + 7');
END;
```

7.2.4 NEXT\_DATE

The NEXT\_DATE procedure sets the date/time of when the job is to be run next.

\[
\text{NEXT\_DATE}(\text{job BINARY\_INTEGER, next\_date DATE})
\]

**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:

```sql
BEGIN
  DBMS_JOB.NEXT\_DATE(104, TO\_DATE('14-DEC-07','DD-MON-YY'));
END;
```
7.2.5 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;
```

7.2.6 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;
```
7.2.7 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');
The job immediately executes procedure, job_proc, populating table, jobrun, with a row:

```sql
SELECT * FROM jobrun;
```

7.2.8 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:

```sql
BEGIN
    DBMS_JOB.WHAT(104,'list_emp;');
END;
```
## 7.3 DBMS_LOB

The DBMS_LOB package provides the capability to operate on large objects.

### Table 7-3 DBMS_LOB Functions/Procedures

<table>
<thead>
<tr>
<th>Function/Procedure</th>
<th>Function or Procedure</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>APPEND(dest_lob IN OUT, src_lob)</td>
<td>Procedure</td>
<td>n/a</td>
<td>Appends one large object to another.</td>
</tr>
<tr>
<td>COMPARE(lob_1, lob_2 [, amount [, offset_1 [, offset_2 ]]])</td>
<td>Function</td>
<td>INTEGER</td>
<td>Compares two large objects.</td>
</tr>
<tr>
<td>CONVERTLOB(dest_lob IN OUT, src_clob, amount, dest_offset IN OUT, src_offset IN OUT, blob_csid, lang_context IN OUT, warning OUT)</td>
<td>Procedure</td>
<td>n/a</td>
<td>Converts character data to binary.</td>
</tr>
<tr>
<td>CONVERTTOCLOB(dest_lob IN OUT, src_blob, amount, dest_offset IN OUT, src_offset IN OUT, blob_csid, lang_context IN OUT, warning OUT)</td>
<td>Procedure</td>
<td>n/a</td>
<td>Converts binary data to character.</td>
</tr>
<tr>
<td>COPY(dest_lob IN OUT, src_lob, amount [, dest_offset [, src_offset ]])</td>
<td>Procedure</td>
<td>n/a</td>
<td>Copies one large object to another.</td>
</tr>
<tr>
<td>ERASE(lob_loc IN OUT, amount IN OUT [, offset ])</td>
<td>Procedure</td>
<td>n/a</td>
<td>Erase a large object.</td>
</tr>
<tr>
<td>GET_STORAGE_LIMIT(lob_loc)</td>
<td>Function</td>
<td>INTEGER</td>
<td>Get the storage limit for large objects.</td>
</tr>
<tr>
<td>GETLENGTH(lob_loc)</td>
<td>Function</td>
<td>INTEGER</td>
<td>Get the length of the large object.</td>
</tr>
<tr>
<td>INSTR(lob_loc, pattern [, offset [, nth ]])</td>
<td>Function</td>
<td>INTEGER</td>
<td>Get the position of the nth occurrence of a pattern in the large object starting at offset.</td>
</tr>
<tr>
<td>READ(lob_loc, amount IN OUT, offset, buffer OUT)</td>
<td>Procedure</td>
<td>n/a</td>
<td>Read a large object.</td>
</tr>
<tr>
<td>SUBSTR(lob_loc [, amount [, offset ]])</td>
<td>Function</td>
<td>RAW, VARCHAR2</td>
<td>Get part of a large object.</td>
</tr>
<tr>
<td>TRIM(lob_loc IN OUT, newlen)</td>
<td>Procedure</td>
<td>n/a</td>
<td>Trim a large object to the specified length.</td>
</tr>
<tr>
<td>WRITE(lob_loc IN OUT, amount, offset, buffer)</td>
<td>Procedure</td>
<td>n/a</td>
<td>Write data to a large object.</td>
</tr>
<tr>
<td>WRITEAPPEND(lob_loc IN OUT, amount, buffer)</td>
<td>Procedure</td>
<td>n/a</td>
<td>Write data from the buffer to the end of a large object.</td>
</tr>
</tbody>
</table>
The following table lists the public variables available in the package.

**Table 7-4 DBMS_LOB Public Variables**

<table>
<thead>
<tr>
<th>Public Variables</th>
<th>Data Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>compress off</td>
<td>INTEGER</td>
<td>0</td>
</tr>
<tr>
<td>compress on</td>
<td>INTEGER</td>
<td>1</td>
</tr>
<tr>
<td>deduplicate_off</td>
<td>INTEGER</td>
<td>0</td>
</tr>
<tr>
<td>deduplicate_on</td>
<td>INTEGER</td>
<td>4</td>
</tr>
<tr>
<td>default_csid</td>
<td>INTEGER</td>
<td>0</td>
</tr>
<tr>
<td>default_lang_ctx</td>
<td>INTEGER</td>
<td>0</td>
</tr>
<tr>
<td>encrypt_off</td>
<td>INTEGER</td>
<td>0</td>
</tr>
<tr>
<td>encrypt_on</td>
<td>INTEGER</td>
<td>1</td>
</tr>
<tr>
<td>file_readonly</td>
<td>INTEGER</td>
<td>0</td>
</tr>
<tr>
<td>lobmaxsize</td>
<td>INTEGER</td>
<td>1073741823</td>
</tr>
<tr>
<td>lob_readonly</td>
<td>INTEGER</td>
<td>0</td>
</tr>
<tr>
<td>lob_readwrite</td>
<td>INTEGER</td>
<td>1</td>
</tr>
<tr>
<td>no_warning</td>
<td>INTEGER</td>
<td>0</td>
</tr>
<tr>
<td>opt_compress</td>
<td>INTEGER</td>
<td>1</td>
</tr>
<tr>
<td>opt_deduplicate</td>
<td>INTEGER</td>
<td>4</td>
</tr>
<tr>
<td>opt_encrypt</td>
<td>INTEGER</td>
<td>2</td>
</tr>
<tr>
<td>warn_inconvertible_char</td>
<td>INTEGER</td>
<td>1</td>
</tr>
</tbody>
</table>

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.

### 7.3.1 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*.
7.3.2 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.

\[
\text{status INTEGER COMPARE}(\text{lob}_1 \ {\text{BLOB | CLOB }}, \ \\
\text{lob}_2 \ {\text{BLOB | CLOB }}, \ \\
[, \ \text{amount INTEGER [}, \ \text{offset}_1 \ \text{INTEGER [}, \ \text{offset}_2 \ \text{INTEGER ]}]])
\]

Parameters

\(\text{lob}_1\)

Large object locator of the first large object to be compared. Must be the same data type as \(\text{lob}_2\).

\(\text{lob}_2\)

Large object locator of the second large object to be compared. Must be the same data type as \(\text{lob}_1\).

\(\text{amount}\)

If the data type of the large objects is BLOB, then the comparison is made for \(\text{amount}\) bytes. If the data type of the large objects is CLOB, then the comparison is made for \(\text{amount}\) characters. The default it the maximum size of a large object.

\(\text{offset}_1\)

Position within the first large object to begin the comparison. The first byte/character is offset 1. The default is 1.

\(\text{offset}_2\)

Position within the second large object to begin the comparison. The first byte/character is offset 1. The default is 1.

\(\text{status}\)

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 \(\text{amount}, \ \text{offset}_1, \text{ or } \text{offset}_2\) are less than zero.
7.3.3 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.

7.3.4 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.

7.3.5 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.

### 7.3.6 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.

7.3.7 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.

7.3.8 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.
Length of the large object in bytes for BLOBs or characters for CLOBs.

7.3.9 INSTR

The INSTR function returns the location of the nth occurrence of a given pattern within a large object.

\[
\text{position INTEGER INSTR(lob_loc \{ BLOB \or CLOB \},}
\text{ pattern \{ RAW \or VARCHAR2 \}[, offset INTEGER [, nth INTEGER ]])}
\]

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.
7.3.10  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.

7.3.11  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.

### 7.3.12 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.

### 7.3.13 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.

7.3.14 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.
7.4 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 7-7-5 DBMS_OUTPUT Functions/Procedures

<table>
<thead>
<tr>
<th>Function/Procedure</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DISABLE</td>
<td>n/a</td>
<td>Disable the capability to send and receive messages.</td>
</tr>
<tr>
<td>ENABLE(buffer_size)</td>
<td>n/a</td>
<td>Enable the capability to send and receive messages.</td>
</tr>
<tr>
<td>GET_LINE(line OUT, status OUT)</td>
<td>n/a</td>
<td>Get a line from the message buffer.</td>
</tr>
<tr>
<td>GET_LINES(lines OUT, numlines IN OUT)</td>
<td>n/a</td>
<td>Get multiple lines from the message buffer.</td>
</tr>
<tr>
<td>NEW_LINE</td>
<td>n/a</td>
<td>Puts an end-of-line character sequence.</td>
</tr>
<tr>
<td>PUT(item)</td>
<td>n/a</td>
<td>Puts a partial line without an end-of-line character sequence.</td>
</tr>
<tr>
<td>PUT_LINE(item)</td>
<td>n/a</td>
<td>Puts a complete line with an end-of-line character sequence.</td>
</tr>
<tr>
<td>SERVEROUTPUT(stdout)</td>
<td>n/a</td>
<td>Direct messages from PUT, PUT_LINE, or NEW_LINE to either standard output or the message buffer.</td>
</tr>
</tbody>
</table>

The following table lists the public variables available in the DBMS_OUTPUT package.

Table 7-7-6 DBMS_OUTPUT Public Variables

<table>
<thead>
<tr>
<th>Public Variables</th>
<th>Data Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>chararr</td>
<td>TABLE</td>
<td>For message lines.</td>
<td></td>
</tr>
</tbody>
</table>

7.4.1 CHARARR

The CHARARR is for storing multiple message lines.

```
TYPE chararr IS TABLE OF VARCHAR2(32767) INDEX BY BINARY_INTEGER;
```

7.4.2 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;
```

#### 7.4.3 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.

```sql
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)`.

```sql
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.

```sql
BEGIN
    DBMS_OUTPUT.ENABLE;
    DBMS_OUTPUT.SERVEROUTPUT(FALSE);
    DBMS_OUTPUT.PUT_LINE('Message sent to buffer');
END;
```

7.4.4 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.

```sql
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.

```sql
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 || ', ' || 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.

```sql
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;
```

<table>
<thead>
<tr>
<th>msg</th>
</tr>
</thead>
<tbody>
<tr>
<td>7369,SMITH,CLERK,7902,17-DEC-80 00:00:00,800.00,,20</td>
</tr>
<tr>
<td>7499,ALLEN,SALESMAN,7698,20-FEB-81 00:00:00,1600.00,300.00,30</td>
</tr>
<tr>
<td>7521,WARD,SALESMAN,7698,22-FEB-81 00:00:00,1250.00,500.00,30</td>
</tr>
<tr>
<td>7566,JONES,MANAGER,7839,02-APR-81 00:00:00,2975.00,,20</td>
</tr>
<tr>
<td>7654,MARTIN,SALESMAN,7698,28-SEP-81 00:00:00,1250.00,1400.00,30</td>
</tr>
<tr>
<td>7698,BLAKE,MANAGER,7839,01-MAY-81 00:00:00,2850.00,,30</td>
</tr>
<tr>
<td>7782,CLARK,MANAGER,7839,09-JUN-81 00:00:00,2450.00,,10</td>
</tr>
<tr>
<td>7788,SCOTT,ANALYST,7566,19-APR-87 00:00:00,3000.00,,20</td>
</tr>
<tr>
<td>7839,KING,PRESIDENT,,17-NOV-81 00:00:00,5000.00,,10</td>
</tr>
<tr>
<td>7844,TURNER,SALESMAN,7698,08-SEP-81 00:00:00,1500.00,,0,0,0,30</td>
</tr>
<tr>
<td>7876,ADAMS,CLERK,7788,23-MAY-87 00:00:00,1100.00,,20</td>
</tr>
<tr>
<td>7900,JAMES,CLERK,7698,03-DEC-81 00:00:00,950.00,,30</td>
</tr>
<tr>
<td>7902,FOOR,ANALYST,7566,03-DEC-81 00:00:00,3000.00,,20</td>
</tr>
<tr>
<td>7934,Miller,CLERK,7782,23-JAN-82 00:00:00,1300.00,,10</td>
</tr>
</tbody>
</table>
(14 rows)
7.4.5 **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 examples uses the **GET_LINES** procedure to store all rows from the **emp** table that were placed on the message buffer, into an array.

```sql
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,'9999.99')),'') || ',' || 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
```

---

Copyright © 2007 - 2015 EnterpriseDB Corporation. All rights reserved.
7.4.6 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.

7.4.7 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.

```sql
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.PUT_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
7690,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

7.4.8 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.

DECL
  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

7.4.9 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.

The default setting of SERVEROUTPUT is implementation dependent. For example, in Oracle SQL*Plus, SERVEROUTPUT(FALSE) is the default. In PSQL, SERVEROUTPUT(TRUE) is the default. Also note that in Oracle SQL*Plus, this setting is controlled using the SQL*Plus SET command, not by a stored procedure as implemented in Postgres Plus Advanced Server.

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
## 7.5 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>
<thead>
<tr>
<th>Function/Procedure</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CREATE_PIPE(pipename [, maxpipesize ] [, private ] )</td>
<td>INTEGER</td>
<td>Explicitly create a private pipe if private is “true” (the default) or a public pipe if private is “false”.</td>
</tr>
<tr>
<td>NEXT_ITEM_TYPE</td>
<td>INTEGER</td>
<td>Determine the data type of the next item in a received message.</td>
</tr>
<tr>
<td>PACK_MESSAGE(item )</td>
<td>n/a</td>
<td>Place item in the session’s local message buffer.</td>
</tr>
<tr>
<td>PURGE(pipename )</td>
<td>n/a</td>
<td>Remove unreceived messages from the specified pipe.</td>
</tr>
<tr>
<td>RECEIVE_MESSAGE(pipename [, timeout ] )</td>
<td>INTEGER</td>
<td>Get a message from a specified pipe.</td>
</tr>
<tr>
<td>REMOVE_PIPE(pipename )</td>
<td>INTEGER</td>
<td>Delete an explicitly created pipe.</td>
</tr>
<tr>
<td>RESET_BUFFER</td>
<td>n/a</td>
<td>Reset the local message buffer.</td>
</tr>
<tr>
<td>SEND_MESSAGE(pipename [, timeout ] [, maxpipesize ] )</td>
<td>INTEGER</td>
<td>Send a message on a pipe.</td>
</tr>
<tr>
<td>UNIQUE_SESSION_NAME</td>
<td>VARCHAR2</td>
<td>Obtain a unique session name.</td>
</tr>
<tr>
<td>UNPACK_MESSAGE(item OUT)</td>
<td>n/a</td>
<td>Retrieve the next data item from a message into a type-compatible variable, item.</td>
</tr>
</tbody>
</table>

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.

### 7.5.1 CREATE_PIPE

The CREATE_PIPE function creates an explicit public pipe or an explicit private pipe with a specified name.

```sql
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

Create a private pipe named, messages:

```sql
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
```

Create a public pipe named, mailbox:

```sql
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
```

7.5.2 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.

```sql
typecode INTEGER NEXT_ITEM_TYPE
```

Parameters

typecode

Code identifying the data type of the next data item as shown in Table 7-7-8.

Table 7-7-8 NEXT_ITEM_TYPE Data Type Codes

<table>
<thead>
<tr>
<th>Type Code</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>No more data items</td>
</tr>
<tr>
<td>9</td>
<td>NUMBER</td>
</tr>
<tr>
<td>11</td>
<td>VARCHAR2</td>
</tr>
<tr>
<td>13</td>
<td>DATE</td>
</tr>
<tr>
<td>23</td>
<td>RAW</td>
</tr>
</tbody>
</table>

Note: The type codes list in the table are not Oracle compatible. Oracle assigns a different numbering sequence to the data types.
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.

```sql
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);
```

7.5.3 **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.

7.5.4 **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:

```sql
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:

```sql
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:

```sql
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.

```sql
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;
```

Copyright © 2007 - 2015 EnterpriseDB Corporation. All rights reserved.
7.5.5 RECEIVE_MESSAGE

The RECEIVE_MESSAGE function obtains a message from a specified pipe.

```sql
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>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success</td>
</tr>
<tr>
<td>1</td>
<td>Time out</td>
</tr>
<tr>
<td>2</td>
<td>Message too large for the buffer</td>
</tr>
</tbody>
</table>

7.5.6 REMOVE_PIPE

The REMOVE_PIPE function deletes an explicit private or explicit public pipe.

```sql
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:

```sql
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:

```sql
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:

```sql
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

7.5.7 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.

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?

7.5.8 SEND_MESSAGE

The SEND_MESSAGE function sends a message from the session’s local message buffer to the specified pipe.

```sql
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>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success</td>
</tr>
<tr>
<td>1</td>
<td>Time out</td>
</tr>
<tr>
<td>3</td>
<td>Function interrupted</td>
</tr>
</tbody>
</table>

7.5.9 UNIQUE_SESSION_NAME

The UNIQUE_SESSION_NAME function returns a name, unique to the current session.

```sql
name VARCHAR2 UNIQUE_SESSION_NAME
```

**Parameters**

`name`
Unique session name.

Examples

The following anonymous block retrieves and displays a unique session name.

```sql
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

7.5.10 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.

7.5.11 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.

```sql
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;
            v_item := DBMS_PIPE.RECEIVE_MESSAGE(p_mailbox);
            DBMS_OUTPUT.PUT_LINE('****** Start message #' || v_msgno || ' ******');
        END LOOP;
    END empty_mailbox;
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 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
7.6 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 the Postgres Plus Advanced Server Performance Features Guide.

Table 7-11 DBMS_PROFILER Functions/Procedures

<table>
<thead>
<tr>
<th>Function/Procedure</th>
<th>Function or Procedure</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLUSH_DATA</td>
<td>Both</td>
<td>Status Code or Exception</td>
<td>Flushes performance data collected in the current session without terminating the session (profiling continues).</td>
</tr>
<tr>
<td>GET_VERSION(major OUT, minor OUT)</td>
<td>Procedure</td>
<td>n/a</td>
<td>Returns the version number of this package.</td>
</tr>
<tr>
<td>INTERNAL_VERSION_CHECK</td>
<td>Function</td>
<td>Status Code</td>
<td>Confirms that the current version of the profiler will work with the current database.</td>
</tr>
<tr>
<td>PAUSE_PROFILER</td>
<td>Both</td>
<td>Status Code or Exception</td>
<td>Pause data collection.</td>
</tr>
<tr>
<td>RESUME_PROFILER</td>
<td>Both</td>
<td>Status Code or Exception</td>
<td>Resume data collection.</td>
</tr>
<tr>
<td>START_PROFILER(run_comment, run_comment1 [, run_number OUT])</td>
<td>Both</td>
<td>Status Code or Exception</td>
<td>Start data collection.</td>
</tr>
<tr>
<td>STOP_PROFILER</td>
<td>Both</td>
<td>Status Code or Exception</td>
<td>Stop data collection and flush performance data to the PLSQL_PROFILER_RAWDATA table.</td>
</tr>
</tbody>
</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 7-12 DBMS_PROFILER Status Codes and Exceptions

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Message</th>
<th>Exception</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>error version</td>
<td>version_mismatch</td>
<td>The profiler version and the database are incompatible.</td>
</tr>
<tr>
<td>0</td>
<td>success</td>
<td>n/a</td>
<td>The operation completed successfully.</td>
</tr>
<tr>
<td>1</td>
<td>error_param</td>
<td>profiler_error</td>
<td>The operation received an incorrect parameter.</td>
</tr>
<tr>
<td>2</td>
<td>error_io</td>
<td>profiler_error</td>
<td>The data flush operation has failed.</td>
</tr>
</tbody>
</table>
7.6.1 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 Postgres Plus Advanced Server Performance Features Guide. The function and procedure signatures are:

```
status INTEGER FLUSH_DATA
```

**Parameters**

`status`

Status code returned by the operation.

7.6.2 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.

7.6.3 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

\textit{status}

Status code returned by the operation.

\subsection*{7.6.4 \texttt{PAUSE\_PROFILER}}

The \texttt{PAUSE\_PROFILER} function/procedure pauses a profiling session. The function and procedure signatures are:

\begin{verbatim}
status INTEGER PAUSE\_PROFILER
\end{verbatim}

Parameters

\textit{status}

Status code returned by the operation.

\subsection*{7.6.5 \texttt{RESUME\_PROFILER}}

The \texttt{RESUME\_PROFILER} function/procedure pauses a profiling session. The function and procedure signatures are:

\begin{verbatim}
status INTEGER RESUME\_PROFILER
\end{verbatim}

Parameters

\textit{status}

Status code returned by the operation.
7.6.6 START_PROFILER

The START_PROFILER function/procedure starts a data collection session. The function and procedure signatures are:

```sql
status INTEGER START_PROFILER(run_comment TEXT := SYSDATE,
       run_comment1 TEXT := '' [, run_number OUT INTEGER ])
```

```sql
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.

---

7.6.7 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:

```sql
status INTEGER STOP_PROFILER
```

**Parameters**

*status*

Status code returned by the operation.
7.7 DBMS_RLS

The DBMS_RLS package enables the implementation of Virtual Private Database on certain Advanced Server database objects.

**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 Oracle compatible 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. See the `GRANT` command in Section 3.3.48 or the `ALTER ROLE` command in Section 3.3.3 for additional information.
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 Oracle compatible catalog views **ALL_POLICIES** (see Section 10.8), **DBA_POLICIES** (see Section 10.27), or **USER_POLICIES** (see Section 10.46).

**Table 7-13 DBMS_RLS Functions/Procedures**

<table>
<thead>
<tr>
<th>Function/Procedure</th>
<th>Function or Procedure</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>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 ]]]]]]]])</td>
<td>Procedure</td>
<td>n/a</td>
<td>Add a security policy to a database object.</td>
</tr>
<tr>
<td>DROP_POLICY(object_schema, object_name, policy_name)</td>
<td>Procedure</td>
<td>n/a</td>
<td>Remove a security policy from a database object.</td>
</tr>
<tr>
<td>ENABLE_POLICY(object_schema, object_name, policy_name, enable)</td>
<td>Procedure</td>
<td>n/a</td>
<td>Enable or disable a security policy.</td>
</tr>
</tbody>
</table>
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:

<table>
<thead>
<tr>
<th>attribute Value</th>
<th>Equivalent Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SESSION USER</td>
<td><code>pg_catalog.session_user</code></td>
</tr>
<tr>
<td>CURRENT USER</td>
<td><code>pg_catalog.current_user</code></td>
</tr>
<tr>
<td>CURRENT_SCHEMA</td>
<td><code>pg_catalog.current_schema</code></td>
</tr>
<tr>
<td>HOST</td>
<td><code>pg_catalog.inet_host</code></td>
</tr>
<tr>
<td>IP ADDRESS</td>
<td><code>pg_catalog.inet_client_addr</code></td>
</tr>
<tr>
<td>SERVER_HOST</td>
<td><code>pg_catalog.inet_server_addr</code></td>
</tr>
</tbody>
</table>

Note: The examples used to illustrate the `DBMS_RLS` package are based on a modified copy of the sample `emp` table provided with Postgres Plus 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:

```sql
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;
7.7.1 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.
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

In Oracle, 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: In Oracle 10g, the policy_type parameter was introduced, which is intended to replace the static_policy parameter. In Oracle, if the policy_type parameter is not set to its default value of NULL, the policy_type parameter setting overrides the static_policy setting.

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

In Oracle, determines 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

In Oracle, allows 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 \textit{sec\_relevant\_cols}.

\textit{sec\_relevant\_cols\_opt}

In Oracle, if \textit{sec\_relevant\_cols\_opt} is set to DBMS\_RLS\_ALL\_ROWS (INTEGER constant of value 1), then the columns listed in \textit{sec\_relevant\_cols} return NULL on all rows where the applied policy predicate is false. (If \textit{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.

\textbf{Note:} Advanced Server does not support the DBMS\_RLS\_ALL\_ROWS functionality. Advanced Server throws an error if \textit{sec\_relevant\_cols\_opt} is set to DBMS\_RLS\_ALL\_ROWS (INTEGER value of 1).

\textbf{Examples}

This example uses the following policy function:

\begin{verbatim}
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;
\end{verbatim}

This function generates the predicate \textit{authid = SYS\_CONTEXT(''USERENV'', '''SESSION_USER'')}), which is added to the \texttt{WHERE} clause of any SQL command of the type specified in the \texttt{ADD\_POLICY} procedure.

This limits the effect of the SQL command to those rows where the content of the \textit{authid} column is the same as the session user.

\textbf{Note:} This example uses the \texttt{SYS\_CONTEXT} function to return the login user name. In Oracle the \texttt{SYS\_CONTEXT} function is used to return attributes of an application context. The first parameter of the \texttt{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. \texttt{USERENV} is a special built-in namespace that describes the current session. Postgres Plus Advanced Server does not support application contexts, but only this specific usage of the \texttt{SYS\_CONTEXT} function.

The following anonymous block calls the \texttt{ADD\_POLICY} procedure to create a policy named \texttt{secure\_update} to be applied to the \texttt{vemp} table using function
verify_session_user whenever an INSERT, UPDATE, or DELETE SQL command is given referencing the vpemp table.

```sql
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:

```sql
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 |         |     10 |         
   7566 | JONES  | MANAGER   | 2975.00 |         |     10 |         
   7788 | SCOTT  | ANALYST   | 3000.00 |         |     10 |         
   7876 | ADAMS  | CLERK     | 1100.00 |         |     10 |         
   7902 | FORD   | ANALYST   | 3000.00 |         |     10 |         
   7499 | ALLEN  | SALESMAN  | 1600.00 |  300.00 |     10 |         
   7521 | WARD   | SALESMAN  | 1250.00 |  500.00 |     10 |         
   7654 | MARTIN | SALESMAN  | 1250.00 | 1400.00 |     10 |         
   7698 | BLAKE  | MANAGER   | 2850.00 |         |     10 |         
   7844 | TURNER | SALESMAN  | 1500.00 |    0.00 |     10 |         
   7900 | JAMES  | CLERK     |  950.00 |         |     10 |         
(14 rows)
```

An unqualified UPDATE command (no WHERE clause) is issued by the salesmgr user:

```sql
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.

```sql
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.

```sql
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.

```sql
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`:

```sql
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`:

```sql
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:

```sql
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
```

Copyright © 2007 - 2015 EnterpriseDB Corporation. All rights reserved.
7.7.2 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;
```
## 7.7.3 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;
```
7.8 DBMS_SQL

The DBMS_SQL package provides an Oracle compatible 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 in an Oracle compatible fashion without modifying your application.

DBMS_SQL assumes the privileges of the current user when executing dynamic SQL statements.

Table 7-14 DBMS_SQL Functions/Procedures

<table>
<thead>
<tr>
<th>Function/Procedure</th>
<th>Function or Procedure</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIND_VARIABLE(c, name, value [, out_value_size ])</td>
<td>Procedure n/a</td>
<td>Bind a value to a variable.</td>
<td></td>
</tr>
<tr>
<td>BIND_VARIABLE_CHAR(c, name, value [, out_value_size ])</td>
<td>Procedure n/a</td>
<td>Bind a CHAR value to a variable.</td>
<td></td>
</tr>
<tr>
<td>BIND_VARIABLE_RAW(c, name, value [, out_value_size ])</td>
<td>Procedure n/a</td>
<td>Bind a RAW value to a variable.</td>
<td></td>
</tr>
<tr>
<td>CLOSE_CURSOR(c IN OUT)</td>
<td>Procedure n/a</td>
<td>Close a cursor.</td>
<td></td>
</tr>
<tr>
<td>COLUMN_VALUE(c, position, value OUT [, column_error OUT [, actual_length OUT ]])</td>
<td>Procedure n/a</td>
<td>Return a column value into a variable.</td>
<td></td>
</tr>
<tr>
<td>COLUMN_VALUE_CHAR(c, position, value OUT [, column_error OUT [, actual_length OUT ]])</td>
<td>Procedure n/a</td>
<td>Return a CHAR column value into a variable.</td>
<td></td>
</tr>
<tr>
<td>COLUMN_VALUE_RAW(c, position, value OUT [, column_error OUT [, actual_length OUT ]])</td>
<td>Procedure n/a</td>
<td>Return a RAW column value into a variable.</td>
<td></td>
</tr>
<tr>
<td>DEFINE_COLUMN(c, position, column [, column_size ])</td>
<td>Procedure n/a</td>
<td>Define a column in the SELECT list.</td>
<td></td>
</tr>
<tr>
<td>DEFINE_COLUMN_CHAR(c, position, column, column_size)</td>
<td>Procedure n/a</td>
<td>Define a CHAR column in the SELECT list.</td>
<td></td>
</tr>
<tr>
<td>DEFINE_COLUMN_RAW(c, position, column, column_size)</td>
<td>Procedure n/a</td>
<td>Define a RAW column in the SELECT list.</td>
<td></td>
</tr>
<tr>
<td>DESCRIBE_COLUMNS</td>
<td>Procedure n/a</td>
<td>Defines columns to hold a cursor result set.</td>
<td></td>
</tr>
<tr>
<td>EXECUTE(c)</td>
<td>Function INTEGER</td>
<td>Execute a cursor.</td>
<td></td>
</tr>
<tr>
<td>EXECUTE_AND_FETCH(c [, exact ])</td>
<td>Function INTEGER</td>
<td>Execute a cursor and fetch a single row.</td>
<td></td>
</tr>
<tr>
<td>FETCH_ROWS(c)</td>
<td>Function INTEGER</td>
<td>Fetch rows from the cursor.</td>
<td></td>
</tr>
<tr>
<td>IS_OPEN(c)</td>
<td>Function BOOLEAN</td>
<td>Check if a cursor is open.</td>
<td></td>
</tr>
<tr>
<td>LAST_ROW_COUNT</td>
<td>Function INTEGER</td>
<td>Return cumulative number of rows fetched.</td>
<td></td>
</tr>
<tr>
<td>OPEN_CURSOR</td>
<td>Function INTEGER</td>
<td>Open a cursor.</td>
<td></td>
</tr>
<tr>
<td>PARSE(c, statement, language_flag)</td>
<td>Procedure n/a</td>
<td>Parse a statement.</td>
<td></td>
</tr>
</tbody>
</table>
The following table lists the public variable available in the `DBMS_SQL` package.

### Table 7-15 DBMS_SQL Public Variables

<table>
<thead>
<tr>
<th>Public Variables</th>
<th>Data Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>native</code></td>
<td>INTEGER</td>
<td>1</td>
<td>Provided for Oracle syntax compatibility. See <code>DBMS_SQL.PARSE</code> for more information.</td>
</tr>
<tr>
<td><code>V6</code></td>
<td>INTEGER</td>
<td>2</td>
<td>Provided for Oracle syntax compatibility. See <code>DBMS_SQL.PARSE</code> for more information.</td>
</tr>
<tr>
<td><code>V7</code></td>
<td>INTEGER</td>
<td>3</td>
<td>Provided for Oracle syntax compatibility. See <code>DBMS_SQL.PARSE</code> for more information.</td>
</tr>
</tbody>
</table>

### 7.8.1 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.

```sql
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.

```sql
DECLARE
  curid INTEGER;
  v_sql VARCHAR2(150) := 'INSERT INTO emp VALUES ' ||
```
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);
    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

### 7.8.2 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**

```
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.

7.8.3 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.
7.8.4 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;
```

7.8.5 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.

```sql
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,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;
```
7.8.6 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.

7.8.7 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.

### 7.8.8 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.
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.

```plsql
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.

```plsql
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_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;
```
7.8.9 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.

\[
\text{DEFINE_COLUMN_CHAR}(c \text{ INTEGER}, \text{ position INTEGER}, \text{ column CHAR}, \text{ column_size INTEGER})
\]

**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 \text{column_size} is truncated to \text{column_size} characters.

7.8.10 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.

\[
\text{DEFINE_COLUMN_RAW}(c \text{ INTEGER}, \text{ position INTEGER}, \text{ column RAW}, \text{ column_size INTEGER})
\]
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.

7.8.11 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:

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>col_type</td>
<td>INTEGER</td>
</tr>
<tr>
<td>col_max(len)</td>
<td>INTEGER</td>
</tr>
<tr>
<td>col_name</td>
<td>VARCHAR2(128)</td>
</tr>
<tr>
<td>col_name_len</td>
<td>INTEGER</td>
</tr>
</tbody>
</table>
### 7.8.12 EXECUTE

The **EXECUTE** function executes a parsed SQL command or SPL block.

\[
\text{status INTEGER EXECUTE}(c \text{ INTEGER})
\]

**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.

```sql
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;
```

### 7.8.13 EXECUTE_AND_FETCH

Function **EXECUTE_AND_FETCH** executes a parsed SELECT command and fetches one row.

\[
\text{status INTEGER EXECUTE_AND_FETCH}(c \text{ INTEGER}
\quad [, \text{ exact BOOLEAN }])
\]
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.

```sql
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.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);
END;
```
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

7.8.14 FETCH_ROWS

The FETCH_ROWS function retrieves a row from a cursor.

```sql
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 examples 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,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;

<table>
<thead>
<tr>
<th>EMPNO</th>
<th>ENAME</th>
<th>HIREDATE</th>
<th>SAL</th>
<th>COMM</th>
</tr>
</thead>
<tbody>
<tr>
<td>7369</td>
<td>SMITH</td>
<td>1980-12-17</td>
<td>800.00</td>
<td>.00</td>
</tr>
<tr>
<td>7499</td>
<td>ALLEN</td>
<td>1981-02-20</td>
<td>1,600.00</td>
<td>300.00</td>
</tr>
<tr>
<td>7521</td>
<td>WARD</td>
<td>1981-02-22</td>
<td>1,250.00</td>
<td>500.00</td>
</tr>
<tr>
<td>7566</td>
<td>JONES</td>
<td>1981-04-02</td>
<td>2,975.00</td>
<td>.00</td>
</tr>
<tr>
<td>7654</td>
<td>MARTIN</td>
<td>1981-03-28</td>
<td>1,250.00</td>
<td>1,400.00</td>
</tr>
<tr>
<td>7698</td>
<td>BLAKE</td>
<td>1981-05-01</td>
<td>2,850.00</td>
<td>.00</td>
</tr>
<tr>
<td>7782</td>
<td>CLARK</td>
<td>1981-06-09</td>
<td>2,450.00</td>
<td>.00</td>
</tr>
<tr>
<td>7788</td>
<td>SCOTT</td>
<td>1987-04-19</td>
<td>3,000.00</td>
<td>.00</td>
</tr>
<tr>
<td>7839</td>
<td>KING</td>
<td>1981-11-17</td>
<td>5,000.00</td>
<td>.00</td>
</tr>
<tr>
<td>7844</td>
<td>TURNER</td>
<td>1981-09-08</td>
<td>1,500.00</td>
<td>.00</td>
</tr>
<tr>
<td>7876</td>
<td>ADAMS</td>
<td>1987-05-23</td>
<td>1,100.00</td>
<td>.00</td>
</tr>
<tr>
<td>7900</td>
<td>JAMES</td>
<td>1981-12-03</td>
<td>950.00</td>
<td>.00</td>
</tr>
<tr>
<td>7902</td>
<td>FORD</td>
<td>1981-12-03</td>
<td>3,000.00</td>
<td>.00</td>
</tr>
<tr>
<td>7934</td>
<td>MILLER</td>
<td>1982-01-23</td>
<td>1,300.00</td>
<td>.00</td>
</tr>
</tbody>
</table>
7.8.15 IS_OPEN

The IS_OPEN function provides the capability to test if the given cursor is open.

\[ \text{status} \text{ BOOLEAN IS_OPEN}(c \text{ INTEGER}) \]

Parameters

c

Cursor ID of the cursor to be tested.

status

Set to \text{TRUE} if the cursor is open, set to \text{FALSE} if the cursor is not open.

7.8.16 LAST_ROW_COUNT

The LAST_ROW_COUNT function returns the number of rows that have been currently fetched.

\[ \text{rowcnt} \text{ INTEGER LAST_ROW_COUNT} \]

Parameters

rowcnt

Number of row fetched thus far.

Examples

The following example uses the \text{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,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

7.8.17 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
Cursor ID number associated with the newly created cursor.

Examples

The following example creates a new cursor:

```sql
DECLARE
   curid           INTEGER;
BEGIN
   curid := DBMS_SQL.OPEN_CURSOR;
   .
   .
END;
```

7.8.18  **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`
  
  Language flag provided for Oracle syntax compatibility. Use `DBMS_SQL.V6`, `DBMS_SQL.V7` or `DBMS_SQL.native`. This flag is ignored, and all syntax is assumed to be in EnterpriseDB 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;

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;
7.9 DBMS_UTILITY

The DBMS_UTILITY package provides various utility programs.

Table 7-16 DBMS_UTILITY Functions/Procedures

<table>
<thead>
<tr>
<th>Function/Procedure</th>
<th>Function or Procedure</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANALYZE_DATABASE(method [, estimate_rows [, estimate_percent [, method_opt ]]])</td>
<td>Procedure</td>
<td>n/a</td>
<td>Analyze database tables.</td>
</tr>
<tr>
<td>ANALYZE_PART_OBJECT(schema, object_name [, object_type [, command_type [, command_opt [, sample_clause ]]]])</td>
<td>Procedure</td>
<td>n/a</td>
<td>Analyze a partitioned table.</td>
</tr>
<tr>
<td>ANALYZE_SCHEMA(schema, method [, estimate_rows [, estimate_percent [, method_opt ]]])</td>
<td>Procedure</td>
<td>n/a</td>
<td>Analyze schema tables.</td>
</tr>
<tr>
<td>CANONICALIZE(name, canon_name OUT, canon_len)</td>
<td>Procedure</td>
<td>n/a</td>
<td>Canonicalizes a string – e.g., strips off white space.</td>
</tr>
<tr>
<td>COMMA_TO_TABLE(list, tablen OUT, tab OUT)</td>
<td>Procedure</td>
<td>n/a</td>
<td>Convert a comma-delimited list of names to a table of names.</td>
</tr>
<tr>
<td>DB_VERSION(version OUT, compatibility OUT)</td>
<td>Procedure</td>
<td>n/a</td>
<td>Get the database version.</td>
</tr>
<tr>
<td>EXEC_DDL_STATEMENT(parse_string)</td>
<td>Procedure</td>
<td>n/a</td>
<td>Execute a DDL statement.</td>
</tr>
<tr>
<td>FORMAT_CALL_STACK</td>
<td>Function</td>
<td>TEXT</td>
<td>Formats the current call stack.</td>
</tr>
<tr>
<td>GET_CPU_TIME</td>
<td>Function</td>
<td>NUMBER</td>
<td>Get the current CPU time.</td>
</tr>
<tr>
<td>GET_DEPENDENCY(type, schema, name)</td>
<td>Procedure</td>
<td>n/a</td>
<td>Get objects that are dependent upon the given object..</td>
</tr>
<tr>
<td>GET_HASH_VALUE(name, base, hash_size)</td>
<td>Function</td>
<td>NUMBER</td>
<td>Compute a hash value.</td>
</tr>
<tr>
<td>GET_PARAMETER_VALUE(parnam, intval OUT, strval OUT)</td>
<td>Procedure</td>
<td>BINARY_INTEGER</td>
<td>Get database initialization parameter settings.</td>
</tr>
<tr>
<td>GET_TIME</td>
<td>Function</td>
<td>NUMBER</td>
<td>Get the current time.</td>
</tr>
<tr>
<td>NAME_TOKENIZE(name, a OUT, b OUT, c OUT, dblink OUT, nextpos OUT)</td>
<td>Procedure</td>
<td>n/a</td>
<td>Parse the given name into its component parts.</td>
</tr>
<tr>
<td>TABLE_TO_COMMA(tab, tablen OUT, list OUT)</td>
<td>Procedure</td>
<td>n/a</td>
<td>Convert a table of names to a comma-delimited list.</td>
</tr>
</tbody>
</table>

The following table lists the public variables available in the DBMS_UTILITY package.

Table 7-17 DBMS_UTILITY Public Variables

<table>
<thead>
<tr>
<th>Public Variables</th>
<th>Data Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>inv_error_on_restrictions</td>
<td>PLS_INTEGER</td>
<td>1</td>
<td>Used by the INVALIDATE procedure.</td>
</tr>
<tr>
<td>lname_array</td>
<td>TABLE</td>
<td></td>
<td>For lists of long names.</td>
</tr>
<tr>
<td>uncl_array</td>
<td>TABLE</td>
<td></td>
<td>For lists of users and names.</td>
</tr>
</tbody>
</table>
7.9.1 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;
```

7.9.2 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;
```

7.9.3 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 analyses 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: \textit{T} – table, \textit{I} – index.

This argument is ignored, but is included for compatibility.

\textit{command\_type}

Type of analyze functionality to perform. Valid values are: \textit{E} - gather estimated statistics based upon on a specified number of rows or a percentage of rows in the \textit{sample\_clause} clause; \textit{C} - compute exact statistics; or \textit{V} – validate the structure and integrity of the partitions.

This argument is ignored, but is included for compatibility.

\textit{command\_opt}

For \textit{command\_type} \textit{C} or \textit{E}, can be any combination of:

- \texttt{[ FOR TABLE ]}
- \texttt{[ FOR ALL COLUMNS ]}
- \texttt{[ FOR ALL LOCAL INDEXES ]}

For \textit{command\_type} \textit{V}, can be \texttt{CASCADE} if \textit{object\_type} is \textit{T}.

This argument is ignored, but is included for compatibility.

\textit{sample\_clause}

If \textit{command\_type} is \textit{E}, contains the following clause to specify the number of rows or percentage or rows on which to base the estimate.

\texttt{SAMPLE n \{ ROWS | PERCENT \}}

This argument is ignored, but is included for compatibility.

\subsection{7.9.4 CANONICALIZE}

The \texttt{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.

```sql
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.

```sql
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.g"hi')
Canonicalized name ==>"abc.def"."GHI"<==
Length: 15

### 7.9.5 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.

```sql
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 7.9.1](#)).
A DBMS_UTILITY UNCL_ARRAY (as described in Section 7.9.2).

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
```

7.9.6 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);  
```
7.9.7 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.

```sql
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
```

In this case, Advanced Server's behavior differs from Oracle's; Oracle accepts the invalid parse_string without complaint.

7.9.8 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.

7.9.9 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
```

7.9.10 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.

case

Name of the object for which dependencies are to be obtained.

Examples

The following anonymous block finds dependencies on the EMP table.

```sql
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()```

7.9.11 GET_HASH_VALUE

The GET_HASH_VALUE function provides the capability to compute a hash value for a given string.

```sql
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.

```sql
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
```

7.9.12  GET_PARAMETER_VALUE

The `GET_PARAMETER_VALUE` procedure provides the capability to retrieve database initialization parameter settings.

```sql
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.

```sql
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
```

### 7.9.13 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
```

### 7.9.14 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.

```sql
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
  DBMSUTILITY.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:

```sql
BEGIN
  name_tokenize('emp');
END;
```

name : emp
 a : EMP
 b :
  c :
 dblink :
 nextpos: 3

Tokenize the name, edb.list_emp:

```sql
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:

```sql
BEGIN
  name_tokenize('"edb"."Emp_Admin".update_emp_sal');
END;
```
7.9.15 TABLE_TO_COMM

The TABLE_TO_COMM 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_COMM(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 7.9.1).

**UNCL_ARRAY**

A DBMS_UTILITY UNCL_ARRAY (as described in Section 7.9.2).

**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.

```sql
CREATE OR REPLACE PROCEDURE table_to_comma (p_list VARCHAR2)
IS
  r_lname DBMSUTILITY.LNAME_ARRAY;
  v_length BINARY_INTEGER;
  v_listlen BINARY_INTEGER;
  v_list VARCHAR2(80);
BEGIN
  DBMSUTILITY.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('------------------');
  DBMSUTILITY.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
7.10 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 7-7-18 UTL_FILE Functions/Procedures

<table>
<thead>
<tr>
<th>Function/Procedure</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FCLOSE(file IN OUT)</td>
<td>n/a</td>
<td>Closes the specified file identified by file.</td>
</tr>
<tr>
<td>FCLOSE_ALL</td>
<td>n/a</td>
<td>Closes all open files.</td>
</tr>
<tr>
<td>FCOPY(location, filename, dest_dir, dest_file [, start_line [, end_line ]])</td>
<td>n/a</td>
<td>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.</td>
</tr>
<tr>
<td>FFLUSH(file)</td>
<td>n/a</td>
<td>Forces data in the buffer to be written to disk in the file identified by file.</td>
</tr>
<tr>
<td>FOPEN(location, filename, open_mode [, max_linesize ]])</td>
<td>FILE_TYPE</td>
<td>Opens file, filename, in the directory identified by location.</td>
</tr>
<tr>
<td>FREMOVE(location, filename)</td>
<td>n/a</td>
<td>Removes the specified file from the file system.</td>
</tr>
<tr>
<td>FRENAGE(location, filename, dest_dir, dest_file [, overwrite ])</td>
<td>n/a</td>
<td>Renames the specified file.</td>
</tr>
<tr>
<td>GET_LINE(file, buffer OUT)</td>
<td>n/a</td>
<td>Reads a line of text into variable, buffer, from the file identified by file.</td>
</tr>
<tr>
<td>IS_OPEN(file)</td>
<td>BOOLEAN</td>
<td>Determines whether or not the given file is open.</td>
</tr>
<tr>
<td>NEW_LINE(file [, lines ])</td>
<td>n/a</td>
<td>Writes an end-of-line character sequence into the file.</td>
</tr>
</tbody>
</table>
### Function/Procedure

<table>
<thead>
<tr>
<th>Function/Procedure</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUT(file, buffer)</td>
<td>n/a</td>
<td>Writes buffer to the given file. PUT does not write an end-of-line character sequence.</td>
</tr>
<tr>
<td>PUT_LINE(file, buffer)</td>
<td>n/a</td>
<td>Writes buffer to the given file. An end-of-line character sequence is added by the PUT_LINE procedure.</td>
</tr>
<tr>
<td>PUTF(file, format [, arg1 ] [, ...])</td>
<td>n/a</td>
<td>Writes a formatted string to the given file. Up to five substitution parameters, arg1,...arg5 may be specified for replacement in format.</td>
</tr>
</tbody>
</table>

### 7.10.1 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.

### 7.10.2 FCLOSE_ALL

The FCLOSE_ALL procedures closes all open files. The procedure executes successfully even if there are no open files to close.

FCLOSE_ALL

### 7.10.3 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.

```sql
CREATE DIRECTORY empdir AS 'C:/TEMP/EMPDIR';
DECLARE
    v_empfile       UTL_FILE.F
    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

7.10.4 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) := 'emppfile.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;

7.10.5 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. This behavior is not Oracle compatible - Oracle does count the end-of-line character(s).
filetype

Variable of type FILE_TYPE containing the file handle of the opened file.

7.10.6 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

7.10.7 FRENAME

The FRENAME procedure renames a given file. This effectively moves a file from one location to another.

FRENAME(location VARCHAR2, filename VARCHAR2,
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.

```sql
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 || '';');
END 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,TAMES,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

7.10.8 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
7696,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
7888,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

7.10.9 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.
7.10.10 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.

```sql
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);
END;

Created file: empfile.csv
```

This file is then displayed:
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.

```sql
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 00:00:00,1600.00,500.00,30
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,BLACK,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

7.10.12 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 ||
                   ',' || 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,7782,23-MAY-87 00:00:00,1100.00,,20
7900,JAMES,CLERK,7698,03-DEC-81 00:00:00,950.00,,30
7902,FOX,ANALYST,7566,03-DEC-81 00:00:00,3000.00,,20
7934,MILLER,CLERK,7782,23-JAN-82 00:00:00,1300.00,,10
```
7.10.13 PUTF

The PUTF procedure writes a formatted string to the given file.

```sql
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 Postgres Plus Advanced Server, a new line character must be specified with two consecutive backslashes instead of one - \\n. This characteristic is not Oracle compatible.

**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 which are not Oracle compatible.

```
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
Salary: $%s Commission: $%s
';
    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
7.11 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 7-19 UTL_MAIL Functions/Procedures

<table>
<thead>
<tr>
<th>Function/Procedure</th>
<th>Function or Procedure</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEND(sender, recipients, cc, bcc, subject, message [, mime_type [, priority ]])</td>
<td>Procedure</td>
<td>n/a</td>
<td>Packages and sends an e-mail to an SMTP server.</td>
</tr>
<tr>
<td>SEND_ATTACH_RAW(sender, recipients, cc, bcc, subject, message, mime_type, priority, attachment [, att_inline [, att_mime_type [, att_filename ]]]))</td>
<td>Procedure</td>
<td>n/a</td>
<td>Same as the SEND procedure, but with BYTEA or large object attachments.</td>
</tr>
<tr>
<td>SEND_ATTACH_VARCHAR2(sender, recipients, cc, bcc, subject, message, mime_type, priority, attachment [, att_inline [, att_mime_type [, att_filename ]]]))</td>
<td>Procedure</td>
<td>n/a</td>
<td>Same as the SEND procedure, but with VARCHAR2 attachments.</td>
</tr>
</tbody>
</table>

7.11.1 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.

```sql
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. 15th.';
    UTL_MAIL.SEND(v_sender,v_recipients,NULL,NULL,v_subj,v_msg);
END;
```

---

**7.11.2 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:

```sql
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]
```
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.

7.11.3 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.
7.12 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 7-20 UTL_SMTP Functions/Procedures

<table>
<thead>
<tr>
<th>Function/Procedure</th>
<th>Function or Procedure</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLOSE_DATA(c IN OUT)</td>
<td>Procedure</td>
<td>n/a</td>
<td>Ends an e-mail message.</td>
</tr>
<tr>
<td>COMMAND(c IN OUT, cmd [, arg ])</td>
<td>Both</td>
<td>REPLY</td>
<td>Execute an SMTP command.</td>
</tr>
<tr>
<td>COMMAND REPLIES(c IN OUT, cmd [, arg ])</td>
<td>Function</td>
<td>REPLIES</td>
<td>Execute an SMTP command where multiple reply lines are expected.</td>
</tr>
<tr>
<td>DATA(c IN OUT, body VARCHAR2)</td>
<td>Procedure</td>
<td>n/a</td>
<td>Specify the body of an e-mail message.</td>
</tr>
<tr>
<td>EHLO(c IN OUT, domain)</td>
<td>Procedure</td>
<td>n/a</td>
<td>Perform initial handshaking with an SMTP server and return extended information.</td>
</tr>
<tr>
<td>HELO(c IN OUT, domain)</td>
<td>Procedure</td>
<td>n/a</td>
<td>Perform initial handshaking with an SMTP server</td>
</tr>
<tr>
<td>HELP(c IN OUT [, command ])</td>
<td>Function</td>
<td>REPLIES</td>
<td>Send the HELP command.</td>
</tr>
<tr>
<td>MAIL(c IN OUT, sender [, parameters ])</td>
<td>Procedure</td>
<td>n/a</td>
<td>Start a mail transaction.</td>
</tr>
<tr>
<td>NOOP(c IN OUT)</td>
<td>Both</td>
<td>REPLY</td>
<td>Send the null command.</td>
</tr>
<tr>
<td>OPEN_CONNECTION(host [, port [, tx_timeout ]])</td>
<td>Function</td>
<td>CONNECTION</td>
<td>Open a connection.</td>
</tr>
<tr>
<td>OPEN_DATA(c IN OUT)</td>
<td>Both</td>
<td>REPLY</td>
<td>Send the DATA command.</td>
</tr>
<tr>
<td>QUIT(c IN OUT)</td>
<td>Procedure</td>
<td>n/a</td>
<td>Terminate the SMTP session and disconnect.</td>
</tr>
<tr>
<td>RCPT(c IN OUT, recipient [, parameters ])</td>
<td>Procedure</td>
<td>n/a</td>
<td>Specify the recipient of an e-mail message.</td>
</tr>
<tr>
<td>RSET(c IN OUT)</td>
<td>Procedure</td>
<td>n/a</td>
<td>Terminate the current mail transaction.</td>
</tr>
<tr>
<td>VRFY(c IN OUT, recipient)</td>
<td>Function</td>
<td>REPLIES</td>
<td>Validate an e-mail address.</td>
</tr>
<tr>
<td>WRITE_DATA(c IN OUT, data)</td>
<td>Procedure</td>
<td>n/a</td>
<td>Write a portion of the e-mail message.</td>
</tr>
</tbody>
</table>

The following table lists the public variables available in the UTL_SMTP package.

Table 7-21 UTL_SMTP Public Variables

<table>
<thead>
<tr>
<th>Public Variables</th>
<th>Data Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>connection</td>
<td>RECORD</td>
<td>Value</td>
<td>Description of an SMTP connection.</td>
</tr>
<tr>
<td>reply</td>
<td>RECORD</td>
<td>Value</td>
<td>SMTP reply line.</td>
</tr>
</tbody>
</table>
7.12.1 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
);
```

7.12.2 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;
```

7.12.3 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.
7.12.4 COMMAND

The COMMAND procedure provides the capability to execute an SMTP command. If you are expecting multiple reply lines, use COMMAND_REPLIES.

```sql
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 7.12.2 for a description of REPLY and REPLIES.

7.12.5 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.

```sql
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 7.12.2 for a description of \texttt{REPLY} and \texttt{REPLIES}.

\section{DATA}

The \texttt{DATA} procedure provides the capability to specify the body of the e-mail message. The message is terminated with a \texttt{<CR><LF>..<CR><LF>} sequence.

\texttt{DATA}(c \texttt{IN OUT CONNECTION, body VARCHAR2})

\textbf{Parameters}

\textit{c}

The SMTP connection to which the command is to be sent.

\textit{body}

Body of the e-mail message to be sent.

\section{EHLO}

The \texttt{EHLO} procedure performs initial handshaking with the SMTP server after establishing the connection. The \texttt{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 \texttt{HELO} procedure performs the equivalent functionality, but returns less information about the server.

\texttt{EHLO}(c \texttt{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.

### 7.12.8 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.

### 7.12.9 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 7.12.2 for a description of `REPLY` and `REPLIES`.

7.12.10 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.

7.12.11 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 7.12.2 for a description of REPLY and REPLIES.

7.12.12 OPEN_CONNECTION

The OPEN_CONNECTION functions open a connection to an SMTP server.

\[
c \text{ CONNECTION OPEN_CONNECTION}(\text{host VARCHAR2 [, port PLS_INTEGER [, tx_timeout PLS_INTEGER DEFAULT NULL]}])
\]

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.

7.12.13 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.

7.12.14 QUIT

The QUIT procedure closes the session with an SMTP server.

QUIT(c IN OUT CONNECTION)

Parameters

c

SMTP connection to be terminated.

7.12.15 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.
7.12.16  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.

7.12.17  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 7.12.2 for a description of REPLY and REPLIES.

7.12.18  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
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.

### 7.12.19 Comprehensive Example

The following procedure constructs and sends a text e-mail message using the UTL_SMTP package.

```sql
CREATE OR REPLACE PROCEDURE send_mail (  
p_sender        VARCHAR2,  
p_recipient     VARCHAR2,  
p_subj          VARCHAR2,  
p_msg           VARCHAR2,  
p_mailhost      VARCHAR2
) IS  
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    || v_crlf, 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.

```sql
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');
8 Object Types and Objects

This chapter discusses how object-oriented programming techniques can be exploited in SPL. Object-oriented programming as seen in programming languages such as Java and C++ centers on the concept of objects. An object is a representation of a real-world entity such as a person, place, or thing. The generic description or definition of a particular object such as a person for example, is called an object type. Specific people such as “Joe” or “Sally” are said to be objects of object type, person, or equivalently, instances of the object type, person, or simply, person objects.

Note: The terms “database objects” and “objects” that have been used in this document up to this point should not be confused with an object type and object as used in this chapter. The previous usage of these terms relates to the entities that can be created in a database such as tables, views, indexes, users, etc. Within the context of this chapter, object type and object refer to specific data structures supported by the SPL programming language to implement object-oriented concepts.

Note: In Oracle, the term abstract data type (ADT) is used to describe object types in PL/SQL. The SPL implementation of object types is intended to be compatible with Oracle abstract data types.

Note: Implementation of SPL object types and objects is following a phased approach. As of this release, support of certain features of most object-oriented programming languages have not yet been implemented. This chapter documents only those features that have currently been implemented.

8.1 Basic Object Concepts

As was stated at the beginning of this chapter, an object type is a description or definition of some entity. This definition of an object type is characterized by two components:

- **Attributes** – fields that describe particular characteristics of an object instance. For a person object, examples might be name, address, gender, date of birth, height, weight, eye color, occupation, etc.
- **Methods** – programs that perform some type of function or operation on, or related to an object. For a person object, examples might be calculating the person’s age, displaying the person’s attributes, changing the values assigned to the person’s attributes, etc.

The following sections elaborate on some basic object concepts.
8.1.1 Attributes

Every object type must contain at least one attribute. The data type of an attribute can be any of the following:

- A base data type such as NUMBER, VARCHAR2, etc.
- Another object type
- A globally defined collection type (created by the CREATE TYPE command) such as a nested table or varray

An attribute gets its initial value (which may be null) when an object instance is initially created. Each object instance has its own set of attribute values.

8.1.2 Methods

Methods are SPL procedures or functions defined within an object type. Methods can be categorized into three general types:

- Member Methods – procedures or functions that operate within the context of an object instance. Member methods have access to, and can change the attributes of the object instance on which they are operating.
- Static Methods – procedures or functions that operate independently of any particular object instance. Static methods do not have access to, and cannot change the attributes of an object instance.
- Constructor Methods – functions used to create an instance of an object type. A default constructor method is always provided when an object type is defined.

Note: SPL currently supports member methods, static methods, and the default constructor method. Explicit creation of user-defined constructor methods is not supported.

8.1.3 Overloading Methods

In an object type it is permissible to define two or more identically named methods of the same type (this is, either a procedure or function), but with different signatures. Such methods are referred to as overloaded methods.

A method’s signature consists of the number of formal parameters, the data types of its formal parameters, and their order.
8.2 Object Type Components

Object types are created and stored in the database by using the following two constructs of the SPL language:

- The **object type specification** - This is the public interface specifying the attributes and method signatures of the object type.
- The **object type body** - This contains the implementation of the methods specified in the object type specification.

The following sections describe the commands to create the object type specification and the object type body.

8.2.1 Object Type Specification Syntax

The following is the syntax of the object type specification:

```sql
CREATE [ OR REPLACE ] TYPE name
[ AUTHID { DEFINER | CURRENT_USER } ]
{ IS | AS } OBJECT
( { attribute { datatype | objtype | collecttype } }
[ , ... ]
[ method_spec ] [ , ... ]
) [ [ NOT ] { FINAL | INSTANTIABLE } ] ...;
```

where `method_spec` is the following:

```sql
[ [ NOT ] { FINAL | INSTANTIABLE } ] ...
[ OVERRIDING ]
subprogram_spec
```

where `subprogram_spec` is the following:

```sql
{ MEMBER | STATIC }
{ PROCEDURE proc_name
[ ( [ SELF [ IN | IN OUT ] name ]
[, parm1 [ IN | IN OUT | OUT ] datatype1
[ DEFAULT value1 ] ]
[, parm2 [ IN | IN OUT | OUT ] datatype2
[ DEFAULT value2 ]
] ...)
]

| FUNCTION func_name
[ ( [ SELF [ IN | IN OUT ] name ]
[, parm1 [ IN | IN OUT | OUT ] datatype1
[ DEFAULT value1 ] ]
[, parm2 [ IN | IN OUT | OUT ] datatype2
```
[ DEFAULT value2 ]

] ...) } }

RETURN return_type } }

**Note:** The OR REPLACE option cannot be currently used to add, delete, or modify the attributes of an existing object type. Use the DROP TYPE command to first delete the existing object type. The OR REPLACE option can be used to add, delete, or modify the methods in an existing object type.

**Note:** The PostgreSQL form of the ALTER TYPE ALTER ATTRIBUTE command can be used to change the data type of an attribute in an existing object type. However, the ALTER TYPE command cannot add or delete attributes in the object type.

name is an identifier (optionally schema-qualified) assigned to the object type.

If the AUTHID clause is omitted or DEFINER is specified, the rights of the object type owner are used to determine access privileges to database objects. If CURRENT_USER is specified, the rights of the current user executing a method in the object are used to determine access privileges.

attribute is an identifier assigned to an attribute of the object type. datatype is a base data type. objtype is a previously defined object type. collecttype is a previously defined collection type.

Following the closing parenthesis of the CREATE TYPE definition, [ NOT ] FINAL specifies whether or not a subtype can be derived from this object type. FINAL, which is the default, means that no subtypes can be derived from this object type. Specify NOT FINAL if you want to allow subtypes to be defined under this object type.

**Note:** Even though the specification of NOT FINAL is accepted in the CREATE TYPE command, SPL does not currently support the creation of subtypes.

Following the closing parenthesis of the CREATE TYPE definition, [ NOT ] INSTANTIABLE specifies whether or not an object instance can be created of this object type. INSTANTIABLE, which is the default, means that an instance of this object type can be created. Specify NOT INSTANTIABLE if this object type is to be used only as a parent “template” from which other specialized subtypes are to be defined. If NOT INSTANTIABLE is specified, then NOT FINAL must be specified as well. If any method in the object type contains the NOT INSTANTIABLE qualifier, then the object type, itself, must be defined with NOT INSTANTIABLE and NOT FINAL.

**Note:** Even though the specification of NOT INSTANTIABLE is accepted in the CREATE TYPE command, SPL does not currently support the creation of subtypes.
method_spec denotes the specification of a member method or static method.

Prior to the definition of a method, \[ \text{NOT } \text{FINAL} \] specifies whether or not the method can be overridden in a subtype. \text{NOT FINAL} is the default meaning the method can be overridden in a subtype.

Prior to the definition of a method specify \texttt{OVERRIDING} if the method overrides an identically named method in a supertype. The overriding method must have the same number of identically named method parameters with the same data types and parameter modes, in the same order, and the same return type (if the method is a function) as defined in the supertype.

Prior to the definition of a method, \[ \text{NOT } \text{INSTANTIABLE} \] specifies whether or not the object type definition provides an implementation for the method. If \texttt{INSTANTIABLE} is specified, then the \texttt{CREATE TYPE BODY} command for the object type must specify the implementation of the method. If \texttt{NOT INSTANTIABLE} is specified, then the \texttt{CREATE TYPE BODY} command for the object type must not contain the implementation of the method. In this latter case, it is assumed a subtype contains the implementation of the method, overriding the method in this object type. If there are any \texttt{NOT INSTANTIABLE} methods in the object type, then the object type definition itself, must specify \texttt{NOT INSTANTIABLE} and \texttt{NOT FINAL} following the closing parenthesis of the object type specification. The default is \texttt{INSTANTIABLE}.

subprogram_spec denotes the specification of a procedure or function and begins with the specification of either \texttt{MEMBER} or \texttt{STATIC}. A member subprogram must be invoked with respect to a particular object instance while a static subprogram is not invoked with respect to any object instance.

proc_name is an identifier of a procedure. If the \texttt{SELF} parameter is specified, \texttt{name} is the object type name given in the \texttt{CREATE TYPE} command. If specified, \texttt{parm1, parm2, …} are the formal parameters of the procedure. \texttt{datatype1, datatype2, …} are the data types of \texttt{parm1, parm2, …} respectively. \texttt{IN, IN OUT, and OUT} are the possible parameter modes for each formal parameter. If none are specified, the default is \texttt{IN}. \texttt{value1, value2, …} are default values that may be specified for \texttt{IN} parameters.

func_name is an identifier of a function. If the \texttt{SELF} parameter is specified, \texttt{name} is the object type name given in the \texttt{CREATE TYPE} command. If specified, \texttt{parm1, parm2, …} are the formal parameters of the function. \texttt{datatype1, datatype2, …} are the data types of \texttt{parm1, parm2, …} respectively. \texttt{IN, IN OUT, and OUT} are the possible parameter modes for each formal parameter. If none are specified, the default is \texttt{IN}. \texttt{value1, value2, …} are default values that may be specified for \texttt{IN} parameters. \texttt{return_type} is the data type of the value the function returns.

The following points should be noted about an object type specification:
There must be at least one attribute defined in the object type.
There may be none, one, or more methods defined in the object type.
For a member method there is an implicit, built-in parameter named `SELF` whose data type is that of the object type being defined. `SELF` refers to the object instance that is currently invoking the method. `SELF` can be explicitly declared as an IN or IN OUT parameter in the parameter list. For example as `MEMBER FUNCTION (SELF IN OUT object_type ...)` If explicitly declared, `SELF` must be the first parameter in the parameter list. If `SELF` is not explicitly declared, its parameter mode defaults to IN OUT for member procedures and IN for member functions.
A static method cannot be overridden (OVERRIDING and STATIC cannot be specified together in `method_spec`).
A static method must be instantiable (NOT INSTANTIABLE and STATIC cannot be specified together in `method_spec`).

### 8.2.2 Object Type Body Syntax

The following is the syntax of the object type body:

```sql
CREATE [ OR REPLACE ] TYPE BODY name
{ IS | AS }
method_spec [...] END;
```

where `method_spec` is the following:

```sql
{ MEMBER | STATIC }
{ PROCEDURE proc_name
  [ ( [ SELF [ IN | IN OUT ] name ]
    [, parm1 [ IN | IN OUT | OUT ] datatype1
      [ DEFAULT value1 ] ]
    [, parm2 [ IN | IN OUT | OUT ] datatype2
      [ DEFAULT value2 ]
    ] ...)
  ]
  [ IS | AS ]
  [ variable_declaration; ] ...
BEGIN
  statement; ...
[ EXCEPTION
  WHEN ... THEN
```

Copyright © 2007 - 2015 EnterpriseDB Corporation. All rights reserved.
FUNCTION func_name
| [ ( [ SELF [ IN | IN OUT ] name ]
| [, parm1 [ IN | IN OUT | OUT ] datatype1
| [ DEFAULT value1 ] ]
| [, parm2 [ IN | IN OUT | OUT ] datatype2
| [ DEFAULT value2 ] ] ...) ]
| RETURN return_type
| { IS | AS } [ variable_declaration; ] ... BEGIN
| statement; ... [ EXCEPTION
| WHEN ... THEN
| statement; ...] END;
|

name is an identifier (optionally schema-qualified) assigned to the object type.

method_spec denotes the implementation of an instantiable method that was specified in the CREATE TYPE command.

If INSTANTIABLE was specified or omitted in method_spec of the CREATE TYPE command, then there must be a method_spec for this method in the CREATE TYPE BODY command.

If NOT INSTANTIABLE was specified in method_spec of the CREATE TYPE command, then there must be no method_spec for this method in the CREATE TYPE BODY command.

subprogram_spec denotes the specification of a procedure or function and begins with the specification of either MEMBER or STATIC. The same qualifier must be used as was specified in subprogram_spec of the CREATE TYPE command.

proc_name is an identifier of a procedure specified in the CREATE TYPE command. The parameter declarations have the same meaning as described for the CREATE TYPE command, and must be specified in the CREATE TYPE BODY command in the same manner as specified in the CREATE TYPE command.

func_name is an identifier of a function specified in the CREATE TYPE command. The parameter declarations have the same meaning as described for the CREATE TYPE command, and must be specified in the CREATE TYPE BODY command in the same
manner as specified in the `CREATE TYPE` command. `return_type` is the data type of the value the function returns and must match `return_type` given in the `CREATE TYPE` command.

`variable_declaration` is a declaration of a variable, which can be accessed only from within the subprogram. There can be none, one, or more variable declarations. `statement` is an SPL program statement.
8.3 Creating Object Types

The CREATE TYPE command is used to create the object type specification while the CREATE TYPE BODY command creates the object type body. This section provides some examples of their usage.

The following example creates the addr_obj_typ object type that contains only attributes and no methods.

```
CREATE OR REPLACE TYPE addr_obj_typ AS OBJECT {
    street          VARCHAR2(30),
    city            VARCHAR2(20),
    state           CHAR(2),
    zip             NUMBER(5)
};
```

Since there are no methods in this object type, an object type body is not created for it.

The following object type specification creates the emp_obj_typ object type. In this example, the addr attribute is defined by the addr_obj_typ object type. Object type emp_obj_typ also contains a member method display_emp. The SELF parameter, which passes the object instance on which the method is invoked, is always the first parameter in a member procedure or function regardless of whether or not it is explicitly declared in the parameter list.

```
CREATE OR REPLACE TYPE emp_obj_typ AS OBJECT {
    empno           NUMBER(4),
    ename           VARCHAR2(20),
    addr            ADDR_OBJ_TYP,
    MEMBER PROCEDURE display_emp (SELF IN OUT emp_obj_typ)
};
```

The following is the object type body for emp_obj_typ.

```
CREATE OR REPLACE TYPE BODY emp_obj_typ AS
    MEMBER PROCEDURE display_emp (SELF IN OUT emp_obj_typ)
    IS
    BEGIN
        DBMS_OUTPUT.PUT_LINE('Employee No : ' || empno);
        DBMS_OUTPUT.PUT_LINE('Name    : ' || ename);
        DBMS_OUTPUT.PUT_LINE('Street  : ' || addr.street);
        DBMS_OUTPUT.PUT_LINE('City/State/Zip: ' || addr.city || ', ' || addr.state || ' ' || LPAD(addr.zip,5,'0'));
    END;
END;
```

To illustrate how the SELF parameter would be used in the object type body, the preceding object type body could have been written as follows:

```
CREATE OR REPLACE TYPE BODY emp_obj_typ AS
    MEMBER PROCEDURE display_emp (SELF IN OUT emp_obj_typ)
    IS
    BEGIN
```
Both versions of the `emp_obj_typ` body are completely equivalent.

The following object type specification includes a static function `get_dname` and a member procedure `display_dept`.

```sql
CREATE OR REPLACE TYPE dept_obj_typ AS OBJECT (  
defptno          NUMBER(2),
  STATIC FUNCTION get_dname (p_deptno IN NUMBER) RETURN VARCHAR2,
  MEMBER PROCEDURE display_dept
);
```

The object type body for `dept_obj_typ` defines the implementation of the static function `get_dname` and the member procedure `display_dept`.

```sql
CREATE OR REPLACE TYPE BODY dept_obj_typ AS
  STATIC FUNCTION get_dname (p_deptno IN NUMBER) RETURN VARCHAR2
    IS
      v_dname     VARCHAR2(14);
    BEGIN
      CASE p_deptno
        WHEN 10 THEN v_dname := 'ACCOUNTING';
        WHEN 20 THEN v_dname := 'RESEARCH';
        WHEN 30 THEN v_dname := 'SALES';
        WHEN 40 THEN v_dname := 'OPERATIONS';
        ELSE v_dname := 'UNKNOWN';
      END CASE;
      RETURN v_dname;
    END;
    MEMBER PROCEDURE display_dept
    IS
      BEGIN
        DBMS_OUTPUT.PUT_LINE('Dept No    : ' || SELF.deptno);
        DBMS_OUTPUT.PUT_LINE('Dept Name  : ' || dept_obj_typ.get_dname(SELF.deptno));
      END;
    END;
```

Within the static function `get_dname`, there can be no references to attributes of `dept_obj_typ`. Since a static function is invoked independently of any object instance, it has no access to any object attribute.

Member procedure `display_dept` does access the `deptno` attribute of the object instance passed in the `SELF` parameter. Note that it is not necessary to explicitly declare the `SELF` parameter in the `display_dept` parameter list.

The last `DBMS_OUTPUT.PUT_LINE` statement includes a call to the static function `get_dname` qualified by its object type name `dept_obj_typ`.
8.4 Creating Object Instances

Creating instances of an object type requires the following steps.

- Declare an object variable of the object type
- Initialize the declared object variable with initial values

The syntax for declaring an object variable is as follows.

\[\text{object objtype}\]

*object* is an identifier assigned to the object variable. *objtype* is the identifier of a previously defined object type.

The next step is to initialize the object variable with values. The initialization process is done with a constructor method. The following is the syntax of the constructor method.

\[\text{objtype (\{ expr1 | NULL \}[, \{ expr2 | NULL \}] [, \ldots])}\]

*objtype* is the identifier of the object type’s constructor method, which has the same name as the object type. *expr1, expr2,* … are expressions that are type-compatible with the first attribute of the object type, the second attribute of the object type, etc. If *NULL* is specified, the corresponding object attribute is set to null. If an attribute is of an object type, then the corresponding expression can be null, an object initialization expression, or any expression that returns that object type.

**Note:** In Advanced Server, the following alternate syntax can be used in place of the constructor method.

\[\text{[ ROW ] (\{ expr1 | NULL \}[, \{ expr2 | NULL \}] [, \ldots])}\]

*ROW* is an optional keyword if two or more terms are specified within the parenthesis-enclosed, comma-delimited list. If only one term is specified, then specification of the *ROW* keyword is mandatory.

The following anonymous block declares a variable of type *emp_obj_typ* named *v_emp*, and initializes it.

```
DECLARE
  v_emp           EMP_OBJ_TYP;
BEGIN
  v_emp := emp_obj_typ (9001,'JONES',
                        addr_obj_typ('123 MAIN STREET','EDISON','NJ',08817));
END;
```
8.5 Referencing an Object

Once an object variable is created and initialized, individual attributes can be referenced using dot notation of the form:

\[ \text{object.attribute} \]

\textit{object} is the identifier assigned to the object variable. \textit{attribute} is the identifier of an object type attribute.

If \textit{attribute}, itself, is of an object type, then the reference must take the form:

\[ \text{object.attribute.attribute\_inner} \]

\textit{attribute\_inner} is an identifier belonging to the object type to which \textit{attribute} references in its definition of \textit{object}.

The following example expands upon the previous anonymous block to display the values assigned to the \texttt{emp\_obj\_typ} object.

```sql
DECLARE
  v_emp EMP_OBJ_TYP;
BEGIN
  v_emp := emp_obj_typ(9001,'JONES',
                     addr_obj_typ('123 MAIN STREET','EDISON','NJ',08817));
  DBMS_OUTPUT.PUT_LINE('Employee No   : ' || v_emp.empno);
  DBMS_OUTPUT.PUT_LINE('Name          : ' || v_emp.ename);
  DBMS_OUTPUT.PUT_LINE('Street        : ' || v_emp.addr.street);
  DBMS_OUTPUT.PUT_LINE('City/State/Zip: ' || v_emp.addr.city || ', ' ||
                        v_emp.addr.state || ' ' || LPAD(v_emp.addr.zip,5,'0'));
END;
```

The following is the output from this anonymous block.

<table>
<thead>
<tr>
<th>Employee No</th>
<th>9001</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>JONES</td>
</tr>
<tr>
<td>Street</td>
<td>123 MAIN STREET</td>
</tr>
<tr>
<td>City/State/Zip</td>
<td>EDISON, NJ 08817</td>
</tr>
</tbody>
</table>

Methods are called in a similar manner as attributes.

Once an object variable is created and initialized, member procedures or functions are called using dot notation of the form:

\[ \text{object.prog\_name} \]

\textit{object} is the identifier assigned to the object variable. \textit{prog\_name} is the identifier of the procedure or function.
Static procedures or functions are not called utilizing an object variable. Instead the procedure or function is called utilizing the object type name:

\[ \text{object_type.prog_name} \]

\text{object_type} is the identifier assigned to the object type. \text{prog_name} is the identifier of the procedure or function.

The results of the previous anonymous block can be duplicated by calling the member procedure \text{display_emp}:

```sql
DECLARE
    v_emp EMP_OBJ_TYP;
BEGIN
    v_emp := emp_obj_typ(9001,'JONES',
                        addr_obj_typ('123 MAIN STREET','EDISON','NJ',08817));
    v_emp.display_emp;
END;
```

The following is the output from this anonymous block.

Employee No   : 9001
Name          : JONES
Street        : 123 MAIN STREET
City/State/Zip: EDISON, NJ 08817

The following anonymous block creates an instance of \text{dept_obj_typ} and calls the member procedure \text{display_dept}:

```sql
DECLARE
    v_dept          DEPT_OBJ_TYP := dept_obj_typ (20);
BEGIN
    v_dept.display_dept;
END;
```

The following is the output from this anonymous block.

Dept No    : 20
Dept Name  : RESEARCH

The static function defined in \text{dept_obj_typ} can be called directly by qualifying it by the object type name as follows:

```sql
BEGIN
    DBMS_OUTPUT.PUT_LINE(dept_obj_typ.get_dname(20));
END;
```

RESEARCH
8.6 Dropping an Object Type

The syntax for deleting an object type is as follows.

```
DROP TYPE objtype;
```

`objtype` is the identifier of the object type to be dropped. If the definition of `objtype` contains attributes that are themselves object types or collection types, these nested object types or collection types must be dropped last.

If an object type body is defined for the object type, the `DROP TYPE` command deletes the object type body as well as the object type specification. In order to recreate the complete object type, both the `CREATE TYPE` and `CREATE TYPE BODY` commands must be reissued.

The following example drops the `emp_obj_typ` and the `addr_obj_typ` object types created earlier in this chapter. `emp_obj_typ` must be dropped first since it contains `addr_obj_typ` within its definition as an attribute.

```
DROP TYPE emp_obj_typ;
DROP TYPE addr_obj_typ;
```

The syntax for deleting an object type body, but not the object type specification is as follows.

```
DROP TYPE BODY objtype;
```

The object type body can be recreated by issuing the `CREATE TYPE BODY` command.

The following example drops only the object type body of the `dept_obj_typ`.

```
DROP TYPE BODY dept_obj_typ;
```
9 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 a Postgres Plus 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.

9.1 Comparison with Oracle Call Interface

The following diagram compares the Open Client Library and Oracle Call Interface application stacks.

![Diagram comparing Open Client Library and Oracle Call Interface](image-url)

Figure 6 Open Client Library
9.2 Ref Cursor Support

The Advanced Server Open Client Library supports the use of Oracle-compatible REF CURSOR's as OUT parameters in PL/SQL procedures. 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:

```c
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 executing 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:

```sql
OCIStmtPrepare(stmtOpenRefCursor,
    errhp,
    (text *) openCursor,
    (ub4) strlen(openCursor),
    OCI_NTV_SYNTAX,
    OCI_DEFAULT));
```

Bind the PL/SQL openCursor OUT parameter:

```sql
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:

```sql
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:

```sql
/* Define the output variables for the ref cursor */
OCIDefineByPos(stmtUseRefCursor,
    &defnEmpNo,
    errhp,
    (ub4) 1,
    (dvoid *) &empNo,
    (sb4) sizeof(empNo),
    OCI_DEFAULT);
```
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))
9.3 **OCL 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. Postgres Plus Advanced Server does not supply any such files.

### 9.3.1 Connect, Authorize and Initialize Functions

Table 9-9-1 Connect, Authorize, Terminate and Initialize Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCIBreak</td>
<td>Aborts the specified OCI function.</td>
</tr>
<tr>
<td>OCIEnvCreate</td>
<td>Create an OCI environment.</td>
</tr>
<tr>
<td>OCIEnvInit</td>
<td>Initialize an OCI environment handle.</td>
</tr>
<tr>
<td>OCIInitialize</td>
<td>Initialize the OCI environment.</td>
</tr>
<tr>
<td>OCILogoff</td>
<td>Release a session.</td>
</tr>
<tr>
<td>OCILogon</td>
<td>Create a logon connection.</td>
</tr>
<tr>
<td>OCILogon2</td>
<td>Create a logon session in various modes.</td>
</tr>
<tr>
<td>OCIServerAttach</td>
<td>Establish an access path to a data source.</td>
</tr>
<tr>
<td>OCIServerDetach</td>
<td>Remove access to a data source.</td>
</tr>
<tr>
<td>OCISessionBegin</td>
<td>Create a user session.</td>
</tr>
<tr>
<td>OCISessionEnd</td>
<td>End a user session.</td>
</tr>
<tr>
<td>OCISessionGet</td>
<td>Get session from session pool.</td>
</tr>
<tr>
<td>OCISessionRelease</td>
<td>Release a session.</td>
</tr>
<tr>
<td>OCITerminate</td>
<td>Detach from shared memory subsystem.</td>
</tr>
</tbody>
</table>

### 9.3.2 Handle and Descriptor Functions

Table 9-9-2 Handle and Descriptor Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCIAttrGet</td>
<td>Get handle attributes.</td>
</tr>
<tr>
<td>OCIAttrSet</td>
<td>Set handle attributes.</td>
</tr>
<tr>
<td>OCIPropertyDescriptor</td>
<td>Allocate and initialize a descriptor.</td>
</tr>
<tr>
<td>OCIHandleAlloc</td>
<td>Allocate and initialize a handle.</td>
</tr>
<tr>
<td>OCIHandleFree</td>
<td>Free an allocated handle.</td>
</tr>
<tr>
<td>OCIParamGet</td>
<td>Get a parameter descriptor.</td>
</tr>
<tr>
<td>OCIParamSet</td>
<td>Set a parameter descriptor.</td>
</tr>
</tbody>
</table>

### 9.3.3 Bind, Define and Describe Functions

Table 9-9-3 Bind, Define, and Describe Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCIBindByName</td>
<td>Bind by name.</td>
</tr>
<tr>
<td>OCIBindByPos</td>
<td>Bind by position.</td>
</tr>
<tr>
<td>OCIBindDynamic</td>
<td>Set additional attributes after bind.</td>
</tr>
<tr>
<td>OCIBindArrayOfStruct</td>
<td>Bind an array of structures for bulk operations.</td>
</tr>
<tr>
<td>OCIDefineByPos</td>
<td>Define an output variable association.</td>
</tr>
<tr>
<td>OCIDefineDynamic</td>
<td>Set additional attributes for define.</td>
</tr>
</tbody>
</table>
### 9.3.4 Statement Functions

Table 9-9-4 Statement Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCIStmtExecute</td>
<td>Execute a prepared SQL statement.</td>
</tr>
<tr>
<td>OCIStmtFetch</td>
<td>Fetch rows of data (deprecated).</td>
</tr>
<tr>
<td>OCIStmtFetch2</td>
<td>Fetch rows of data.</td>
</tr>
<tr>
<td>OCIStmtPrepare</td>
<td>Prepare a SQL statement.</td>
</tr>
<tr>
<td>OCIStmtPrepare2</td>
<td>Prepare a SQL statement.</td>
</tr>
<tr>
<td>OCIStmtRelease</td>
<td>Release a statement handle.</td>
</tr>
</tbody>
</table>

### 9.3.5 Transaction Functions

Table 9-9-5 Transaction Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCITransCommit</td>
<td>Commit a transaction.</td>
</tr>
<tr>
<td>OCITransRollback</td>
<td>Roll back a transaction.</td>
</tr>
</tbody>
</table>

### 9.3.6 Date and Datetime Functions

Table 9-9-6 Date and Datetime Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCIDateAddDays</td>
<td>Add or subtract a number of days.</td>
</tr>
<tr>
<td>OCIDateAddMonths</td>
<td>Add or subtract a number of months.</td>
</tr>
<tr>
<td>OCIDateAssign</td>
<td>Assign a date.</td>
</tr>
<tr>
<td>OCIDateCheck</td>
<td>Check if the given date is valid.</td>
</tr>
<tr>
<td>OCIDateCompare</td>
<td>Compare two dates.</td>
</tr>
<tr>
<td>OCIDateDaysBetween</td>
<td>Find the number of days between two dates.</td>
</tr>
<tr>
<td>OCIDateFromText</td>
<td>Convert a string to a date.</td>
</tr>
<tr>
<td>OCIDateGetDate</td>
<td>Get the date portion of a date.</td>
</tr>
<tr>
<td>OCIDateGetTime</td>
<td>Get the time portion of a date.</td>
</tr>
<tr>
<td>OCIDateLastDay</td>
<td>Get the date of the last day of the month.</td>
</tr>
<tr>
<td>OCIDateNextDay</td>
<td>Get the date of the next day.</td>
</tr>
<tr>
<td>OCIDateSetDate</td>
<td>Set the date portion of a date.</td>
</tr>
<tr>
<td>OCIDateSetTime</td>
<td>Set the time portion of a date.</td>
</tr>
<tr>
<td>OCIDateSysDate</td>
<td>Get the current system date and time.</td>
</tr>
<tr>
<td>OCIDateToText</td>
<td>Convert a date to a string.</td>
</tr>
<tr>
<td>OCIDateTimeAssign</td>
<td>Perform datetime assignment.</td>
</tr>
<tr>
<td>OCIDateTimeCheck</td>
<td>Check if the date is valid.</td>
</tr>
<tr>
<td>OCIDateTimeCompare</td>
<td>Compare two datetime values.</td>
</tr>
<tr>
<td>OCIDateTimeConstruct</td>
<td>Construct a datetime descriptor.</td>
</tr>
<tr>
<td>OCIDateTimeConvert</td>
<td>Convert one datetime type to another.</td>
</tr>
<tr>
<td>OCIDateTimeFromArray</td>
<td>Convert an array of size OCI_DT_ARRAYLEN to an OCIDateTime descriptor.</td>
</tr>
<tr>
<td>OCIDateTimeFromText</td>
<td>Convert the given string to Oracle datetime type in the</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>OCIDateTimeGetDate</td>
<td>OCIDateTime descriptor according to the specified format.</td>
</tr>
<tr>
<td>OCIDateTimeGetTime</td>
<td>Get the date portion of a datetime value.</td>
</tr>
<tr>
<td>OCIDateTimeGetTimeZoneName</td>
<td>Get the time zone name portion of a datetime value.</td>
</tr>
<tr>
<td>OCIDateTimeGetTimeZoneOffset</td>
<td>Get the time zone (hour, minute) portion of a datetime value.</td>
</tr>
<tr>
<td>OCIDateTimeSubtract</td>
<td>Take two datetime values as input and return their difference as an interval.</td>
</tr>
<tr>
<td>OCIDateTimeSysTimeStamp</td>
<td>Get the system current date and time as a timestamp with time zone.</td>
</tr>
<tr>
<td>OCIDateTimeToArray</td>
<td>Convert an OCIDateTime descriptor to an array.</td>
</tr>
<tr>
<td>OCIDateTimeToText</td>
<td>Convert the given date to a string according to the specified format.</td>
</tr>
</tbody>
</table>

### 9.3.7 Interval Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCIIntervalAdd</td>
<td>Adds two interval values.</td>
</tr>
<tr>
<td>OCIIntervalAssign</td>
<td>Copies one interval value into another interval value.</td>
</tr>
<tr>
<td>OCIIntervalCompare</td>
<td>Compares two interval values.</td>
</tr>
<tr>
<td>OCIIntervalGetDaySecond</td>
<td>Extracts days, hours, minutes, seconds and fractional seconds from an interval.</td>
</tr>
<tr>
<td>OCIIntervalGetYearMonth</td>
<td>Extracts year and month values from an interval.</td>
</tr>
<tr>
<td>OCIIntervalSetYearMonth</td>
<td>Modifies year and month values in an interval.</td>
</tr>
<tr>
<td>OCIIntervalDivide</td>
<td>Implements division of OCIInterval values by OCINumber values.</td>
</tr>
<tr>
<td>OCIIntervalMultiply</td>
<td>Implements multiplication of OCIInterval values by OCINumber values.</td>
</tr>
<tr>
<td>OCIIntervalSubtract</td>
<td>Subtracts one interval value from another interval value.</td>
</tr>
<tr>
<td>OCIIntervalToText</td>
<td>Extrapolates a character string from an interval.</td>
</tr>
<tr>
<td>OCIIntervalCheck</td>
<td>Verifies the validity of an interval value.</td>
</tr>
<tr>
<td>OCIIntervalToNumber</td>
<td>Converts an OCIInterval value into a OCINumber value.</td>
</tr>
<tr>
<td>OCIIntervalFromNumber</td>
<td>Converts a OCINumber value into an OCIInterval value.</td>
</tr>
<tr>
<td>OCIDateTimeIntervalAdd</td>
<td>Adds an OCIInterval value to an OCIDatetime value, resulting in an OCIDatetime value.</td>
</tr>
<tr>
<td>OCIDateTimeIntervalSub</td>
<td>Subtracts an OCIInterval value from an OCIDatetime value, resulting in an OCIDatetime value.</td>
</tr>
<tr>
<td>OCIIntervalFromText</td>
<td>Converts a text string into an interval.</td>
</tr>
<tr>
<td>OCIIntervalFromTZ</td>
<td>Converts a time zone specification into an interval.</td>
</tr>
</tbody>
</table>

### 9.3.8 Number Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCINumberAbs</td>
<td>Compute the absolute value.</td>
</tr>
<tr>
<td>OCINumberAdd</td>
<td>Adds NUMBERs.</td>
</tr>
<tr>
<td>OCINumberArcCos</td>
<td>Compute the arc cosine.</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------</td>
<td>-------------------------------------------------------</td>
</tr>
<tr>
<td>OCINumberArcSin</td>
<td>Compute the arc sine.</td>
</tr>
<tr>
<td>OCINumberArcTan</td>
<td>Compute the arc tangent.</td>
</tr>
<tr>
<td>OCINumberArcTan2</td>
<td>Compute the arc tangent of two NUMBERs.</td>
</tr>
<tr>
<td>OCINumberAssign</td>
<td>Assign one NUMBER to another.</td>
</tr>
<tr>
<td>OCINumberCeil</td>
<td>Compute the ceiling of NUMBER.</td>
</tr>
<tr>
<td>OCINumberCmp</td>
<td>Compare NUMBERS.</td>
</tr>
<tr>
<td>OCINumberCos</td>
<td>Compute the cosine.</td>
</tr>
<tr>
<td>OCINumberDec</td>
<td>Decrement a NUMBER.</td>
</tr>
<tr>
<td>OCINumberDiv</td>
<td>Divide two NUMBERS.</td>
</tr>
<tr>
<td>OCINumberExp</td>
<td>Raise e to the specified NUMBER power.</td>
</tr>
<tr>
<td>OCINumberFloor</td>
<td>Compute the floor of a NUMBER.</td>
</tr>
<tr>
<td>OCINumberFromInt</td>
<td>Convert an integer to an Oracle NUMBER.</td>
</tr>
<tr>
<td>OCINumberFromReal</td>
<td>Convert a real to an Oracle NUMBER.</td>
</tr>
<tr>
<td>OCINumberFromText</td>
<td>Convert a string to an Oracle NUMBER.</td>
</tr>
<tr>
<td>OCINumberHypCos</td>
<td>Compute the hyperbolic cosine.</td>
</tr>
<tr>
<td>OCINumberHypSin</td>
<td>Compute the hyperbolic sine.</td>
</tr>
<tr>
<td>OCINumberHypTan</td>
<td>Compute the hyperbolic tangent.</td>
</tr>
<tr>
<td>OCINumberInc</td>
<td>Increments a NUMBER.</td>
</tr>
<tr>
<td>OCINumberIntPower</td>
<td>Raise a given base to an integer power.</td>
</tr>
<tr>
<td>OCINumberIsInt</td>
<td>Test if a NUMBER is an integer.</td>
</tr>
<tr>
<td>OCINumberIsZero</td>
<td>Test if a NUMBER is zero.</td>
</tr>
<tr>
<td>OCINumberLn</td>
<td>Compute the natural logarithm.</td>
</tr>
<tr>
<td>OCINumberLog</td>
<td>Compute the logarithm to an arbitrary base.</td>
</tr>
<tr>
<td>OCINumberMod</td>
<td>Modulo division.</td>
</tr>
<tr>
<td>OCINumberMul</td>
<td>Multiply NUMBERS.</td>
</tr>
<tr>
<td>OCINumberNeg</td>
<td>Negate a NUMBER.</td>
</tr>
<tr>
<td>OCINumberPower</td>
<td>Exponentiation to base e.</td>
</tr>
<tr>
<td>OCINumberPrec</td>
<td>Round a NUMBER to a specified number of decimal places.</td>
</tr>
<tr>
<td>OCINumberRound</td>
<td>Round a NUMBER to a specified decimal place.</td>
</tr>
<tr>
<td>OCINumberSetPi</td>
<td>Initialize a NUMBER to Pi.</td>
</tr>
<tr>
<td>OCINumberSetZero</td>
<td>Initialize a NUMBER to zero.</td>
</tr>
<tr>
<td>OCINumberShift</td>
<td>Multiply by 10, shifting specified number of decimal places.</td>
</tr>
<tr>
<td>OCINumberSign</td>
<td>Obtain the sign of a NUMBER.</td>
</tr>
<tr>
<td>OCINumberSin</td>
<td>Compute the sine.</td>
</tr>
<tr>
<td>OCINumberSqrt</td>
<td>Compute the square root of a NUMBER.</td>
</tr>
<tr>
<td>OCINumberSub</td>
<td>Subtract NUMBERS.</td>
</tr>
<tr>
<td>OCINumberTan</td>
<td>Compute the tangent.</td>
</tr>
<tr>
<td>OCINumberToInt</td>
<td>Convert a NUMBER to an integer.</td>
</tr>
<tr>
<td>OCINumberToReal</td>
<td>Convert a NUMBER to a real.</td>
</tr>
<tr>
<td>OCINumberToRealArray</td>
<td>Convert an array of NUMBER to a real array.</td>
</tr>
<tr>
<td>OCINumberToText</td>
<td>Converts a NUMBER to a string.</td>
</tr>
<tr>
<td>OCINumberTrunc</td>
<td>Truncate a NUMBER at a specified decimal place.</td>
</tr>
</tbody>
</table>
### 9.3.9 String Functions

Table 9-9-9 String Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCIStringAllocSize</td>
<td>Get allocated size of string memory in bytes.</td>
</tr>
<tr>
<td>OCIStringAssign</td>
<td>Assign string to a string.</td>
</tr>
<tr>
<td>OCIStringAssignText</td>
<td>Assign text string to a string.</td>
</tr>
<tr>
<td>OCIStringPtr</td>
<td>Get string pointer.</td>
</tr>
<tr>
<td>OCIStringResize</td>
<td>Resize string memory.</td>
</tr>
<tr>
<td>OCIStringLength</td>
<td>Get string size.</td>
</tr>
</tbody>
</table>

### 9.3.10 Cartridge Services and File I/O Interface Functions

Table 9-9-10 Cartridge Services and File I/O Interface Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCIFileClose</td>
<td>Close an open file.</td>
</tr>
<tr>
<td>OCIFileExists</td>
<td>Test to see if the file exists.</td>
</tr>
<tr>
<td>OCIFileFlush</td>
<td>Write buffered data to a file.</td>
</tr>
<tr>
<td>OCIFileGetSize</td>
<td>Get the length of a file.</td>
</tr>
<tr>
<td>OCIFilenInit</td>
<td>Initialize the OCIFilen package.</td>
</tr>
<tr>
<td>OCIFilenOpen</td>
<td>Open a file.</td>
</tr>
<tr>
<td>OCIFilenRead</td>
<td>Read from a file into a buffer.</td>
</tr>
<tr>
<td>OCIFilenSeek</td>
<td>Change the current position in a file.</td>
</tr>
<tr>
<td>OCIFilenTerm</td>
<td>Terminate the OCIFilen package.</td>
</tr>
<tr>
<td>OCIFilenWrite</td>
<td>Write buflen bytes into the file.</td>
</tr>
<tr>
<td>SQLT_STR</td>
<td>Zero-terminated string</td>
</tr>
<tr>
<td>SQLT_TIMESTAMP</td>
<td>Timestamp</td>
</tr>
<tr>
<td>SQLT_TIMESTAMP_TZ</td>
<td>Timestamp with time zone</td>
</tr>
<tr>
<td>SQLT_TIMESTAMP_LTZ</td>
<td>Timestamp with local time zone</td>
</tr>
<tr>
<td>SQLT_UINT</td>
<td>Unsigned integer</td>
</tr>
<tr>
<td>SQLT_VBI</td>
<td>VCS format binary</td>
</tr>
<tr>
<td>SQLT_VCS</td>
<td>Variable character</td>
</tr>
<tr>
<td>SQLT_VNU</td>
<td>Number with preceding length byte</td>
</tr>
<tr>
<td>SQLT_VST</td>
<td>OCI string type</td>
</tr>
</tbody>
</table>

### 9.3.11 Miscellaneous Functions

Table 9-9-11 Miscellaneous Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCIClientVersion</td>
<td>Return client library version.</td>
</tr>
<tr>
<td>OCIErrorGet</td>
<td>Return error message.</td>
</tr>
<tr>
<td>OCIPasswordChange</td>
<td>Change password.</td>
</tr>
<tr>
<td>OCIPing</td>
<td>Confirm that the connection and server are active.</td>
</tr>
<tr>
<td>OCIServerVersion</td>
<td>Get the Oracle version string.</td>
</tr>
</tbody>
</table>
### 9.3.12 Supported Data Types

**Table 9-9-12 Supported Data Types**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANSI_DATE</td>
<td>ANSI date</td>
</tr>
<tr>
<td>SQLT_AFC</td>
<td>ANSI fixed character</td>
</tr>
<tr>
<td>SQLT_AVC</td>
<td>ANSI variable character</td>
</tr>
<tr>
<td>SQLT_BDOUBLE</td>
<td>Binary double</td>
</tr>
<tr>
<td>SQLT_BIN</td>
<td>Binary data</td>
</tr>
<tr>
<td>SQLT_BFLOAT</td>
<td>Binary float</td>
</tr>
<tr>
<td>SQLT_CHR</td>
<td>Character string</td>
</tr>
<tr>
<td>SQLT_DAT</td>
<td>Oracle date</td>
</tr>
<tr>
<td>SQLT_DATE</td>
<td>ANSI date</td>
</tr>
<tr>
<td>SQLT_FLT</td>
<td>Float</td>
</tr>
<tr>
<td>SQLT_INT</td>
<td>Integer</td>
</tr>
<tr>
<td>SQLT_LBI</td>
<td>Long binary</td>
</tr>
<tr>
<td>SQLT_LNG</td>
<td>Long</td>
</tr>
<tr>
<td>SQLT_LVB</td>
<td>Longer long binary</td>
</tr>
<tr>
<td>SQLT_LVC</td>
<td>Longer longs (character)</td>
</tr>
<tr>
<td>SQLT_NUM</td>
<td>Oracle numeric</td>
</tr>
<tr>
<td>SQLT_ODT</td>
<td>OCI date type</td>
</tr>
</tbody>
</table>
10 Oracle Catalog Views

The Oracle Catalog Views provide information on Oracle compatible database objects in a manner compatible with the Oracle data dictionary views found in an Oracle database.

10.1 ALL_ALL_TABLES

The ALL_ALL_TABLES view provides information about the tables accessible by the current user.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>owner</td>
<td>TEXT</td>
<td>User name of the table’s owner.</td>
</tr>
<tr>
<td>schema_name</td>
<td>TEXT</td>
<td>Name of the schema in which the table belongs.</td>
</tr>
<tr>
<td>table_name</td>
<td>TEXT</td>
<td>The name of the table.</td>
</tr>
<tr>
<td>tablespace_name</td>
<td>TEXT</td>
<td>Name of the tablespace in which the table resides if other than the default tablespace.</td>
</tr>
<tr>
<td>temporary</td>
<td>TEXT</td>
<td>Y if the table is temporary; N if the table is permanent.</td>
</tr>
<tr>
<td>status</td>
<td>VARCHAR2</td>
<td>Included for compatibility only; always set to VALID.</td>
</tr>
</tbody>
</table>

10.2 ALL_CONS_COLUMNS

The ALL_CONS_COLUMNS view provides information about the columns specified in constraints placed on tables accessible by the current user.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>owner</td>
<td>TEXT</td>
<td>User name of the constraint’s owner.</td>
</tr>
<tr>
<td>schema_name</td>
<td>TEXT</td>
<td>Name of the schema in which the constraint belongs.</td>
</tr>
<tr>
<td>constraint_name</td>
<td>TEXT</td>
<td>The name of the constraint.</td>
</tr>
<tr>
<td>table_name</td>
<td>TEXT</td>
<td>The name of the table to which the constraint belongs.</td>
</tr>
<tr>
<td>column_name</td>
<td>TEXT</td>
<td>The name of the column referenced in the constraint.</td>
</tr>
<tr>
<td>position</td>
<td>SMALLINT</td>
<td>The position of the column within the object definition.</td>
</tr>
<tr>
<td>constraint_def</td>
<td>TEXT</td>
<td>The definition of the constraint.</td>
</tr>
</tbody>
</table>

10.3 ALL_CONSTRAINTS

The ALL_CONSTRAINTS view provides information about the constraints placed on tables accessible by the current user.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>owner</td>
<td>TEXT</td>
<td>User name of the constraint’s owner.</td>
</tr>
<tr>
<td>schema_name</td>
<td>TEXT</td>
<td>Name of the schema in which the constraint belongs.</td>
</tr>
<tr>
<td>constraint_name</td>
<td>TEXT</td>
<td>The name of the constraint.</td>
</tr>
<tr>
<td>constraint_type</td>
<td>TEXT</td>
<td>The constraint type. Possible values are: C – check constraint</td>
</tr>
</tbody>
</table>
### 10.4 ALL_DB_LINKS

The **ALL_DB_LINKS** view provides information about the database links accessible by the current user.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>owner</td>
<td>TEXT</td>
<td>User name of the database link’s owner.</td>
</tr>
<tr>
<td>schema_name</td>
<td>TEXT</td>
<td>Name of the schema in which the link belongs.</td>
</tr>
<tr>
<td>db_link</td>
<td>TEXT</td>
<td>The name of the database link.</td>
</tr>
<tr>
<td>type</td>
<td>CHARACTER VARYING</td>
<td>Type of remote server. Value will be either REDWOOD or EDB.</td>
</tr>
<tr>
<td>username</td>
<td>TEXT</td>
<td>User name of the user logging in.</td>
</tr>
<tr>
<td>host</td>
<td>TEXT</td>
<td>Name or IP address of the remote server.</td>
</tr>
</tbody>
</table>

### 10.5 ALL_IND_COLUMNS

The **ALL_IND_COLUMNS** view provides information about columns included in indexes on the tables accessible by the current user.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>index_owner</td>
<td>TEXT</td>
<td>User name of the index’s owner.</td>
</tr>
<tr>
<td>schema_name</td>
<td>TEXT</td>
<td>Name of the schema in which the index belongs.</td>
</tr>
<tr>
<td>index_name</td>
<td>TEXT</td>
<td>The name of the index.</td>
</tr>
<tr>
<td>table_owner</td>
<td>TEXT</td>
<td>User name of the table owner.</td>
</tr>
</tbody>
</table>
### 10.6 **ALL_INDEXES**

The **ALL_INDEXES** view provides information about the indexes on tables that may be accessed by the current user.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>owner</td>
<td>TEXT</td>
<td>User name of the index’s owner.</td>
</tr>
<tr>
<td>index_schema</td>
<td>TEXT</td>
<td>Name of the schema in which the index belongs.</td>
</tr>
<tr>
<td>index_name</td>
<td>TEXT</td>
<td>The name of the index.</td>
</tr>
<tr>
<td>index_type</td>
<td>TEXT</td>
<td>The index type is always BTREE. Included for compatibility only.</td>
</tr>
<tr>
<td>table_owner</td>
<td>TEXT</td>
<td>User name of the owner of the indexed table.</td>
</tr>
<tr>
<td>table_name</td>
<td>TEXT</td>
<td>The name of the indexed table.</td>
</tr>
<tr>
<td>table_type</td>
<td>TEXT</td>
<td>Included for compatibility only. Always set to TABLE.</td>
</tr>
<tr>
<td>uniqueness</td>
<td>TEXT</td>
<td>Indicates if the index is UNIQUE or NONUNIQUE.</td>
</tr>
<tr>
<td>compression</td>
<td>CHAR(1)</td>
<td>Always set to N (not compressed). Included for compatibility only.</td>
</tr>
<tr>
<td>tablespace_name</td>
<td>TEXT</td>
<td>Name of the tablespace in which the table resides if other than the default tablespace.</td>
</tr>
<tr>
<td>logging</td>
<td>TEXT</td>
<td>Always set to LOGGING. Included for compatibility only.</td>
</tr>
<tr>
<td>status</td>
<td>TEXT</td>
<td>Included for compatibility only; always set to VALID.</td>
</tr>
<tr>
<td>partitioned</td>
<td>CHAR(3)</td>
<td>Indicates that the index is partitioned. Currently, always set to NO.</td>
</tr>
<tr>
<td>temporary</td>
<td>CHAR(1)</td>
<td>Indicates that an index is on a temporary table. Always set to N; included for compatibility only.</td>
</tr>
<tr>
<td>secondary</td>
<td>CHAR(1)</td>
<td>Included for compatibility only. Always set to N.</td>
</tr>
<tr>
<td>join_index</td>
<td>CHAR(3)</td>
<td>Included for compatibility only. Always set to NO.</td>
</tr>
<tr>
<td>dropped</td>
<td>CHAR(3)</td>
<td>Included for compatibility only. Always set to NO.</td>
</tr>
</tbody>
</table>
10.7 **ALL_OBJECTS**

The **ALL_OBJECTS** view provides information on the following database objects – tables, indexes, sequences, views, triggers, functions, procedures, packages, and package bodies. Note that only SPL triggers, functions, procedures, packages, and package bodies are shown – PL/pgSQL triggers and functions do not appear in the **ALL_OBJECTS** view.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>owner</td>
<td>VARCHAR2</td>
<td>User name of the object’s owner.</td>
</tr>
<tr>
<td>schemaname</td>
<td>VARCHAR2</td>
<td>Name of the schema in which the object belongs.</td>
</tr>
<tr>
<td>object_name</td>
<td>VARCHAR2</td>
<td>Name of the object.</td>
</tr>
<tr>
<td>object_type</td>
<td>VARCHAR2</td>
<td>Type of the object – possible values are: INDEX, FUNCTION, PACKAGE, PACKAGE BODY, PROCEDURE, SEQUENCE, SYNONYM, TABLE, TRIGGER, and VIEW.</td>
</tr>
<tr>
<td>status</td>
<td>VARCHAR2</td>
<td>Whether or not the state of the object is valid. Currently, Included for compatibility only; always set to VALID.</td>
</tr>
</tbody>
</table>

10.8 **ALL_POLICIES**

The **ALL_POLICIES** view provides information on all policies in the database. This view is accessible only to superusers.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object_owner</td>
<td>NAME</td>
<td>Name of the schema in which the object belongs.</td>
</tr>
<tr>
<td>object_name</td>
<td>NAME</td>
<td>Name of the object on which the policy applies.</td>
</tr>
<tr>
<td>policy_group</td>
<td>TEXT</td>
<td>Name of the policy group. Included for compatibility only; always set to an empty string.</td>
</tr>
<tr>
<td>policy_name</td>
<td>NAME</td>
<td>Name of the policy.</td>
</tr>
<tr>
<td>pf_owner</td>
<td>NAME</td>
<td>Name of the schema containing the policy function, or the schema containing the package that contains the policy function.</td>
</tr>
<tr>
<td>package</td>
<td>NAME</td>
<td>Name of the package containing the policy function if the function belongs to a package.</td>
</tr>
<tr>
<td>function</td>
<td>NAME</td>
<td>Name of the policy function.</td>
</tr>
<tr>
<td>sel</td>
<td>TEXT</td>
<td>Whether or not the policy applies to SELECT commands. Possible values are YES or NO.</td>
</tr>
<tr>
<td>ins</td>
<td>TEXT</td>
<td>Whether or not the policy applies to INSERT commands. Possible values are YES or NO.</td>
</tr>
<tr>
<td>upd</td>
<td>TEXT</td>
<td>Whether or not the policy applies to UPDATE commands. Possible values are YES or NO.</td>
</tr>
<tr>
<td>del</td>
<td>TEXT</td>
<td>Whether or not the policy applies to DELETE commands. Possible values are YES or NO.</td>
</tr>
<tr>
<td>idx</td>
<td>TEXT</td>
<td>Whether or not the policy applies to index maintenance. Possible values are YES or NO.</td>
</tr>
<tr>
<td>chk_option</td>
<td>TEXT</td>
<td>Whether or not the check option is in force for INSERT and UPDATE commands. Possible values are YES or NO.</td>
</tr>
<tr>
<td>enable</td>
<td>TEXT</td>
<td>Whether or not the policy is enabled on the object. Possible values are YES or NO.</td>
</tr>
<tr>
<td>Name</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------</td>
<td>---------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>static_policy</td>
<td>TEXT</td>
<td>Whether or not the policy is static. Included for compatibility only; always set to NO.</td>
</tr>
<tr>
<td>policy_type</td>
<td>TEXT</td>
<td>Policy type. Included for compatibility only; always set to UNKNOWN.</td>
</tr>
<tr>
<td>long_predicate</td>
<td>TEXT</td>
<td>Whether or not predicates longer than 4000 bytes can be returned by the policy function. Included for compatibility only; always set to YES.</td>
</tr>
</tbody>
</table>

### 10.9 ALL SEQUENCES

The **ALL SEQUENCES** view provides information about all user-defined sequences on which the user has **SELECT**, or **UPDATE** privileges.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequence_owner</td>
<td>TEXT</td>
<td>User name of the sequence's owner.</td>
</tr>
<tr>
<td>sequence_name</td>
<td>TEXT</td>
<td>Name of the sequence.</td>
</tr>
<tr>
<td>min_value</td>
<td>NUMERIC</td>
<td>The lowest value that the server will assign to the sequence.</td>
</tr>
<tr>
<td>max_value</td>
<td>NUMERIC</td>
<td>The highest value that the server will assign to the sequence.</td>
</tr>
<tr>
<td>increment_by</td>
<td>NUMERIC</td>
<td>The value added to the current sequence number to create the next sequent number.</td>
</tr>
<tr>
<td>cycle_flag</td>
<td>VARCHAR</td>
<td>Specifies if the sequence should wrap when it reaches <strong>min_value</strong> or <strong>max_value</strong>.</td>
</tr>
<tr>
<td>order_flag</td>
<td>VARCHAR</td>
<td>Included for Oracle compatibility; will always return <strong>Y</strong>.</td>
</tr>
<tr>
<td>cache_size</td>
<td>NUMERIC</td>
<td>The number of pre-allocated sequence numbers stored in memory.</td>
</tr>
<tr>
<td>last_number</td>
<td>NUMERIC</td>
<td>The value of the last sequence number saved to disk.</td>
</tr>
</tbody>
</table>

### 10.10 ALL SOURCE

The **ALL SOURCE** view provides a source code listing of the following program types – functions, procedures, triggers, package specifications, and package bodies.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>owner</td>
<td>VARCHAR2</td>
<td>User name of the program’s owner.</td>
</tr>
<tr>
<td>schemaname</td>
<td>VARCHAR2</td>
<td>Name of the schema in which the program belongs.</td>
</tr>
<tr>
<td>name</td>
<td>VARCHAR2</td>
<td>Name of the program.</td>
</tr>
<tr>
<td>type</td>
<td>VARCHAR2</td>
<td>Type of program – possible values are: FUNCTION, PACKAGE, PACKAGE BODY, PROCEDURE, and TRIGGER.</td>
</tr>
<tr>
<td>line</td>
<td>INTEGER</td>
<td>Source code line number relative to a given program.</td>
</tr>
<tr>
<td>text</td>
<td>VARCHAR2</td>
<td>Line of source code text.</td>
</tr>
</tbody>
</table>
10.11 ALL_SYNONYMS

The `ALL_SYNONYMS` view provides information on all synonyms that may be referenced by the current user.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>owner</td>
<td>TEXT</td>
<td>User name of the synonym’s owner.</td>
</tr>
<tr>
<td>synonym_name</td>
<td>TEXT</td>
<td>Name of the synonym.</td>
</tr>
<tr>
<td>object_owner</td>
<td>TEXT</td>
<td>User name of the object’s owner.</td>
</tr>
<tr>
<td>object_name</td>
<td>TEXT</td>
<td>The name of the object that the synonym refers to.</td>
</tr>
<tr>
<td>status</td>
<td>CHARACTER</td>
<td>Deprecated in 8.4.</td>
</tr>
<tr>
<td>db_link</td>
<td>TEXT</td>
<td>The name of any associated database link.</td>
</tr>
</tbody>
</table>

10.12 ALL_TAB_COLUMNS

The `ALL_TAB_COLUMNS` view provides information on all columns in all user-defined tables.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>owner</td>
<td>VARCHAR2</td>
<td>User name of the table’s owner.</td>
</tr>
<tr>
<td>schemaname</td>
<td>VARCHAR2</td>
<td>Name of the schema in which the table belongs.</td>
</tr>
<tr>
<td>table_name</td>
<td>VARCHAR2</td>
<td>Name of the table.</td>
</tr>
<tr>
<td>column_name</td>
<td>VARCHAR2</td>
<td>Name of the column.</td>
</tr>
<tr>
<td>data_type</td>
<td>VARCHAR2</td>
<td>Data type of the column.</td>
</tr>
<tr>
<td>data_length</td>
<td>INTEGER</td>
<td>Length of text columns.</td>
</tr>
<tr>
<td>data_precision</td>
<td>INTEGER</td>
<td>Precision (number of digits) for NUMBER columns.</td>
</tr>
<tr>
<td>data_scale</td>
<td>INTEGER</td>
<td>Scale of NUMBER columns.</td>
</tr>
<tr>
<td>column_id</td>
<td>INTEGER</td>
<td>Relative position of the column within the table.</td>
</tr>
<tr>
<td>nullable</td>
<td>CHARACTER</td>
<td>Whether or not the column is nullable – possible values are: Y (column is nullable); N (column does not allow null).</td>
</tr>
<tr>
<td>data_default</td>
<td>VARCHAR2</td>
<td>Default value assigned to the column.</td>
</tr>
</tbody>
</table>

10.13 ALL_TABLES

The `ALL_TABLES` view provides information on all user-defined tables.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>owner</td>
<td>VARCHAR2</td>
<td>User name of the table’s owner.</td>
</tr>
<tr>
<td>schemaname</td>
<td>VARCHAR2</td>
<td>Name of the schema in which the table belongs.</td>
</tr>
<tr>
<td>table_name</td>
<td>VARCHAR2</td>
<td>Name of the table.</td>
</tr>
<tr>
<td>table_space</td>
<td>VARCHAR2</td>
<td>Name of the tablespace in which the table resides if other than the default tablespace.</td>
</tr>
<tr>
<td>status</td>
<td>VARCHAR2</td>
<td>Whether or not the state of the table is valid. Currently, Included for compatibility only; always set to VALID.</td>
</tr>
</tbody>
</table>
10.14 **ALL_TRIGGERS**

The **ALL_TRIGGERS** view provides information about the triggers on tables that may be accessed by the current user.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>owner</td>
<td>TEXT</td>
<td>User name of the trigger’s owner.</td>
</tr>
<tr>
<td>trigger_name</td>
<td>TEXT</td>
<td>The name of the trigger.</td>
</tr>
<tr>
<td>trigger_type</td>
<td>TEXT</td>
<td>The type of the trigger. Possible values are:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BEFORE ROW</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BEFORE STATEMENT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AFTER ROW</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AFTER STATEMENT</td>
</tr>
<tr>
<td>triggering_event</td>
<td>TEXT</td>
<td>The event that fires the trigger.</td>
</tr>
<tr>
<td>table_owner</td>
<td>TEXT</td>
<td>The user name of the owner of the table on which the trigger is defined.</td>
</tr>
<tr>
<td>base_object_type</td>
<td>TEXT</td>
<td>Included for compatibility only. Value will always be TABLE.</td>
</tr>
<tr>
<td>table_name</td>
<td>TEXT</td>
<td>The name of the table on which the trigger is defined.</td>
</tr>
<tr>
<td>referencing_name</td>
<td>TEXT</td>
<td>Included for compatibility only. Value will always be</td>
</tr>
<tr>
<td></td>
<td></td>
<td>REFERENCING NEW AS NEW OLD AS OLD.</td>
</tr>
<tr>
<td>status</td>
<td>TEXT</td>
<td>Status indicates if the trigger is enabled (VALID) or disabled (NOTVALID).</td>
</tr>
<tr>
<td>description</td>
<td>TEXT</td>
<td>Included for compatibility only. Value will always be</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SEE TRIGGER BODY FOR TEXT.</td>
</tr>
<tr>
<td>trigger_body</td>
<td>TEXT</td>
<td>The body of the trigger.</td>
</tr>
<tr>
<td>action_statement</td>
<td>TEXT</td>
<td>The SQL command that executes when the trigger fires.</td>
</tr>
</tbody>
</table>

10.15 **ALL_TYPES**

The **ALL_TYPES** view provides information about the object types available to the current user.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>owner</td>
<td>text</td>
<td>The owner of the object type.</td>
</tr>
<tr>
<td>schema_name</td>
<td>text</td>
<td>The name of the schema in which the type is defined.</td>
</tr>
<tr>
<td>type_name</td>
<td>text</td>
<td>The name of the type.</td>
</tr>
<tr>
<td>type_oid</td>
<td>oid</td>
<td>The object identifier (OID) of the type.</td>
</tr>
<tr>
<td>typecode</td>
<td>text</td>
<td>The typecode of the type. Possible values are:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>OBJECT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>COLLECTION</td>
</tr>
<tr>
<td></td>
<td></td>
<td>OTHER</td>
</tr>
<tr>
<td>attributes</td>
<td>integer</td>
<td>The number of attributes in the type.</td>
</tr>
</tbody>
</table>
10.16 **ALL_USERS**

The ALL_USERS view provides information on all user names.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>username</td>
<td>VARCHAR2</td>
<td>Name of the user.</td>
</tr>
<tr>
<td>user_id</td>
<td>VARCHAR2</td>
<td>Numeric user id assigned to the user.</td>
</tr>
<tr>
<td>created</td>
<td>TIMESTAMP</td>
<td>Always NULL; Included for compatibility only.</td>
</tr>
</tbody>
</table>

10.17 **ALL_VIEW_COLUMNS**

The ALL_VIEW_COLUMNS view provides information on all columns in all user-defined views.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Owner</td>
<td>VARCHAR2</td>
<td>User name of the view’s owner.</td>
</tr>
<tr>
<td>schemaname</td>
<td>VARCHAR2</td>
<td>Name of the schema in which the view belongs.</td>
</tr>
<tr>
<td>view_name</td>
<td>VARCHAR2</td>
<td>Name of the view.</td>
</tr>
<tr>
<td>column_name</td>
<td>VARCHAR2</td>
<td>Name of the column.</td>
</tr>
<tr>
<td>data_type</td>
<td>VARCHAR2</td>
<td>Data type of the column.</td>
</tr>
<tr>
<td>data_length</td>
<td>INTEGER</td>
<td>Length of text columns.</td>
</tr>
<tr>
<td>data_precision</td>
<td>INTEGER</td>
<td>Precision (number of digits) for NUMBER columns.</td>
</tr>
<tr>
<td>data_scale</td>
<td>INTEGER</td>
<td>Scale of NUMBER columns.</td>
</tr>
<tr>
<td>column_id</td>
<td>INTEGER</td>
<td>Relative position of the column within the view.</td>
</tr>
<tr>
<td>nullable</td>
<td>CHARACTER</td>
<td>Whether or not the column is nullable – possible values are: Y – column is nullable; N – column does not allow null.</td>
</tr>
<tr>
<td>data_default</td>
<td>VARCHAR2</td>
<td>Default value assigned to the column.</td>
</tr>
</tbody>
</table>

10.18 **ALL_VIEWS**

The ALL_VIEWS view provides information on all user-defined views.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Owner</td>
<td>VARCHAR2</td>
<td>User name of the view’s owner.</td>
</tr>
<tr>
<td>schemaname</td>
<td>VARCHAR2</td>
<td>Name of the schema in which the view belongs.</td>
</tr>
<tr>
<td>view_name</td>
<td>VARCHAR2</td>
<td>Name of the view.</td>
</tr>
<tr>
<td>status</td>
<td>VARCHAR2</td>
<td>Whether or not the state of the view is valid. Currently, always set to VALID.</td>
</tr>
</tbody>
</table>
10.19 DBA_ALL_TABLES

The **DBA_ALL_TABLES** view provides information about all tables in the database.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>owner</td>
<td>VARCHAR2</td>
<td>User name of the table’s owner.</td>
</tr>
<tr>
<td>schema_name</td>
<td>VARCHAR2</td>
<td>Name of the schema in which the table belongs.</td>
</tr>
<tr>
<td>table_name</td>
<td>VARCHAR2</td>
<td>Name of the table.</td>
</tr>
<tr>
<td>tablespace_name</td>
<td>VARCHAR2</td>
<td>Name of the tablespace in which the table resides if other than the default tablespace.</td>
</tr>
<tr>
<td>status</td>
<td>VARCHAR2</td>
<td>Included for compatibility only; always set to VALID.</td>
</tr>
<tr>
<td>temporary</td>
<td>TEXT</td>
<td>Y if the table is temporary; N if the table is permanent.</td>
</tr>
</tbody>
</table>

10.20 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.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>owner</td>
<td>TEXT</td>
<td>User name of the constraint’s owner.</td>
</tr>
<tr>
<td>schema_name</td>
<td>TEXT</td>
<td>Name of the schema in which the constraint belongs.</td>
</tr>
<tr>
<td>constraint_name</td>
<td>TEXT</td>
<td>The name of the constraint.</td>
</tr>
<tr>
<td>table_name</td>
<td>TEXT</td>
<td>The name of the table to which the constraint belongs.</td>
</tr>
<tr>
<td>column_name</td>
<td>TEXT</td>
<td>The name of the column referenced in the constraint.</td>
</tr>
<tr>
<td>position</td>
<td>SMALLINT</td>
<td>The position of the column within the object definition.</td>
</tr>
<tr>
<td>constraint_def</td>
<td>TEXT</td>
<td>The definition of the constraint.</td>
</tr>
</tbody>
</table>

10.21 DBA_CONSTRAINTS

The **DBA_CONSTRAINTS** view provides information about all constraints on tables in the database.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>owner</td>
<td>TEXT</td>
<td>User name of the constraint’s owner.</td>
</tr>
<tr>
<td>schema_name</td>
<td>TEXT</td>
<td>Name of the schema in which the constraint belongs.</td>
</tr>
<tr>
<td>constraint_name</td>
<td>TEXT</td>
<td>The name of the constraint.</td>
</tr>
<tr>
<td>constraint_type</td>
<td>TEXT</td>
<td>The constraint type. Possible values are:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>C – check constraint</td>
</tr>
<tr>
<td></td>
<td></td>
<td>F – foreign key constraint</td>
</tr>
<tr>
<td></td>
<td></td>
<td>P – primary key constraint</td>
</tr>
<tr>
<td></td>
<td></td>
<td>U – unique key constraint</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R – referential integrity constraint</td>
</tr>
<tr>
<td></td>
<td></td>
<td>V – constraint on a view</td>
</tr>
<tr>
<td></td>
<td></td>
<td>O – with read-only, on a view</td>
</tr>
<tr>
<td>table_name</td>
<td>TEXT</td>
<td>Name of the table to which the constraint belongs.</td>
</tr>
<tr>
<td>Name</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------</td>
<td>-----------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>search_condition</td>
<td>TEXT</td>
<td>Search condition that applies to a check constraint.</td>
</tr>
<tr>
<td>r_owner</td>
<td>TEXT</td>
<td>Owner of a table referenced by a referential constraint.</td>
</tr>
<tr>
<td>r_constraint_name</td>
<td>TEXT</td>
<td>Name of the constraint definition for a referenced table.</td>
</tr>
<tr>
<td>delete_rule</td>
<td>TEXT</td>
<td>The delete rule for a referential constraint. Possible values are:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>C – cascade</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R - restrict</td>
</tr>
<tr>
<td></td>
<td></td>
<td>N – no action</td>
</tr>
<tr>
<td>deferrable</td>
<td>BOOLEAN</td>
<td>Specified if the constraint is deferrable (Y or N).</td>
</tr>
<tr>
<td>deferred</td>
<td>BOOLEAN</td>
<td>Specifies if the constraint has been deferred (Y or N).</td>
</tr>
<tr>
<td>index_owner</td>
<td>TEXT</td>
<td>User name of the index owner.</td>
</tr>
<tr>
<td>index_name</td>
<td>TEXT</td>
<td>The name of the index.</td>
</tr>
<tr>
<td>constraint_def</td>
<td>TEXT</td>
<td>The definition of the constraint.</td>
</tr>
</tbody>
</table>

### 10.22 DBA_DB_LINKS

The **DBA_DB_LINKS** view provides information about all database links in the database.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>owner</td>
<td>TEXT</td>
<td>User name of the database link’s owner.</td>
</tr>
<tr>
<td>schema_name</td>
<td>TEXT</td>
<td>Name of the schema in which the link belongs.</td>
</tr>
<tr>
<td>db_link</td>
<td>TEXT</td>
<td>The name of the database link.</td>
</tr>
<tr>
<td>type</td>
<td>CHARACTER VARYING</td>
<td>Type of remote server. Value will be either REDWOOD or EDB</td>
</tr>
<tr>
<td>username</td>
<td>TEXT</td>
<td>User name of the user logging in.</td>
</tr>
<tr>
<td>host</td>
<td>TEXT</td>
<td>Name or IP address of the remote server.</td>
</tr>
</tbody>
</table>

### 10.23 DBA_IND_COLUMNS

The **DBA_IND_COLUMNS** view provides information about all columns included in indexes, on all tables in the database.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>index_owner</td>
<td>TEXT</td>
<td>User name of the index’s owner.</td>
</tr>
<tr>
<td>schema_name</td>
<td>TEXT</td>
<td>Name of the schema in which the index belongs.</td>
</tr>
<tr>
<td>index_name</td>
<td>TEXT</td>
<td>Name of the index.</td>
</tr>
<tr>
<td>table_owner</td>
<td>TEXT</td>
<td>User name of the table’s owner.</td>
</tr>
<tr>
<td>table_name</td>
<td>TEXT</td>
<td>Name of the table in which the index belongs.</td>
</tr>
<tr>
<td>column_name</td>
<td>TEXT</td>
<td>Name of column or attribute of object column.</td>
</tr>
<tr>
<td>column_position</td>
<td>SMALLINT</td>
<td>The position of the column in the index.</td>
</tr>
<tr>
<td>column_length</td>
<td>SMALLINT</td>
<td>The length of the column (in bytes).</td>
</tr>
<tr>
<td>char_length</td>
<td>NUMERIC</td>
<td>The length of the column (in characters).</td>
</tr>
<tr>
<td>descend</td>
<td>CHAR(1)</td>
<td>Sorted order of the column on disk. Always set to Y (descending); included for compatibility only.</td>
</tr>
</tbody>
</table>
The **DBA_INDEXES** view provides information about all indexes in the database.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>owner</td>
<td>TEXT</td>
<td>User name of the index’s owner.</td>
</tr>
<tr>
<td>index_schema</td>
<td>TEXT</td>
<td>Name of the schema in which the index belongs.</td>
</tr>
<tr>
<td>index_name</td>
<td>TEXT</td>
<td>The name of the index.</td>
</tr>
<tr>
<td>index_type</td>
<td>TEXT</td>
<td>The index type is always BTREE. Included for compatibility only.</td>
</tr>
<tr>
<td>table_owner</td>
<td>TEXT</td>
<td>User name of the owner of the indexed table.</td>
</tr>
<tr>
<td>table_name</td>
<td>TEXT</td>
<td>The name of the indexed table.</td>
</tr>
<tr>
<td>table_type</td>
<td>TEXT</td>
<td>Included for compatibility only. Currently, always set to TABLE.</td>
</tr>
<tr>
<td>uniqueness</td>
<td>TEXT</td>
<td>Indicates if the index is UNIQUE or NONUNIQUE.</td>
</tr>
<tr>
<td>compression</td>
<td>CHAR(1)</td>
<td>Always set to N (not compressed). Included for compatibility only.</td>
</tr>
<tr>
<td>tablespace_name</td>
<td>TEXT</td>
<td>Name of the tablespace in which the table resides if other than the default tablespace.</td>
</tr>
<tr>
<td>logging</td>
<td>TEXT</td>
<td>Included for compatibility only. Currently always set to LOGGING.</td>
</tr>
<tr>
<td>status</td>
<td>TEXT</td>
<td>Whether or not the state of the object is valid. (VALID or INVALID).</td>
</tr>
<tr>
<td>partitioned</td>
<td>CHAR(3)</td>
<td>Indicates that the index is partitioned. Currently, always set to NO.</td>
</tr>
<tr>
<td>temporary</td>
<td>CHAR(1)</td>
<td>Indicates that an index is on a temporary table. Currently, always set to N.</td>
</tr>
<tr>
<td>secondary</td>
<td>CHAR(1)</td>
<td>Included for compatibility only. Currently always set to N.</td>
</tr>
<tr>
<td>join_index</td>
<td>CHAR(3)</td>
<td>Included for compatibility only. Currently always set to NO.</td>
</tr>
<tr>
<td>dropped</td>
<td>CHAR(3)</td>
<td>Included for compatibility only. Currently always set to NO.</td>
</tr>
</tbody>
</table>

The **DBA_JOBS** view provides information about all jobs in the database.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Job</td>
<td>INTEGER</td>
<td>The identifier of the job (Job ID).</td>
</tr>
<tr>
<td>log_user</td>
<td>TEXT</td>
<td>The name of the user that submitted the job.</td>
</tr>
<tr>
<td>priv_user</td>
<td>TEXT</td>
<td>Same as log_user. Included for compatibility only.</td>
</tr>
<tr>
<td>schema_user</td>
<td>TEXT</td>
<td>The name of the schema used to parse the job.</td>
</tr>
<tr>
<td>last_date</td>
<td>TIMESTAMP WITH TIME ZONE</td>
<td>The last date that this job executed successfully.</td>
</tr>
<tr>
<td>last_sec</td>
<td>TEXT</td>
<td>Same as last_date.</td>
</tr>
<tr>
<td>this_date</td>
<td>TIMESTAMP WITH TIME ZONE</td>
<td>The date that the job began executing.</td>
</tr>
<tr>
<td>this_sec</td>
<td>TEXT</td>
<td>Same as this_date</td>
</tr>
<tr>
<td>next_date</td>
<td>TIMESTAMP WITH TIME ZONE</td>
<td>The next date that this job will be executed.</td>
</tr>
</tbody>
</table>
### 10.26 DBA_OBJECTS

The DBA_OBJECTS view provides information about all objects in the database.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>owner</td>
<td>VARCHAR2</td>
<td>User name of the object’s owner.</td>
</tr>
<tr>
<td>schemaname</td>
<td>VARCHAR2</td>
<td>Name of the schema in which the object belongs.</td>
</tr>
<tr>
<td>object_name</td>
<td>VARCHAR2</td>
<td>Name of the object.</td>
</tr>
<tr>
<td>object_type</td>
<td>VARCHAR2</td>
<td>Type of the object – possible values are: INDEX, FUNCTION, PACKAGE, PACKAGE BODY, PROCEDURE, SEQUENCE, SYNONYM, TABLE, TRIGGER, and VIEW.</td>
</tr>
<tr>
<td>status</td>
<td>VARCHAR2</td>
<td>Included for compatibility only; always set to VALID.</td>
</tr>
<tr>
<td>temporary</td>
<td>TEXT</td>
<td>Y if the table is temporary; N if the table is permanent.</td>
</tr>
</tbody>
</table>

### 10.27 DBA_POLICIES

The DBA_POLICIES view provides information on all policies in the database. This view is accessible only to superusers.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object_owner</td>
<td>NAME</td>
<td>Name of the schema in which the object belongs.</td>
</tr>
<tr>
<td>object_name</td>
<td>NAME</td>
<td>Name of the object on which the policy applies.</td>
</tr>
<tr>
<td>policy_group</td>
<td>TEXT</td>
<td>Name of the policy group. Included for compatibility only; always set to an empty string.</td>
</tr>
<tr>
<td>policy_name</td>
<td>NAME</td>
<td>Name of the policy.</td>
</tr>
<tr>
<td>pf_owner</td>
<td>NAME</td>
<td>Name of the schema containing the policy function, or the schema containing the package that contains the policy function.</td>
</tr>
<tr>
<td>package</td>
<td>NAME</td>
<td>Name of the package containing the policy function if the function belongs to a package.</td>
</tr>
<tr>
<td>function</td>
<td>NAME</td>
<td>Name of the policy function.</td>
</tr>
</tbody>
</table>
### Database Compatibility for Oracle® Developer’s Guide

#### 10.28 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.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>grantee</td>
<td>VARCHAR2</td>
<td>User name to whom the role was granted.</td>
</tr>
<tr>
<td>granted_role</td>
<td>VARCHAR2</td>
<td>Name of the role granted to the grantee.</td>
</tr>
<tr>
<td>admin_option</td>
<td>VARCHAR2</td>
<td>YES if the role was granted with the admin option, NO otherwise.</td>
</tr>
<tr>
<td>default_role</td>
<td>VARCHAR2</td>
<td>YES if the role is automatically enabled when the grantee creates a session, NO otherwise. Based on rolinherit in pg_roles. If rolinherit is TRUE, default_role is YES. If rolinherit is FALSE, default_role is NO.</td>
</tr>
</tbody>
</table>
10.29 DBA_ROLES

The DBA_ROLES view provides information on all roles with the NOLOGIN attribute (groups).

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>role</td>
<td>VARCHAR2</td>
<td>Name of a role having the NOLOGIN attribute – i.e., a group.</td>
</tr>
<tr>
<td>password_required</td>
<td>VARCHAR2</td>
<td>Whether or not a password is required to use the role. Always N. Included for compatibility only.</td>
</tr>
</tbody>
</table>

10.30 DBA_SEQUENCES

The DBA_SEQUENCES view provides information about all user-defined sequences.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequence_owner</td>
<td>TEXT</td>
<td>User name of the sequence's owner.</td>
</tr>
<tr>
<td>sequence_name</td>
<td>TEXT</td>
<td>Name of the sequence.</td>
</tr>
<tr>
<td>min_value</td>
<td>NUMERIC</td>
<td>The lowest value that the server will assign to the sequence.</td>
</tr>
<tr>
<td>max_value</td>
<td>NUMERIC</td>
<td>The highest value that the server will assign to the sequence.</td>
</tr>
<tr>
<td>increment_by</td>
<td>NUMERIC</td>
<td>The value added to the current sequence number to create the next sequent number.</td>
</tr>
<tr>
<td>cycle_flag</td>
<td>VARCHAR</td>
<td>Specifies if the sequence should wrap when it reaches min_value or max_value.</td>
</tr>
<tr>
<td>order_flag</td>
<td>VARCHAR</td>
<td>Included for Oracle compatibility; will always return Y.</td>
</tr>
<tr>
<td>cache_size</td>
<td>NUMERIC</td>
<td>The number of pre-allocated sequence numbers stored in memory.</td>
</tr>
<tr>
<td>last_number</td>
<td>NUMERIC</td>
<td>The value of the last sequence number saved to disk.</td>
</tr>
</tbody>
</table>

10.31 DBA_SOURCE

The DBA_SOURCE view provides the source code listing of all objects in the database.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>owner</td>
<td>VARCHAR2</td>
<td>User name of the program’s owner.</td>
</tr>
<tr>
<td>schemaname</td>
<td>VARCHAR2</td>
<td>Name of the schema in which the program belongs.</td>
</tr>
<tr>
<td>name</td>
<td>VARCHAR2</td>
<td>Name of the program.</td>
</tr>
<tr>
<td>type</td>
<td>VARCHAR2</td>
<td>Type of program – possible values are: FUNCTION, PACKAGE, PACKAGE BODY, PROCEDURE, and TRIGGER.</td>
</tr>
<tr>
<td>line</td>
<td>INTEGER</td>
<td>Source code line number relative to a given program.</td>
</tr>
<tr>
<td>text</td>
<td>VARCHAR2</td>
<td>Line of source code text.</td>
</tr>
</tbody>
</table>
## 10.32 DBA_SYNONYMS

The **DBA_SYNONYM** view provides information about all synonyms in the database.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>owner</td>
<td>VARCHAR2</td>
<td>User name of the synonym’s owner.</td>
</tr>
<tr>
<td>schema_name</td>
<td>TEXT</td>
<td>Name of the schema in which the synonym belongs.</td>
</tr>
<tr>
<td>synonym_name</td>
<td>VARCHAR2</td>
<td>Name of the synonym.</td>
</tr>
<tr>
<td>table_owner</td>
<td>VARCHAR2</td>
<td>User name of the table’s owner on which the synonym is defined.</td>
</tr>
<tr>
<td>table_name</td>
<td>VARCHAR2</td>
<td>Name of the table on which the synonym is defined.</td>
</tr>
<tr>
<td>db_link</td>
<td>VARCHAR2</td>
<td>Name of any associated database link.</td>
</tr>
</tbody>
</table>

## 10.33 DBA_TABLES

The **DBA_TABLES** view provides information about all tables in the database.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>owner</td>
<td>TEXT</td>
<td>User name of the table’s owner.</td>
</tr>
<tr>
<td>schemaname</td>
<td>TEXT</td>
<td>Name of the schema in which the table belongs.</td>
</tr>
<tr>
<td>table_name</td>
<td>TEXT</td>
<td>Name of the table.</td>
</tr>
<tr>
<td>table_space</td>
<td>TEXT</td>
<td>Name of the tablespace in which the table resides if other than the default tablespace.</td>
</tr>
<tr>
<td>status</td>
<td>CHAR(5)</td>
<td>Included for compatibility only; always set to <strong>VALID</strong>.</td>
</tr>
<tr>
<td>temporary</td>
<td>CHAR(1)</td>
<td>Y if the table is temporary; N if the table is permanent.</td>
</tr>
</tbody>
</table>

## 10.34 DBA_TRIGGERS

The **DBA_TRIGGERS** view provides information about all triggers in the database.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>owner</td>
<td>TEXT</td>
<td>User name of the trigger’s owner.</td>
</tr>
<tr>
<td>trigger_name</td>
<td>TEXT</td>
<td>The name of the trigger.</td>
</tr>
<tr>
<td>trigger_type</td>
<td>TEXT</td>
<td>The type of the trigger. Possible values are:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BEFORE ROW</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BEFORE STATEMENT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AFTER ROW</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AFTER STATEMENT</td>
</tr>
<tr>
<td>triggering_event</td>
<td>TEXT</td>
<td>The event that fires the trigger.</td>
</tr>
<tr>
<td>table_owner</td>
<td>TEXT</td>
<td>The user name of the owner of the table on which the trigger is defined.</td>
</tr>
<tr>
<td>base_object_type</td>
<td>TEXT</td>
<td>Included for compatibility only. Value will always be <strong>TABLE</strong>.</td>
</tr>
<tr>
<td>table_name</td>
<td>TEXT</td>
<td>The name of the table on which the trigger is defined.</td>
</tr>
<tr>
<td>referencing_name</td>
<td>TEXT</td>
<td>Included for compatibility only. Value will always be</td>
</tr>
<tr>
<td></td>
<td></td>
<td>REFERENCING NEW AS NEW OLD AS OLD.</td>
</tr>
</tbody>
</table>
### Database Compatibility for Oracle® Developer’s Guide

**10.35 DBA_TYPES**

The `DBA_TYPES` view provides information about all object types in the database.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Owner</td>
<td>TEXT</td>
<td>The owner of the object type.</td>
</tr>
<tr>
<td>schema_name</td>
<td>TEXT</td>
<td>The name of the schema in which the type is defined.</td>
</tr>
<tr>
<td>type_name</td>
<td>TEXT</td>
<td>The name of the type.</td>
</tr>
<tr>
<td>type_oid</td>
<td>OID</td>
<td>The object identifier (OID) of the type.</td>
</tr>
<tr>
<td>typecode</td>
<td>TEXT</td>
<td>The typecode of the type. Possible values are:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>OBJECT, COLLECTION, OTHER</td>
</tr>
<tr>
<td>attributes</td>
<td>INTEGER</td>
<td>The number of attributes in the type.</td>
</tr>
</tbody>
</table>

**10.36 DBA_USERS**

The `DBA_USERS` view provides information about all users of the database.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Username</td>
<td>TEXT</td>
<td>User name of the user.</td>
</tr>
<tr>
<td>user_id</td>
<td>OID</td>
<td>ID number of the user.</td>
</tr>
<tr>
<td>password</td>
<td>VARCHAR2(30)</td>
<td>The password (encrypted) of the user.</td>
</tr>
<tr>
<td>account_status</td>
<td>VARCHAR2(32)</td>
<td>The current status of the account. Possible values are:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>EXPIRED &amp; LOCKED, OPEN, LOCKED</td>
</tr>
<tr>
<td>lock_date</td>
<td>TIMESTAMP W/O ZONE</td>
<td>Included for compatibility only. The value is always NULL.</td>
</tr>
<tr>
<td>expiry_date</td>
<td>TIMESTAMP W/O ZONE</td>
<td>The expiration date of the account.</td>
</tr>
<tr>
<td>default_tablespace</td>
<td>VARCHAR2(30)</td>
<td>The default tablespace associated with the account.</td>
</tr>
<tr>
<td>temporary_tablespace</td>
<td>VARCHAR2(30)</td>
<td>Included for compatibility only. The value will always be &quot;&quot; (an empty string).</td>
</tr>
<tr>
<td>created</td>
<td>TIMESTAMP W/O ZONE</td>
<td>Included for compatibility only. The value is always NULL.</td>
</tr>
<tr>
<td>profile</td>
<td>VARCHAR2(30)</td>
<td>Included for compatibility only. The value is always NULL.</td>
</tr>
<tr>
<td>initial_rsnc_consumer_group</td>
<td>VARCHAR2(30)</td>
<td>Included for compatibility only. The value is always NULL.</td>
</tr>
</tbody>
</table>
### 10.37 DBA_VIEWS

The `DBA_VIEWS` view provides information about all views in the database.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>owner</td>
<td>VARCHAR2</td>
<td>User name of the view’s owner.</td>
</tr>
<tr>
<td>schema_name</td>
<td>VARCHAR2</td>
<td>Name of the schema in which the view belongs.</td>
</tr>
<tr>
<td>view_name</td>
<td>VARCHAR2</td>
<td>Name of the view.</td>
</tr>
<tr>
<td>text</td>
<td>TEXT</td>
<td>The text of the <code>SELECT</code> statement that defines the view.</td>
</tr>
</tbody>
</table>

### 10.38 USER_ALL_TABLES

The `USER_ALL_TABLES` view provides information about all tables owned by the current user.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>schema_name</td>
<td>TEXT</td>
<td>Name of the schema in which the table belongs.</td>
</tr>
<tr>
<td>table_name</td>
<td>TEXT</td>
<td>Name of the table.</td>
</tr>
<tr>
<td>tablespace_name</td>
<td>TEXT</td>
<td>Name of the tablespace in which the table resides if other than the default tablespace.</td>
</tr>
<tr>
<td>status</td>
<td>VARCHAR2(5)</td>
<td>Included for compatibility only; always set to <code>VALID</code>.</td>
</tr>
<tr>
<td>temporary</td>
<td>TEXT</td>
<td>Y if the table is temporary; N if the table is permanent.</td>
</tr>
</tbody>
</table>

### 10.39 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.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>owner</td>
<td>TEXT</td>
<td>User name of the constraint’s owner.</td>
</tr>
<tr>
<td>schema_name</td>
<td>TEXT</td>
<td>Name of the schema in which the constraint belongs.</td>
</tr>
<tr>
<td>constraint_name</td>
<td>TEXT</td>
<td>The name of the constraint.</td>
</tr>
<tr>
<td>table_name</td>
<td>TEXT</td>
<td>The name of the table to which the constraint belongs.</td>
</tr>
<tr>
<td>column_name</td>
<td>TEXT</td>
<td>The name of the column referenced in the constraint.</td>
</tr>
<tr>
<td>position</td>
<td>SMALLINT</td>
<td>The position of the column within the object definition.</td>
</tr>
<tr>
<td>constraint_def</td>
<td>TEXT</td>
<td>The definition of the constraint.</td>
</tr>
</tbody>
</table>
10.40 USER_CONSTRAINTS

The USER_CONSTRAINTS view provides information about all constraints placed on tables that are owned by the current user.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>schema_name</td>
<td>TEXT</td>
<td>Name of the schema in which the constraint belongs.</td>
</tr>
<tr>
<td>constraint_name</td>
<td>TEXT</td>
<td>The name of the constraint.</td>
</tr>
<tr>
<td>constraint_type</td>
<td>TEXT</td>
<td>The constraint type. Possible values are:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>C – check constraint</td>
</tr>
<tr>
<td></td>
<td></td>
<td>F – foreign key constraint</td>
</tr>
<tr>
<td></td>
<td></td>
<td>P – primary key constraint</td>
</tr>
<tr>
<td></td>
<td></td>
<td>U – unique key constraint</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R – referential integrity constraint</td>
</tr>
<tr>
<td></td>
<td></td>
<td>V – constraint on a view</td>
</tr>
<tr>
<td></td>
<td></td>
<td>O – with read-only, on a view</td>
</tr>
<tr>
<td>table_name</td>
<td>TEXT</td>
<td>Name of the table to which the constraint belongs.</td>
</tr>
<tr>
<td>search_condition</td>
<td>TEXT</td>
<td>Search condition that applies to a check constraint.</td>
</tr>
<tr>
<td>r_owner</td>
<td>TEXT</td>
<td>Owner of a table referenced by a referential constraint.</td>
</tr>
<tr>
<td>r_constraint_name</td>
<td>TEXT</td>
<td>Name of the constraint definition for a referenced table.</td>
</tr>
<tr>
<td>delete_rule</td>
<td>TEXT</td>
<td>The delete rule for a referential constraint. Possible values are:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>C – cascade</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R – restrict</td>
</tr>
<tr>
<td></td>
<td></td>
<td>N – no action</td>
</tr>
<tr>
<td>deferrable</td>
<td>BOOLEAN</td>
<td>Specified if the constraint is deferrable (Y or N).</td>
</tr>
<tr>
<td>deferred</td>
<td>BOOLEAN</td>
<td>Specifies if the constraint has been deferred (Y or N).</td>
</tr>
<tr>
<td>index_owner</td>
<td>TEXT</td>
<td>User name of the index owner.</td>
</tr>
<tr>
<td>index_name</td>
<td>TEXT</td>
<td>The name of the index.</td>
</tr>
<tr>
<td>constraint_def</td>
<td>TEXT</td>
<td>The definition of the constraint.</td>
</tr>
</tbody>
</table>

10.41 USER_DB_LINKS

The USER_DB_LINKS view provides information about all database links that are owned by the current user.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>schema_name</td>
<td>TEXT</td>
<td>Name of the schema in which the link belongs.</td>
</tr>
<tr>
<td>db_link</td>
<td>TEXT</td>
<td>The name of the database link.</td>
</tr>
<tr>
<td>type</td>
<td>VARCHAR2</td>
<td>Type of remote server. Value will be either REDWOOD or EDB</td>
</tr>
<tr>
<td>username</td>
<td>TEXT</td>
<td>User name of the user logging in.</td>
</tr>
<tr>
<td>password</td>
<td>TEXT</td>
<td>Password used to authenticate on the remote server.</td>
</tr>
<tr>
<td>host</td>
<td>TEXT</td>
<td>Name or IP address of the remote server.</td>
</tr>
</tbody>
</table>
The `USER_IND_COLUMNS` view provides information about all columns referred to in indexes on tables that are owned by the current user.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>schema_name</td>
<td>TEXT</td>
<td>Name of the schema in which the index belongs.</td>
</tr>
<tr>
<td>index_name</td>
<td>TEXT</td>
<td>The name of the index.</td>
</tr>
<tr>
<td>table_owner</td>
<td>TEXT</td>
<td>User name of the table owner.</td>
</tr>
<tr>
<td>table_name</td>
<td>TEXT</td>
<td>The name of the table to which the index belongs.</td>
</tr>
<tr>
<td>column_name</td>
<td>TEXT</td>
<td>The name of the column.</td>
</tr>
<tr>
<td>column_position</td>
<td>SMALLINT</td>
<td>The position of the column within the index.</td>
</tr>
<tr>
<td>column_length</td>
<td>SMALLINT</td>
<td>The length of the column (in bytes).</td>
</tr>
<tr>
<td>char_length</td>
<td>NUMERIC</td>
<td>The length of the column (in characters).</td>
</tr>
<tr>
<td>descend</td>
<td>CHAR(1)</td>
<td>Sorted order of the column on disk. Always set to Y (descending); included for compatibility only.</td>
</tr>
</tbody>
</table>

The `USER_INDEXES` view provides information about all indexes on tables that are owned by the current user.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>owner</td>
<td>TEXT</td>
<td>User name of the index’s owner.</td>
</tr>
<tr>
<td>index_schema</td>
<td>TEXT</td>
<td>Name of the schema in which the index belongs.</td>
</tr>
<tr>
<td>index_name</td>
<td>TEXT</td>
<td>The name of the index.</td>
</tr>
<tr>
<td>index_type</td>
<td>TEXT</td>
<td>The index type is always BTREE. Included for compatibility only.</td>
</tr>
<tr>
<td>table_owner</td>
<td>TEXT</td>
<td>User name of the owner of the indexed table.</td>
</tr>
<tr>
<td>table_name</td>
<td>TEXT</td>
<td>The name of the indexed table.</td>
</tr>
<tr>
<td>table_type</td>
<td>TEXT</td>
<td>Included for compatibility only. Currently, always set to TABLE.</td>
</tr>
<tr>
<td>uniqueness</td>
<td>TEXT</td>
<td>Indicates if the index is UNIQUE or NONUNIQUE.</td>
</tr>
<tr>
<td>compression</td>
<td>CHAR(1)</td>
<td>Always set to N (not compressed). Included for compatibility only.</td>
</tr>
<tr>
<td>tablespace_name</td>
<td>TEXT</td>
<td>Name of the tablespace in which the table resides if other than the default tablespace.</td>
</tr>
<tr>
<td>logging</td>
<td>TEXT</td>
<td>Included for compatibility only. Currently always set to LOGGING.</td>
</tr>
<tr>
<td>status</td>
<td>TEXT</td>
<td>Whether or not the state of the object is valid. (VALID or INVALID).</td>
</tr>
<tr>
<td>partitioned</td>
<td>CHAR(3)</td>
<td>Indicates that the index is partitioned. Currently, always set to NO.</td>
</tr>
<tr>
<td>temporary</td>
<td>CHAR(1)</td>
<td>Indicates that an index is on a temporary table. Currently, always set to N.</td>
</tr>
<tr>
<td>secondary</td>
<td>CHAR(1)</td>
<td>Included for compatibility only. Currently always set to N.</td>
</tr>
<tr>
<td>join_index</td>
<td>CHAR(3)</td>
<td>Included for compatibility only. Currently always set to NO.</td>
</tr>
</tbody>
</table>
### 10.44 USER_JOBS

The USER_JOBS view provides information about all jobs owned by the current user.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>job</td>
<td>INTEGER</td>
<td>The identifier of the job (Job ID).</td>
</tr>
<tr>
<td>log_user</td>
<td>TEXT</td>
<td>The name of the user that submitted the job.</td>
</tr>
<tr>
<td>priv_user</td>
<td>TEXT</td>
<td>Same as log_user. Included for compatibility only.</td>
</tr>
<tr>
<td>schema_user</td>
<td>TEXT</td>
<td>The name of the schema used to parse the job.</td>
</tr>
<tr>
<td>last_date</td>
<td>TIMESTAMP WITH TIME ZONE</td>
<td>The last date that this job executed successfully.</td>
</tr>
<tr>
<td>last_sec</td>
<td>TEXT</td>
<td>Same as last_date.</td>
</tr>
<tr>
<td>this_date</td>
<td>TIMESTAMP WITH TIME ZONE</td>
<td>The date that the job began executing.</td>
</tr>
<tr>
<td>this_sec</td>
<td>TEXT</td>
<td>Same as this_date.</td>
</tr>
<tr>
<td>next_date</td>
<td>TIMESTAMP WITH TIME ZONE</td>
<td>The next date that this job will be executed.</td>
</tr>
<tr>
<td>next_sec</td>
<td>TEXT</td>
<td>Same as next_date.</td>
</tr>
<tr>
<td>total_time</td>
<td>integer</td>
<td>The execution time of this job (in seconds).</td>
</tr>
<tr>
<td>broken</td>
<td>TEXT</td>
<td>If Y, no attempt will be made to run this job. If N, this job will attempt to execute.</td>
</tr>
<tr>
<td>interval</td>
<td>TEXT</td>
<td>Determines how often the job will repeat.</td>
</tr>
<tr>
<td>what</td>
<td>TEXT</td>
<td>The job definition (PL/SQL code block) that runs when the job executes.</td>
</tr>
<tr>
<td>failures</td>
<td>BIGINT</td>
<td>The number of times that the job has failed to complete since it’s last successful execution.</td>
</tr>
<tr>
<td>nls_env</td>
<td>VARCHAR2(4000)</td>
<td>Always NULL. Provided for compatibility only.</td>
</tr>
<tr>
<td>misc_env</td>
<td>BYTEA</td>
<td>Always NULL. Provided for compatibility only.</td>
</tr>
<tr>
<td>instance</td>
<td>NUMERIC</td>
<td>Always 0. Provided for compatibility only.</td>
</tr>
</tbody>
</table>

### 10.45 USER_OBJECTS

The USER_OBJECTS view provides information about all objects that are owned by the current user.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>owner</td>
<td>VARCHAR2</td>
<td>User name of the object’s owner.</td>
</tr>
<tr>
<td>schemaname</td>
<td>VARCHAR2</td>
<td>Name of the schema in which the object belongs.</td>
</tr>
<tr>
<td>object_name</td>
<td>VARCHAR2</td>
<td>Name of the object.</td>
</tr>
<tr>
<td>object_type</td>
<td>VARCHAR2</td>
<td>Type of the object – possible values are: INDEX, FUNCTION,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PACKAGE, PACKAGE BODY, PROCEDURE, SEQUENCE, SYNONYM, TABLE, TRIGGER, and VIEW.</td>
</tr>
<tr>
<td>status</td>
<td>VARCHAR2</td>
<td>Included for compatibility only; always set to VALID.</td>
</tr>
</tbody>
</table>
10.46 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.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object_name</td>
<td>NAME</td>
<td>Name of the object on which the policy applies.</td>
</tr>
<tr>
<td>policy_group</td>
<td>TEXT</td>
<td>Name of the policy group. Included for compatibility only; always set to an empty string.</td>
</tr>
<tr>
<td>policy_name</td>
<td>NAME</td>
<td>Name of the policy.</td>
</tr>
<tr>
<td>pf_owner</td>
<td>NAME</td>
<td>Name of the schema containing the policy function, or the schema containing the package that contains the policy function.</td>
</tr>
<tr>
<td>package</td>
<td>NAME</td>
<td>Name of the package containing the policy function if the function belongs to a package.</td>
</tr>
<tr>
<td>function</td>
<td>NAME</td>
<td>Name of the policy function.</td>
</tr>
<tr>
<td>sel</td>
<td>TEXT</td>
<td>Whether or not the policy applies to SELECT commands. Possible values are YES or NO.</td>
</tr>
<tr>
<td>ins</td>
<td>TEXT</td>
<td>Whether or not the policy applies to INSERT commands. Possible values are YES or NO.</td>
</tr>
<tr>
<td>upd</td>
<td>TEXT</td>
<td>Whether or not the policy applies to UPDATE commands. Possible values are YES or NO.</td>
</tr>
<tr>
<td>del</td>
<td>TEXT</td>
<td>Whether or not the policy applies to DELETE commands. Possible values are YES or NO.</td>
</tr>
<tr>
<td>idx</td>
<td>TEXT</td>
<td>Whether or not the policy applies to index maintenance. Possible values are YES or NO.</td>
</tr>
<tr>
<td>chk_option</td>
<td>TEXT</td>
<td>Whether or not the check option is in force for INSERT and UPDATE commands. Possible values are YES or NO.</td>
</tr>
<tr>
<td>enable</td>
<td>TEXT</td>
<td>Whether or not the policy is enabled on the object. Possible values are YES or NO.</td>
</tr>
<tr>
<td>static_policy</td>
<td>TEXT</td>
<td>Whether or not the policy is static. Included for compatibility only; always set to NO.</td>
</tr>
<tr>
<td>policy_type</td>
<td>TEXT</td>
<td>Policy type. Included for compatibility only; always set to UNKNOWN.</td>
</tr>
<tr>
<td>long_predicate</td>
<td>TEXT</td>
<td>Whether or not predicates longer than 4000 bytes can be returned by the policy function. Included for compatibility only; always set to YES.</td>
</tr>
</tbody>
</table>
10.47 USER_SEQUENCES

The USER_SEQUENCES view provides information about all user-defined sequences that belong to the current user.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequence_name</td>
<td>TEXT</td>
<td>Name of the sequence.</td>
</tr>
<tr>
<td>min_value</td>
<td>NUMERIC</td>
<td>The lowest value that the server will assign to the sequence.</td>
</tr>
<tr>
<td>max_value</td>
<td>NUMERIC</td>
<td>The highest value that the server will assign to the sequence.</td>
</tr>
<tr>
<td>increment_by</td>
<td>NUMERIC</td>
<td>The value added to the current sequence number to create the next sequent number.</td>
</tr>
<tr>
<td>cycle_flag</td>
<td>VARCHAR</td>
<td>Specifies if the sequence should wrap when it reaches min_value or max_value.</td>
</tr>
<tr>
<td>order_flag</td>
<td>VARCHAR</td>
<td>Included for Oracle compatibility; will always return Y.</td>
</tr>
<tr>
<td>cache_size</td>
<td>NUMERIC</td>
<td>The number of pre-allocated sequence numbers stored in memory.</td>
</tr>
<tr>
<td>last_number</td>
<td>NUMERIC</td>
<td>The value of the last sequence number saved to disk.</td>
</tr>
</tbody>
</table>

10.48 USER_SOURCE

The USER_SOURCE view provides information about all programs owned by the current user.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>schema_name</td>
<td>VARCHAR2</td>
<td>Name of the schema in which the program belongs.</td>
</tr>
<tr>
<td>name</td>
<td>VARCHAR2</td>
<td>Name of the program.</td>
</tr>
<tr>
<td>type</td>
<td>VARCHAR2</td>
<td>Type of program – possible values are: FUNCTION, PACKAGE, PACKAGE BODY, PROCEDURE, and TRIGGER.</td>
</tr>
<tr>
<td>line</td>
<td>INTEGER</td>
<td>Source code line number relative to a given program.</td>
</tr>
<tr>
<td>text</td>
<td>VARCHAR2</td>
<td>Line of source code text.</td>
</tr>
</tbody>
</table>

10.49 USER_SYNONYMS

The USER_SYNONYMS view provides information about all synonyms owned by the current user.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>owner</td>
<td>TEXT</td>
<td>User name of the synonym’s owner.</td>
</tr>
<tr>
<td>synonym_name</td>
<td>TEXT</td>
<td>Name of the synonym.</td>
</tr>
<tr>
<td>object_owner</td>
<td>TEXT</td>
<td>User name of the table’s owner on which the synonym is defined.</td>
</tr>
<tr>
<td>object_name</td>
<td>TEXT</td>
<td>Name of the table on which the synonym is defined.</td>
</tr>
<tr>
<td>status</td>
<td>VARCHAR2(5)</td>
<td>Included for compatibility only; always set to VALID.</td>
</tr>
<tr>
<td>db_link</td>
<td>TEXT</td>
<td>Name of any associated database link.</td>
</tr>
</tbody>
</table>
10.50 USER_TAB_COLUMNS

The USER_TAB_COLUMNS view displays information about all columns in tables owned by the current user.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>owner</td>
<td>TEXT</td>
<td>User name of the table’s owner.</td>
</tr>
<tr>
<td>schemaname</td>
<td>TEXT</td>
<td>Name of the schema in which the table belongs.</td>
</tr>
<tr>
<td>table_name</td>
<td>TEXT</td>
<td>Name of the table.</td>
</tr>
<tr>
<td>column_name</td>
<td>TEXT</td>
<td>Name of the column.</td>
</tr>
<tr>
<td>data_type</td>
<td>VARCHAR2</td>
<td>Data type of the column.</td>
</tr>
<tr>
<td>data_length</td>
<td>INTEGER</td>
<td>Length of text columns.</td>
</tr>
<tr>
<td>data_precision</td>
<td>INTEGER</td>
<td>Precision (number of digits) for NUMBER columns.</td>
</tr>
<tr>
<td>data_scale</td>
<td>INTEGER</td>
<td>Scale of NUMBER columns.</td>
</tr>
<tr>
<td>column_id</td>
<td>INTEGER</td>
<td>Relative position of the column within the table.</td>
</tr>
<tr>
<td>nullable</td>
<td>BPCHAR</td>
<td>Whether or not the column is nullable – possible values are: Y – column is nullable; N – column does not allow null.</td>
</tr>
<tr>
<td>data_default</td>
<td>VARCHAR2</td>
<td>Default value assigned to the column.</td>
</tr>
</tbody>
</table>

10.51 USER_TABLES

The USER_TABLES view displays information about all tables owned by the current user.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>owner</td>
<td>TEXT</td>
<td>User name of the table’s owner.</td>
</tr>
<tr>
<td>schemaname</td>
<td>TEXT</td>
<td>Name of the schema in which the table belongs.</td>
</tr>
<tr>
<td>table_name</td>
<td>TEXT</td>
<td>Name of the table.</td>
</tr>
<tr>
<td>table_space</td>
<td>TEXT</td>
<td>Name of the tablespace in which the table resides if other than the default tablespace.</td>
</tr>
<tr>
<td>status</td>
<td>VARCHAR2(5)</td>
<td>Included for compatibility only; always set to VALID.</td>
</tr>
</tbody>
</table>

10.52 USER_TRIGGERS

The USER_TRIGGERS view displays information about all triggers on tables owned by the current user.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>trigger_name</td>
<td>TEXT</td>
<td>The name of the trigger.</td>
</tr>
<tr>
<td>trigger_type</td>
<td>TEXT</td>
<td>The type of the trigger. Possible values are:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BEFORE ROW</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BEFORE STATEMENT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AFTER ROW</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AFTER STATEMENT</td>
</tr>
<tr>
<td>triggering_event</td>
<td>TEXT</td>
<td>The event that fires the trigger.</td>
</tr>
<tr>
<td>table_owner</td>
<td>TEXT</td>
<td>The user name of the owner of the table on which the trigger is defined.</td>
</tr>
</tbody>
</table>
### 10.53 USER_TYPES

The **USER_TYPES** view provides information about all object types owned by the current user.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>schema_name</td>
<td>TEXT</td>
<td>The name of the schema in which the type is defined.</td>
</tr>
<tr>
<td>type_name</td>
<td>TEXT</td>
<td>The name of the type.</td>
</tr>
<tr>
<td>type_oid</td>
<td>OID</td>
<td>The object identifier (OID) of the type.</td>
</tr>
<tr>
<td>typecode</td>
<td>TEXT</td>
<td>The typecode of the type. Possible values are:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>OBJECT, COLLECTION, OTHER</td>
</tr>
<tr>
<td>attributes</td>
<td>INTEGER</td>
<td>The number of attributes in the type.</td>
</tr>
</tbody>
</table>

### 10.54 USER_USERS

The **USER_USERS** view provides information about the current user.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>username</td>
<td>TEXT</td>
<td>User name of the user.</td>
</tr>
<tr>
<td>user_id</td>
<td>OID</td>
<td>ID number of the user.</td>
</tr>
<tr>
<td>account_status</td>
<td>VARCHAR2(32)</td>
<td>The current status of the account. Possible values are:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>EXPIRED &amp; LOCKED, OPEN, LOCKED</td>
</tr>
<tr>
<td>lock_date</td>
<td>TIMESTAMP W/O ZONE</td>
<td>Included for compatibility only. The value is always NULL.</td>
</tr>
<tr>
<td>expiry_date</td>
<td>TIMESTAMP W/O ZONE</td>
<td>The expiration date of the account.</td>
</tr>
<tr>
<td>default_tablespace</td>
<td>VARCHAR2(30)</td>
<td>The default tablespace associated with the account.</td>
</tr>
<tr>
<td>temporary_tablespace</td>
<td>VARCHAR2(30)</td>
<td>Included for compatibility only. The value will always be &quot;&quot; (an empty string).</td>
</tr>
<tr>
<td>created</td>
<td>TIMESTAMP W/O ZONE</td>
<td>Included for compatibility only. The value will always be NULL.</td>
</tr>
<tr>
<td>Name</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------</td>
<td>-----------------</td>
<td>-----------------------------------------------------</td>
</tr>
<tr>
<td>initial_rsrc_consumer_group</td>
<td>VARCHAR2(30)</td>
<td>Included for compatibility only. The value will always be NULL.</td>
</tr>
<tr>
<td>external_name</td>
<td>VARCHAR2(4000)</td>
<td>Included for compatibility only; always set to NULL.</td>
</tr>
</tbody>
</table>

### 10.55 USER_VIEW_COLUMNS

The USER_VIEW_COLUMNS view provides information about all columns in views owned by the current user.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Owner</td>
<td>VARCHAR2</td>
<td>User name of the view’s owner.</td>
</tr>
<tr>
<td>schemaname</td>
<td>VARCHAR2</td>
<td>Name of the schema in which the view belongs.</td>
</tr>
<tr>
<td>view_name</td>
<td>VARCHAR2</td>
<td>Name of the view.</td>
</tr>
<tr>
<td>column_name</td>
<td>VARCHAR2</td>
<td>Name of the column.</td>
</tr>
<tr>
<td>data_type</td>
<td>VARCHAR2</td>
<td>Data type of the column.</td>
</tr>
<tr>
<td>data_length</td>
<td>INTEGER</td>
<td>Length of text columns.</td>
</tr>
<tr>
<td>data_precision</td>
<td>INTEGER</td>
<td>Precision (number of digits) for NUMBER columns.</td>
</tr>
<tr>
<td>data_scale</td>
<td>INTEGER</td>
<td>Scale of NUMBER columns.</td>
</tr>
<tr>
<td>column_id</td>
<td>INTEGER</td>
<td>Relative position of the column within the view.</td>
</tr>
<tr>
<td>nullable</td>
<td>CHARACTER</td>
<td>Whether or not the column is nullable – possible values are: Y – column is nullable; N – column does not allow null.</td>
</tr>
<tr>
<td>data_default</td>
<td>VARCHAR2</td>
<td>Default value assigned to the column.</td>
</tr>
</tbody>
</table>

### 10.56 USER_VIEWS

The USER_VIEWS view provides information about all views owned by the current user.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Owner</td>
<td>VARCHAR2</td>
<td>User name of the view’s owner.</td>
</tr>
<tr>
<td>schemaname</td>
<td>VARCHAR2</td>
<td>Name of the schema in which the view belongs.</td>
</tr>
<tr>
<td>view_name</td>
<td>VARCHAR2</td>
<td>Name of the view.</td>
</tr>
<tr>
<td>status</td>
<td>VARCHAR2</td>
<td>Included for compatibility only; always set to VALID.</td>
</tr>
</tbody>
</table>

### 10.57 V$VERSION

The V$VERSION view provides information about product compatibility.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>banner</td>
<td>TEXT</td>
<td>Displays product compatibility information.</td>
</tr>
</tbody>
</table>
The `PRODUCT_COMPONENT_VERSION` view provides version information about product version compatibility.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>product</td>
<td>VARCHAR(74)</td>
<td>The name of the product.</td>
</tr>
<tr>
<td>version</td>
<td>VARCHAR(74)</td>
<td>The version number of the product.</td>
</tr>
<tr>
<td>status</td>
<td>VARCHAR(74)</td>
<td>Included for compatibility; always Available.</td>
</tr>
</tbody>
</table>
11 Utilities

The sections in this chapter describe various utility programs. These include:

- EDB*Plus
- EDB*Loader
- EDB*Wrap
- Dynamic Runtime Instrumentation

11.1 EDB*Plus

EDB*Plus is a utility program that provides a command line user interface to the Postgres Plus Advanced Server. EDB*Plus accepts SQL commands, SPL anonymous blocks, and EDB*Plus commands. EDB*Plus commands are compatible with Oracle SQL*Plus commands and provide 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 a Postgres Plus Advanced Server database using EDB*Plus. The final section provides a summary of the EDB*Plus commands.

11.1.1 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 [ -SILENT ] [ 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.)

```plaintext
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 Postgres Plus Advanced Server home directory.

```plaintext
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.

```plaintext
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\9.2AS\edbplus>edbplus enterprisedb/password
Connected to EnterpriseDB 9.2.0.0 (localhost:5444/edb) AS enterprisedb
EDB*Plus: Release 9.2
Copyright (c) 2008-2012, 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\9.2AS\edbplus>edbplus enterprisedb/password@localhost:5445/edb
Connected to EnterpriseDB 9.2.0.0 (localhost:5445/edb) AS enterprisedb
EDB*Plus: Release 9.2
Copyright (c) 2008-2012, 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\9.2AS\edbplus>edbplus enterprisedb/password@hr_5445
Connected to EnterpriseDB 9.2.0.0 (localhost:5445/hr) AS enterprisedb
EDB*Plus: Release 9.2 (Build 28)
Copyright (c) 2008-2012, EnterpriseDB Corporation. All rights reserved.
SQL>
```

The following is the content of the `login.sql` file used in the previous example.

```sql
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\9.2AS\edbplus>edbplus
enterprisedb/password @dept_query
Connected to EnterpriseDB 9.2.0.0 (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
```

Copyright © 2007 - 2015 EnterpriseDB Corporation. All rights reserved.
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
```

11.1.2 Command Summary

This section contains a summary of EDB*Plus commands.

11.1.2.1 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.

```
ACCEPT 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"
```

11.1.2.2 APPEND

APPEND is a line editor command that appends the given text to the end of the current line in the SQL buffer.

```
APPEND 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
```
11.1.2.3  CHANGE

CHANGE is a line editor command performs a search-and-replace on the current line in the SQL buffer.

```
C[CHANGE ] /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
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
```

11.1.2.4  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[EEEN ] ]
```

- `BUFFER` clears the SQL buffer.
- `COLUMNS` removes column definitions.
- `SCREEN` clears the screen. This is the default if no options are specified.
11.1.2.5 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.

```
COLUMN
 [ column
   { CLEAR | FORMAT spec | HEADING 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:

1. No other options
2. CLEAR
3. 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:

```
An
```

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, \textit{spec} is comprised of the following elements.

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>Display a leading dollar sign.</td>
</tr>
<tr>
<td>,</td>
<td>Display a comma in the indicated position.</td>
</tr>
<tr>
<td>.</td>
<td>Marks the location of the decimal point.</td>
</tr>
<tr>
<td>0</td>
<td>Display leading zeros.</td>
</tr>
<tr>
<td>9</td>
<td>Number of significant digits to display.</td>
</tr>
</tbody>
</table>

If loss of significant digits occurs due to overflow of the format, then all \#’s are displayed.

\textit{text}

Text to be used for the column heading of \textit{column}.

OFF | ON

If \textit{OFF} is specified, formatting options are reverted back to their defaults, but are still available within the session. If \textit{ON} is specified, the formatting options specified by previous \texttt{COLUMN} commands for \textit{column} within the session are re-activated.

The following example shows the effect of changing the display width of the \textit{job} column.

```sql
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  MANAGER
7654 MARTIN  SALES MAN
7698 BLAKE  MANAGER
```
The following example applies a format to the sal column.

```
SQL> COLUMN sal FORMAT $99,999.00
SQL> COLUMN JOB ON FORMAT A5
   wrapped
SQL> 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
    7521 WARD       SALES   $1,250.00
    7566 JONES      MANAGER  $2,975.00
    7654 MARTIN     SALES   $1,250.00
    7698 BLAKE      MANAGER  $2,850.00
    7782 CLARK      MANAGER  $2,450.00
    7788 SCOTT      ANALYST  $3,000.00
    7839 KING       PRESIDENT  $5,000.00
    7844 TURNER     SALES   $1,500.00
```
11.1.2.6 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 Postgres Plus 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 9.2.0.0 (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 9.2.0.0 (localhost:5444/edb) AS enterprisedb
```
11.1.2.7 DEFINE

The DEFINE command creates or replaces the value of a user variable (also called a substitution variable).

```
DEFINE [ 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 EDB = "localhost:5445/edb"
SQL> DEFINE DEPT = "20"
SQL> DEFINE NAME = "John Smith"
```

Note: The variable `EDB` is read from the `login.sql` file located in the `edbplus` subdirectory of the Postgres Plus Advanced Server home directory.

11.1.2.8 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 `n`th through the `m`th lines
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
```

### 11.1.2.9 DESCRIBE

The `DESCRIBE` command displays a list of columns, data types, and lengths for a table or view; a list of parameters for a procedure or function; or a list of procedures and functions and their respective parameters for a package.

```
DESCRIBE [ 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.

### 11.1.2.10 DISCONNECT

The `DISCONNECT` command closes the current database connection, but does not terminate EDB*Plus.

```
DISCONNECT
```
11.1.2.11 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.

11.1.2.12 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.

11.1.2.13 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.

11.1.2.14 GET

The GET command loads the contents of the given file to the SQL buffer.

GET filename[.ext ] [ LIST | NOLIST ]

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.

11.1.2.15 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.

11.1.2.16 HOST

The HOST command executes an operating system command from EDB*Plus.

HOST [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.

11.1.2.17 INPUT

The INPUT line editor command adds a line of text to the SQL buffer after the current line.

INPUT 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
```

11.1.2.18 LIST

LIST is a line editor command that displays the contents of the SQL buffer.

LIST [ 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 LAST

n LAST 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.

* LAST

* LAST displays a list of lines that range from the current line through the last line.

LAST

LAST displays the last line.

11.1.2.19 PASSWORD

Use the PASSWORD command to change your database password.

PASSWORD [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.
```

11.1.2.20 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.

11.1.2.21  PRINT

The PRINT command displays the value of a bind variable.

PRINT [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.

11.1.2.22  PROMPT

The PROMPT command displays a message to the user before continuing.

PROMPT [message_text]

message_text specifies the text displayed to the user. Double quote the string to include quotes in the output.

11.1.2.23  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.

11.1.2.24  REMARK

Use REMARK to include comments in a script.

REMARK [optional_text]

You may also use the following convention to include a comment:

/*
11.1.2.25  SAVE

Use the SAVE command to write the SQL Buffer to an operating system file.

```
SAVE [E] file_name
[CREATE] | REPLACE | APPEND]
```

`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.
```

11.1.2.26  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 FLUSH {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 HEADING {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 HEADSEP

SET LINESIZE

Use the SET LINESIZE command to specify the width of a line in characters.

    SET LINESIZE 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 NEWPAGE 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 PAGESIZE 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 SQLCASE {MIXED | UPPER | LOWER}
```

**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**.

\[
\text{SET PAUSE} \{ \text{ON | OFF} \}
\]

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:

\[
\text{SET SPACE number_of_spaces}
\]

**SET SQLPROMPT**

Use **SET SQLPROMPT** to set a value for a user-interactive prompt:

\[
\text{SET SQLPROMPT "prompt"}
\]

By default, **SQLPROMPT** is set to **"SQL>"**

**SQL TERMOUT**

Use the **SQL TERMOUT** command to specify if command output should be displayed onscreen.

\[
\text{SET TERMOUT} \{ \text{ON | OFF} \}
\]

**SQL TIMING**

The **SQL TIMING** command specifies if Advanced Server should display the execution time for each SQL statement after it is executed.

\[
\text{SET TIMING} \{ \text{ON | OFF} \}
\]

**SET VERIFY**

Specifies if both the old and new values of a SQL statement are displayed when a substitution variable is encountered.

\[
\text{SET VERIFY} \{ \text{ON | OFF} \}
\]
11.1.2.27  SHOW

Use the SHOW command to display current parameter values.

    SHOW [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          
termout         ON
timing          OFF
verify          ON
USER is         "enterprisedb"
HOST is         "localhost"
PORT is         "5444"
DATABASE is     "edb"
VERSION is      "9.2.0.0"
```

Or display a specific parameter setting by including the parameter_name in the SHOW command:

```
SQL> SHOW VERSION
VERSION is "9.2.0.0"
```

11.1.2.28  SPOOL

The SPOOL command sends output from the display to a file.

    SPOOL output_file | OFF

Use the output_file parameter to specify a path name for the output file.
11.1.2.29 **START**

Use the START command to run an EDB*Plus script file; START is an alias for @ command.

```
START script_file
```

Specify the name of a script file in the `script_file` parameter.

11.1.2.30 **UNDEFINE**

The UNDEFINE command erases a user variable created by the DEFINE command.

```
UNDEFINE variable_name [ variable_name...]
```

Use the `variable_name` parameter to specify the name of a variable or variables.

11.1.2.31 **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.
11.2 EDB*Loader

EDB*Loader is a high-performance bulk data loader that provides an Oracle compatible interface for Postgres Plus 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 the following:

- Support of 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.
11.2.1 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.

The exception is that Postgres Plus 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: Postgres Plus Advanced Server rules are created with the CREATE RULE command. Postgres Plus 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 11.2.5 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 11.2.6 for information on parallel direct path loads.
11.2.2 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 are expected to be the same. Character set conversion of the data file to the database encoding is not 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
### 11.2.3 EDB*Loader Control File

When you invoke EDB*Loader, the list of arguments 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 following:

- Name of the input data file containing the data to be loaded
- 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
- Name of the bad file for collecting illegally formatted records from the data file
- Name of the discard file for collecting records from the data file that do not meet the selection criteria of any table

The syntax for the EDB*Loader control file is as follows:

```plaintext
[ OPTIONS (param=value [, param=value ] ...) ]
LOAD DATA
  INFILE '{ data_file | stdin }'
  [ BADFILE 'bad_file' ]
  [ DISCARDFILE 'discard_file' ]
  [ INSERT | APPEND | REPLACE | TRUNCATE ]
  { 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 a `data field`, or simply, a `field`, in `data_file` that describes the location, data format, or value of the data to be inserted into `column_name` of `target_table`. The syntax of `field_def` is the following:

```plaintext
column_name {
  CONSTANT val |
  FILLER [ POSITION (start:end) ] [ fieldtype ] |
  [ POSITION (start:end) ] [ fieldtype ] [ "expr" ]
}
```
where `fieldtype` is one of:

```
CHAR | INTEGER EXTERNAL | FLOAT EXTERNAL | DECIMAL EXTERNAL |
    | ZONED EXTERNAL    | DATE [ "datemask" ]
```

**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.

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 the DISCARDFILE clause is not specified, nor is the DISCARD parameter given on the command line when `edbldr` is invoked, the discard file is not created even if the EDB*Loader session results in discarded records.

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**
param=value

param=value represents one or more of the following command line parameter/value pairs: skip, errors, skip_index_maintenance, direct, parallel.

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.

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.

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.
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.

**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*.

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 consistent with 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 consistent with 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 | INTEGER EXTERNAL | FLOAT EXTERNAL | DECIMAL EXTERNAL | ZONED EXTERNAL | DATE [ "datemask" ]

Field type that describes the format of the data field in data_file.

Note: Specification of any of these field types is optional. All data in data_file must be in human-readable, text form. Specification of a field type is 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 column data type and its compatibility with 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.
Specifies the ordering and abbreviation of the day, month, and year components of a date field.

See Section 3.5.7 for date mask formatting information.

**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.

### 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;
```

<table>
<thead>
<tr>
<th>empno</th>
<th>ename</th>
<th>job</th>
<th>mgr</th>
<th>hiredate</th>
<th>sal</th>
<th>comm</th>
<th>deptno</th>
</tr>
</thead>
<tbody>
<tr>
<td>9101</td>
<td>ROGERS</td>
<td>CLERK</td>
<td>7902</td>
<td>17-DEC-10</td>
<td>1980.00</td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>9102</td>
<td>PETERSON</td>
<td>SALESMAN</td>
<td>7698</td>
<td>20-DEC-10</td>
<td>2600.00</td>
<td>2300.00</td>
<td>30</td>
</tr>
<tr>
<td>9103</td>
<td>WARREN</td>
<td>SALESMAN</td>
<td>7698</td>
<td>22-DEC-10</td>
<td>5250.00</td>
<td>2500.00</td>
<td>30</td>
</tr>
<tr>
<td>9104</td>
<td>JONES, JR.</td>
<td>MANAGER</td>
<td>7839</td>
<td>02-APR-09</td>
<td>7975.00</td>
<td></td>
<td>20</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>empno</th>
<th>ename</th>
<th>job</th>
<th>mgr</th>
<th>hiredate</th>
<th>sal</th>
<th>comm</th>
<th>deptno</th>
</tr>
</thead>
<tbody>
<tr>
<td>9101</td>
<td>ROGERS</td>
<td>CLERK</td>
<td>7902</td>
<td>17-DEC-10</td>
<td>1980.0000</td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>9102</td>
<td>PETERSON</td>
<td>SALESMAN</td>
<td>7698</td>
<td>20-DEC-10</td>
<td>2600.0030</td>
<td>2300.00</td>
<td>30</td>
</tr>
<tr>
<td>9103</td>
<td>WARREN</td>
<td>SALESMAN</td>
<td>7698</td>
<td>22-DEC-10</td>
<td>5250.0030</td>
<td>2500.00</td>
<td>30</td>
</tr>
<tr>
<td>9104</td>
<td>JONES, JR.</td>
<td>MANAGER</td>
<td>7839</td>
<td>02-APR-09</td>
<td>7975.0020</td>
<td></td>
<td>20</td>
</tr>
</tbody>
</table>

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
```

```
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
```
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;
```

<table>
<thead>
<tr>
<th>empno</th>
<th>ename</th>
<th>job</th>
<th>mgr</th>
<th>hiredate</th>
<th>sal</th>
<th>comm</th>
<th>deptno</th>
</tr>
</thead>
<tbody>
<tr>
<td>9101</td>
<td>ROGERS</td>
<td>CLERK</td>
<td>7902</td>
<td>17-DEC-10 00:00:00</td>
<td></td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>9102</td>
<td>PETERSON</td>
<td>SALESMAN</td>
<td>7698</td>
<td>20-DEC-10 00:00:00</td>
<td></td>
<td></td>
<td>30</td>
</tr>
<tr>
<td>9103</td>
<td>WARREN</td>
<td>SALESMAN</td>
<td>7698</td>
<td>22-DEC-10 00:00:00</td>
<td></td>
<td></td>
<td>30</td>
</tr>
<tr>
<td>9104</td>
<td>JONES, JR.</td>
<td>MANAGER</td>
<td>7839</td>
<td>02-APR-09 00:00:00</td>
<td></td>
<td></td>
<td>20</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(4 rows)</td>
</tr>
</tbody>
</table>

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:

```sql
9101ROGERS   CLERK   790217-DEC-10 1980.0020
9102PETERSON SALESMAN 769820-DEC-10 2600.00030 2300.00
9103JONES, JR MANAGER 783902-APR-09 7975.0020
9104ARNOLDS   CLERK   778213-SEP-10 3750.0010
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:

```sql
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:

```sql
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)
```sql
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)
```
11.2.4 Invoking EDB*Loader

You must have superuser privileges to run EDB*Loader. Use the following command to invoke EDB*Loader from the command line:

```bash
edbldr [ -d dbname ] [ -p port ] [ -h host ]
[ USERID={ username/password | username/ | username | / } ]
CONTROL=control_file
[ BAD=bad_file ]
[ DISCARD=discard_file ]
[ LOG=log_file ]
[ PARFILE=param_file ]
[ SKIP=skip_count ]
[ ERRORS=error_count ]
[ SKIP_INDEX_MAINTENANCE={ FALSE | TRUE } ]
[ DIRECT={ FALSE | TRUE } ]
[ PARALLEL={ FALSE | TRUE } ]
```

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 Postgres Plus 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 11.2.3.

The specification of control_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 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.

The operating system account used to invoke edbldr must have read permission on the directories and files specified by control_file and param_file.

The operating system accountentreprise db 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, bad_file, discard_file, and log_file include extensions of .ctl, .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 Postgres Plus Advanced Server connection environment variables PGUSER and PGPASSWORD are ignored by EDB*Loader. See the PostgreSQL core documentation for information on the PGPASSFILE environment variable and the password file.

control_file

Control file containing the directives to be used by EDB*Loader. If a file extension is not provided, an extension of .ctl is assumed. See Section 11.2.3 for a description of the control file.

bad_file
File that receives input data records that cannot be loaded due to errors. See the description of *bad_file* in Section 11.2.3.

**Note:** The specification of *bad_file* using the command line **BAD** parameter overrides the **BADFILE** clause in the control file.

discard_file

File that receives input data records that do not meet any table’s selection criteria. See the description of *discard_file* in Section 11.2.3.

**Note:** The specification of *discard_file* using the command line **DISCARD** parameter overrides the **DISCARDFILE** clause in the control file.

log_file

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 current working 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 enterprise db must have write permission on the directory where the log file is to be written.

param_file

Parameter file which EDB*Loader uses to obtain 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.

skip_count

Number of records at the beginning of the input data file that should be skipped before loading begins. The default is 0.

count


Number of errors permitted before aborting the EDB*Loader session. The default is 50.

\[ \text{SKIP\_INDEX\_MAINTENANCE} = \{ \text{FALSE} | \text{TRUE} \} \]

If set to \text{TRUE}, index maintenance is not performed as part of a direct path load. Indexes on the loaded table are marked as invalid. Use the \text{REINDEX} command to rebuild the indexes. The default is \text{FALSE}.

\[ \text{DIRECT} = \{ \text{FALSE} | \text{TRUE} \} \]

If set to \text{TRUE}, a direct path load is performed instead of a conventional path load. See Section 11.2.5 for information on direct path loads. The default is \text{FALSE}.

\[ \text{PARALLEL} = \{ \text{FALSE} | \text{TRUE} \} \]

If set to \text{TRUE}, it is assumed this EDB*Loader session is one of a number of concurrent EDB*Loader sessions participating in a parallel direct path load. The \text{DIRECT=TRUE} parameter must also be set when \text{PARALLEL=TRUE}. See Section 11.2.6 for information on parallel direct path loads. The default is \text{FALSE}.

\section*{Examples}

In the following example EDB*Loader is invoked using a control file named \text{emp.ctl} located in the current working directory to load a table in database \text{edb}:

\$ /opt/PostgresPlus/9.2AS/bin/edbldr -d edb USERID=enterprisedb/password CONTROL=emp.ctl

EDB*Loader: Copyright (c) 2007-2012, 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 \text{BAD} and \text{LOG} command line parameters.

\$ /opt/PostgresPlus/9.2AS/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) 2007-2012, 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 \text{SKIP} and \text{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/9.2AS/bin/edbldr -d edb PARFILE=emp.par
Enter the user name : enterprisedb
Enter the password :
EDB*Loader: Copyright (c) 2007-2012, EnterpriseDB Corporation.
Successfully loaded (3) records
```
11.2.5 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.

However, 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.

In addition, 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.

To run a direct path load, add the DIRECT=TRUE option as shown by the following example:

```
$ /opt/PostgresPlus/9.2AS/bin/edbldr -d edb USERID=enterprisedb/password CONTROL=emp.ctl DIRECT=TRUE
EDB*Loader: Copyright (c) 2007-2012, EnterpriseDB Corporation.
```
Successfully loaded (4) records
11.2.6 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 Postgres Plus 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/9.2AS/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) 2007-2012, 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/9.2AS/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) 2007-2012, 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 Oracle compatible view of indexes 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)
```
11.2.7 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 edblldr 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 11.2.3 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 edblldr is invoked.
- When invoking EDB*Loader, use the -h option to specify the IP address of the remote database server. See Section 11.2.4 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 | ./edblldr -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:

```
./edblldr -d edb -h 192.168.1.14 USERID=enterprisedb/password
CONTROL=remote.ctl < datasource
```
11.3 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 the PostgreSQL server and the server 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 EDBWrap 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.
11.3.1 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 compatible with Oracle's wrap utility:

```
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 the PostgreSQL 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:

1. Create the source code file.
2. Invoke EDB*Wrap to obfuscate the code.
3. 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` procedure (in plaintext form):

```
[bash] cat listemp.sql
CREATE OR REPLACE PROCEDURE list_emp
```
IS
v_empno NUMBER(4);
v_ename VARCHAR2(10);
CURSOR emp_cur IS
  SELECT empno, ename FROM emp ORDER BY empno;
BEGIN
  OPEN emp_cur;
  DBMS_OUTPUT.PUT_LINE('EMPNO  ENAME');
  DBMS_OUTPUT.PUT_LINE('-----  ------');
  LOOP
    FETCH emp_cur INTO v_empno, v_ename;
    EXIT WHEN emp_cur%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE(v_empno || '   ' || v_ename);
  END LOOP;
  CLOSE emp_cur;
END;
/

You can import the list_emp procedure with a client application such as edb-psql:

[bash] edb-psql edb
Welcome to edb-psql 8.4.3.2, the EnterpriseDB interactive terminal.

Type: \copyright for distribution terms
\h for help with SQL commands
? for help with edb-psql commands
\g or terminate with semicolon to execute query
\q to quit

edb=# \i listemp.sql
CREATE PROCEDURE

You can view the plaintext source code (stored in the server) by examining the pg_proc system table:

edb=# SELECT prosrc FROM pg_proc WHERE proname = 'list_emp';
prosrc
--------------------------------------------------------------
v_empno NUMBER(4);
v_ename VARCHAR2(10);
CURSOR emp_cur IS
  SELECT empno, ename FROM emp ORDER BY empno;
BEGIN
  OPEN emp_cur;
  DBMS_OUTPUT.PUT_LINE('EMPNO  ENAME');
  DBMS_OUTPUT.PUT_LINE('-----  ------');
  LOOP
    FETCH emp_cur INTO v_empno, v_ename;
    EXIT WHEN emp_cur%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE(v_empno || '   ' || v_ename);
  END LOOP;
  CLOSE emp_cur;
END
(1 row)

edb=# quit

Next, obfuscate the plaintext file with EDB*Wrap:
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 the PostgreSQL server using the same tools that work with plaintext code:

```bash
[bash] edb-psql edb
Welcome to edb-psql 8.4.3.2, the EnterpriseDB interactive terminal.
Type: \copyright for distribution terms
     \h for help with SQL commands
     \? for help with edb-psql commands
     \g or terminate with semicolon to execute query
     \q to quit
edb=# \i listemp.plb
CREATE PROCEDURE
Now, the pg_proc system table contains the obfuscated code:

edb=# SELECT prosrc FROM pg_proc WHERE proname = 'list_emp';
prosrc
$\_EDBwrapped\$
UTF8
dw4B9Tz69J3WOsy0GgYJQa+G2sLz31OyxS8pDyu0TPuiYe/EXiEatwwG3h3tdJk
ea+Ap35dS/4i0bN8weqem43s994dQ3R97NHyvTQnO2vtd4wQtsQ/2c4v4Lhfj
n1v+4UpHIs0q5EmexArch2LcRD87hkU0uc1EseQV8IrXaj9Bz2r+ueROwHgs/EC
pva/rV4m9RusFn0vyr38u42w4dfnFW18Y3o6I4t3h3A07WxTkWrMlmo2W1jJ
Nu6u4o+ez064G9QKpxgehsal4JB9NQnuocAnfDSqMP7R7anmgw
$\_EDBwrapped\$
(1 row)
```
Invoke the obfuscated code in the same way that you would invoke the plaintext form:

```
edb=# exec list_emp;

EMPNO  ENAME
------- -------
7369    SMITH
7499    ALLEN
7521    WARD
7566    JONES
7654    MARTIN
7698    BLAKE
7782    CLARK
7788    SCOTT
7839    KING
7844    TURNER
7876    ADAMS
7900    JAMES
7902    FORD
7934    MILLER

EDB-SPL Procedure successfully completed
edb=# quit
```

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 the Postgres 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
```
11.4 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 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.

DRITA consumes minimal system resources.

11.4.1 Initialization Parameters

DRITA includes a configuration parameter, timed_statistics, to control the collection of timing data. This is a dynamic parameter that can be set in the postgresql.conf file or while a session is in progress. The valid values are TRUE or FALSE; the default value is FALSE.

11.4.2 Setting up and Using DRITA

First, take a beginning snapshot. The beginning snapshot will be compared to a later snapshot to gauge system performance. To take a beginning snapshot:

```
SELECT * from edbsnap()
```

Then, run the workload that you would like to evaluate; when the workload has completed (or at a strategic point during the workload), take an ending snapshot:

```
SELECT * from edbsnap()
```
11.5 DRITA Functions

11.5.1 get_snaps()

The get_snaps() function returns a list of the current snapshots. The signature is:

```sql
get_snaps()
```

The following example demonstrates using the get_snaps() function to display a list of snapshots:

```sql
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 list of snapshots contains the session identifier; the DRITA functions use the session identifier to operate on a specific snapshot.

11.5.2 sys_rpt()

The sys_rpt() function returns system wait information. The signature is:

```sql
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`
This example demonstrates a call to the `sys_rpt()` function:

```
edb=# SELECT * FROM sys_rpt(9, 10, 10);
```

<table>
<thead>
<tr>
<th>WAIT NAME</th>
<th>COUNT</th>
<th>WAIT TIME</th>
<th>% WAIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>wal write</td>
<td>21250</td>
<td>104.723772</td>
<td>36.31</td>
</tr>
<tr>
<td>db file read</td>
<td>121407</td>
<td>72.143274</td>
<td>25.01</td>
</tr>
<tr>
<td>wal flush</td>
<td>84185</td>
<td>51.652495</td>
<td>17.91</td>
</tr>
<tr>
<td>wal file sync</td>
<td>712</td>
<td>29.482206</td>
<td>10.22</td>
</tr>
<tr>
<td>infinitecache write</td>
<td>84178</td>
<td>15.844444</td>
<td>5.48</td>
</tr>
<tr>
<td>db file write</td>
<td>84177</td>
<td>14.447718</td>
<td>5.01</td>
</tr>
<tr>
<td>infinitecache read</td>
<td>672</td>
<td>0.098691</td>
<td>0.03</td>
</tr>
<tr>
<td>db file extend</td>
<td>63</td>
<td>0.024000</td>
<td>0.01</td>
</tr>
<tr>
<td>query plan</td>
<td>52</td>
<td>0.000837</td>
<td>0.00</td>
</tr>
<tr>
<td>wal insert lock acquire</td>
<td>4</td>
<td>0.000837</td>
<td>0.00</td>
</tr>
</tbody>
</table>
(12 rows)

11.5.3 `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);
```

<table>
<thead>
<tr>
<th>ID</th>
<th>USER</th>
<th>WAIT NAME</th>
<th>COUNT</th>
<th>TIME(ms)</th>
<th>%WAIT SES</th>
<th>%WAIT ALL</th>
</tr>
</thead>
<tbody>
<tr>
<td>17373</td>
<td>enterprise</td>
<td>db file read</td>
<td>30</td>
<td>0.175713</td>
<td>85.24</td>
<td>85.24</td>
</tr>
</tbody>
</table>
11.5.4 **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);
```

<table>
<thead>
<tr>
<th>ID</th>
<th>USER</th>
<th>WAIT NAME</th>
<th>COUNT</th>
<th>TIME(ms)</th>
<th>%WAIT SES</th>
<th>%WAIT ALL</th>
</tr>
</thead>
<tbody>
<tr>
<td>17373</td>
<td></td>
<td>enterprise query plan</td>
<td>18</td>
<td>0.014930</td>
<td>7.24</td>
<td>7.24</td>
</tr>
<tr>
<td>17373</td>
<td></td>
<td>enterprise wal flush</td>
<td>6</td>
<td>0.004067</td>
<td>1.97</td>
<td>1.97</td>
</tr>
<tr>
<td>17373</td>
<td></td>
<td>enterprise wal write</td>
<td>1</td>
<td>0.004063</td>
<td>1.97</td>
<td>1.97</td>
</tr>
<tr>
<td>17373</td>
<td></td>
<td>enterprise wal file sync</td>
<td>1</td>
<td>0.003664</td>
<td>1.78</td>
<td>1.78</td>
</tr>
<tr>
<td>17373</td>
<td></td>
<td>enterprise infinitecache read</td>
<td>38</td>
<td>0.003076</td>
<td>1.49</td>
<td>1.49</td>
</tr>
<tr>
<td>17373</td>
<td></td>
<td>enterprise infinitecache write</td>
<td>5</td>
<td>0.000548</td>
<td>0.27</td>
<td>0.27</td>
</tr>
<tr>
<td>17373</td>
<td></td>
<td>enterprise db file extend</td>
<td>190</td>
<td>0.04386</td>
<td>0.03</td>
<td>0.03</td>
</tr>
<tr>
<td>17373</td>
<td></td>
<td>enterprise db file write</td>
<td>5</td>
<td>0.000082</td>
<td>0.04</td>
<td>0.04</td>
</tr>
<tr>
<td>17373</td>
<td></td>
<td>enterprise wal write lock acquire</td>
<td>0</td>
<td>0.000000</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>17373</td>
<td></td>
<td>enterprise bgwriter comm lock acquire</td>
<td>0</td>
<td>0.000000</td>
<td>0.00</td>
<td>0.00</td>
</tr>
</tbody>
</table>

(13 rows)
11.5.5 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);
```
11.5.6 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
```

```
(27 rows)
```
11.5.7  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)
11.6 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.

11.6.1 `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 following code sample demonstrates a call to the `edbreport()` function:

```
edb=# SELECT * FROM edbreport(9, 10);
```

```
edbreport
--------------------------------------------------------------------------------
EnterpriseDB Report for database edb    23-AUG-12
Version: EnterpriseDB 9.2.0.0 on i686-pc-linux-gnu, compiled by gcc (GCC)
4.1.2 20080704 (Red Hat 4.1.2-52), 32-bit

Begin snapshot: 9 at 23-AUG-12 13:45:07.165123
End snapshot:   10 at 23-AUG-12 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

<table>
<thead>
<tr>
<th>Schema</th>
<th>Size:</th>
<th>Owner:</th>
</tr>
</thead>
<tbody>
<tr>
<td>pg toast temp 1</td>
<td>0 bytes</td>
<td>enterprisedb</td>
</tr>
<tr>
<td>public</td>
<td>0 bytes</td>
<td>enterprisedb</td>
</tr>
<tr>
<td>enterprise db</td>
<td>143 MB</td>
<td>enterprisedb</td>
</tr>
<tr>
<td>pgagent</td>
<td>192 kB</td>
<td>enterprisedb</td>
</tr>
<tr>
<td>dbms_job_procedure</td>
<td>0 bytes</td>
<td>enterprisedb</td>
</tr>
</tbody>
</table>

Top 10 Relations by pages

<table>
<thead>
<tr>
<th>TABLE</th>
<th>RELPAGES</th>
</tr>
</thead>
<tbody>
<tr>
<td>pgbench_accounts</td>
<td>15874</td>
</tr>
<tr>
<td>pg_proc</td>
<td>102</td>
</tr>
<tr>
<td>edb$statio_all_indexes</td>
<td>73</td>
</tr>
<tr>
<td>edb$stat_all_indexes</td>
<td>73</td>
</tr>
<tr>
<td>pg_attribute</td>
<td>67</td>
</tr>
<tr>
<td>pg_depend</td>
<td>58</td>
</tr>
<tr>
<td>edb$statio_all_tables</td>
<td>49</td>
</tr>
<tr>
<td>edb$stat_all_tables</td>
<td>47</td>
</tr>
<tr>
<td>pgbench_tellers</td>
<td>37</td>
</tr>
<tr>
<td>pg_description</td>
<td>32</td>
</tr>
</tbody>
</table>

Top 10 Indexes by pages

<table>
<thead>
<tr>
<th>INDEX</th>
<th>RELPAGES</th>
</tr>
</thead>
<tbody>
<tr>
<td>pgbench_accounts_pkey</td>
<td>2198</td>
</tr>
<tr>
<td>pg_depend_depender_index</td>
<td>32</td>
</tr>
<tr>
<td>pg_depend_reference_index</td>
<td>31</td>
</tr>
<tr>
<td>pg_proc_proname_args_nsp_index</td>
<td>30</td>
</tr>
<tr>
<td>pg_attribute_relid_attnam_index</td>
<td>23</td>
</tr>
<tr>
<td>pg_attribute_relid_attnum_index</td>
<td>17</td>
</tr>
<tr>
<td>pg_description_o_c_o_index</td>
<td>15</td>
</tr>
<tr>
<td>edb$statio_idx_pk</td>
<td>11</td>
</tr>
<tr>
<td>edb$stat_idx_pk</td>
<td>11</td>
</tr>
<tr>
<td>pg_proc_oid_index</td>
<td>9</td>
</tr>
</tbody>
</table>

Top 10 Relations by DML

<table>
<thead>
<tr>
<th>SCHEMA</th>
<th>RELATION</th>
<th>UPDATES</th>
</tr>
</thead>
<tbody>
<tr>
<td>enterprise db</td>
<td>pgbench_accounts</td>
<td>10400</td>
</tr>
<tr>
<td></td>
<td>pgbench_tellers</td>
<td>10400</td>
</tr>
<tr>
<td></td>
<td>pgbench_branches</td>
<td>10400</td>
</tr>
<tr>
<td></td>
<td>pgbench_history</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>pga_jobclass</td>
<td>0</td>
</tr>
</tbody>
</table>

Copyright © 2007 - 2015 EnterpriseDB Corporation. All rights reserved.
### DATA from pg_stat_database

<table>
<thead>
<tr>
<th>DATABASE</th>
<th>NUMBACKENDS</th>
<th>XACT COMMIT</th>
<th>XACT ROLLBACK</th>
<th>BLKS READ</th>
<th>BLKS HIT</th>
<th>BLKS ICACHE HIT</th>
<th>HIT RATIO</th>
<th>ICACHE HIT RATIO</th>
</tr>
</thead>
<tbody>
<tr>
<td>edb</td>
<td>0</td>
<td>142</td>
<td>0</td>
<td>78</td>
<td>10446</td>
<td>0</td>
<td>99.26</td>
<td>0.00</td>
</tr>
</tbody>
</table>

DATA from pg_buffercache not included because pg_buffercache is not installed

DATA from pg_stat_all_tables ordered by seq scan

<table>
<thead>
<tr>
<th>SCHEMA</th>
<th>RELATION</th>
<th>SEQ SCAN</th>
<th>REL TUP READ</th>
</tr>
</thead>
<tbody>
<tr>
<td>pg_catalog</td>
<td>pg_class</td>
<td>16</td>
<td>7162</td>
</tr>
<tr>
<td>546</td>
<td>319</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>pg_am</td>
<td>13</td>
<td>13</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>pg_database</td>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td>42</td>
<td>42</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>pg_index</td>
<td>4</td>
<td>660</td>
</tr>
<tr>
<td>145</td>
<td>149</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>pg_namespace</td>
<td>4</td>
<td>100</td>
</tr>
<tr>
<td>49</td>
<td>49</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sys</td>
<td>edb$snap</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>pg_authid</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>25</td>
<td>25</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sys</td>
<td>edb$session_wait_history</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>50</td>
<td>0</td>
</tr>
<tr>
<td>sys</td>
<td>edb$session_waits</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>sys</td>
<td>edb$stat_all_indexes</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>165</td>
<td>0</td>
</tr>
</tbody>
</table>

DATA from pg_stat_all_tables ordered by rel tup read

<table>
<thead>
<tr>
<th>SCHEMA</th>
<th>RELATION</th>
<th>SEQ SCAN</th>
<th>REL TUP READ</th>
</tr>
</thead>
<tbody>
<tr>
<td>pg_catalog</td>
<td>pg_class</td>
<td>16</td>
<td>7162</td>
</tr>
<tr>
<td>546</td>
<td>319</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>pg_index</td>
<td>4</td>
<td>660</td>
</tr>
<tr>
<td>145</td>
<td>149</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>pg_namespace</td>
<td>4</td>
<td>100</td>
</tr>
<tr>
<td>49</td>
<td>49</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
### DATA from pg_statio_all_tables

<table>
<thead>
<tr>
<th>SCHEMA</th>
<th>RELATION</th>
<th>HEAP</th>
<th>HEAP</th>
<th>HEAP</th>
<th>IDX</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>READ</td>
<td>HIT</td>
<td>ICACHE</td>
<td>READ</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>pg_class</td>
<td>0</td>
<td>539</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>enterprisedb</td>
<td>pgbench_accounts</td>
<td>48</td>
<td>485</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>pg_attribute</td>
<td>0</td>
<td>447</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>enterprisedb</td>
<td>pgbench_branches</td>
<td>0</td>
<td>439</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>enterprisedb</td>
<td>pgbench_tellers</td>
<td>0</td>
<td>357</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>pg_statistic</td>
<td>1</td>
<td>293</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>pg_index</td>
<td>0</td>
<td>159</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>pg_opclass</td>
<td>0</td>
<td>145</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>pg_proc</td>
<td>0</td>
<td>135</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>pg_type</td>
<td>0</td>
<td>103</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

### DATA from pg_stat_all_indexes

<table>
<thead>
<tr>
<th>SCHEMA</th>
<th>RELATION</th>
<th>INDEX</th>
</tr>
</thead>
<tbody>
<tr>
<td>pg_catalog</td>
<td>pg_attribute</td>
<td></td>
</tr>
<tr>
<td>pg_attribute_relid_attnum_index</td>
<td>427</td>
<td>907</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>pg_class</td>
<td></td>
</tr>
<tr>
<td>pg_class_relname_nsp_index</td>
<td>289</td>
<td>62</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>pg_class</td>
<td></td>
</tr>
<tr>
<td>pg_class_oid_index</td>
<td>257</td>
<td>257</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>pg_statistic</td>
<td></td>
</tr>
<tr>
<td>pg_statistic_relid_attr_index</td>
<td>207</td>
<td>196</td>
</tr>
<tr>
<td>enterprisedb</td>
<td>pgbench_accounts</td>
<td></td>
</tr>
<tr>
<td>pgbench_accounts_pkey</td>
<td>200</td>
<td>200</td>
</tr>
</tbody>
</table>
### DATA from pg_statio_all_indexes

<table>
<thead>
<tr>
<th>SCHEMA</th>
<th>RELATION</th>
<th>INDEX</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDX BLKS READ</td>
<td>IDX BLKS HIT</td>
<td>IDX BLKS ICACHE HIT</td>
</tr>
<tr>
<td>---------------------</td>
<td>-----------------------------</td>
<td>------------------------</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>pg_attribute</td>
<td></td>
</tr>
<tr>
<td>199</td>
<td>50</td>
<td>pg_cast_source_target_index 50</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>pg_proc</td>
<td></td>
</tr>
<tr>
<td>116</td>
<td>116</td>
<td>pg_proc_oid_index       116</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>edb_partition</td>
<td></td>
</tr>
<tr>
<td>112</td>
<td>0</td>
<td>edb_partition_partrelid_index 0</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>edb_policy</td>
<td></td>
</tr>
<tr>
<td>112</td>
<td>0</td>
<td>edb_policy_object_name_index 0</td>
</tr>
<tr>
<td>enterprisedb</td>
<td>pgbench_branches</td>
<td></td>
</tr>
<tr>
<td>101</td>
<td>110</td>
<td>pgbench_branches_pkey   110</td>
</tr>
</tbody>
</table>

### System Wait Information

<table>
<thead>
<tr>
<th>WAIT NAME</th>
<th>COUNT</th>
<th>WAIT TIME</th>
<th>% WAIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>query plan</td>
<td>0</td>
<td>0.000407</td>
<td>100.00</td>
</tr>
<tr>
<td>db file read</td>
<td>0</td>
<td>0.000000</td>
<td>0.00</td>
</tr>
</tbody>
</table>

### Database Parameters from postgresql.conf

<table>
<thead>
<tr>
<th>PARAMETER</th>
<th>SETTING</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONTEXT</td>
<td>MINVAL</td>
</tr>
<tr>
<td>-------------------------</td>
<td>---------</td>
</tr>
<tr>
<td>allow_system_table_mods</td>
<td>off</td>
</tr>
<tr>
<td>postmaster</td>
<td>psql</td>
</tr>
<tr>
<td>archive_command</td>
<td>(disabled)</td>
</tr>
<tr>
<td>archive_mode</td>
<td>off</td>
</tr>
<tr>
<td>archive_timeout</td>
<td>0</td>
</tr>
<tr>
<td>array_nulls</td>
<td>on</td>
</tr>
<tr>
<td>user</td>
<td></td>
</tr>
<tr>
<td>Parameter</td>
<td>Value</td>
</tr>
<tr>
<td>------------------------</td>
<td>-----------</td>
</tr>
<tr>
<td>authentication_timeout</td>
<td>60</td>
</tr>
<tr>
<td>sighup</td>
<td>1</td>
</tr>
<tr>
<td>autovacuum</td>
<td>on</td>
</tr>
<tr>
<td>sighup</td>
<td></td>
</tr>
<tr>
<td>autovacuum_analyze_scale_factor</td>
<td>0.1</td>
</tr>
<tr>
<td>sighup</td>
<td>0</td>
</tr>
<tr>
<td>autovacuum_analyze_threshold</td>
<td>50</td>
</tr>
<tr>
<td>sighup</td>
<td>0</td>
</tr>
<tr>
<td>autovacuum_freeze_max_age</td>
<td>20000000</td>
</tr>
<tr>
<td>sighup</td>
<td></td>
</tr>
<tr>
<td>autovacuum_max_workers</td>
<td>3</td>
</tr>
<tr>
<td>postmaster</td>
<td>1</td>
</tr>
<tr>
<td>autovacuum_naptime</td>
<td>60</td>
</tr>
<tr>
<td>sighup</td>
<td>1</td>
</tr>
<tr>
<td>autovacuum_vacuum_cost_delay</td>
<td>20</td>
</tr>
<tr>
<td>sighup</td>
<td>-1</td>
</tr>
<tr>
<td>autovacuum_vacuum_cost_limit</td>
<td>-1</td>
</tr>
<tr>
<td>sighup</td>
<td>-1</td>
</tr>
<tr>
<td>autovacuum_vacuum_scale_factor</td>
<td>0.2</td>
</tr>
<tr>
<td>sighup</td>
<td>0</td>
</tr>
<tr>
<td>autovacuum_vacuum_threshold</td>
<td>50</td>
</tr>
<tr>
<td>sighup</td>
<td>0</td>
</tr>
<tr>
<td>backslash_quote</td>
<td>safe_encoding</td>
</tr>
<tr>
<td>user</td>
<td></td>
</tr>
<tr>
<td>bgwriter_delay</td>
<td>200</td>
</tr>
<tr>
<td>sighup</td>
<td>10</td>
</tr>
<tr>
<td>bgwriter_lru_maxpages</td>
<td>100</td>
</tr>
<tr>
<td>sighup</td>
<td>0</td>
</tr>
<tr>
<td>bgwriter_lru_multiplier</td>
<td>2</td>
</tr>
<tr>
<td>sighup</td>
<td>0</td>
</tr>
<tr>
<td>block_size</td>
<td>8192</td>
</tr>
<tr>
<td>internal</td>
<td>8192</td>
</tr>
<tr>
<td>bonjour</td>
<td>off</td>
</tr>
<tr>
<td>postmaster</td>
<td></td>
</tr>
<tr>
<td>bonjour_name</td>
<td></td>
</tr>
<tr>
<td>postmaster</td>
<td></td>
</tr>
<tr>
<td>bytea_output</td>
<td>hex</td>
</tr>
<tr>
<td>user</td>
<td></td>
</tr>
<tr>
<td>check_function_bodies</td>
<td>on</td>
</tr>
<tr>
<td>user</td>
<td></td>
</tr>
<tr>
<td>checkpoint_completion_target</td>
<td>0.5</td>
</tr>
<tr>
<td>sighup</td>
<td>0</td>
</tr>
<tr>
<td>checkpoint_segments</td>
<td>1</td>
</tr>
<tr>
<td>sighup</td>
<td></td>
</tr>
<tr>
<td>checkpoint_timeout</td>
<td>2147483647</td>
</tr>
<tr>
<td>sighup</td>
<td>300</td>
</tr>
<tr>
<td>checkpoint_warning</td>
<td>30</td>
</tr>
<tr>
<td>sighup</td>
<td>2147483647</td>
</tr>
<tr>
<td>client_encoding</td>
<td>UTF8</td>
</tr>
<tr>
<td>user</td>
<td></td>
</tr>
<tr>
<td>client_min_messages</td>
<td>notice</td>
</tr>
<tr>
<td>user</td>
<td></td>
</tr>
<tr>
<td>commit_delay</td>
<td>0</td>
</tr>
<tr>
<td>user</td>
<td>0</td>
</tr>
<tr>
<td>commit_siblings</td>
<td>5</td>
</tr>
<tr>
<td>user</td>
<td>0</td>
</tr>
<tr>
<td>cpu_index_tuple_cost</td>
<td>0.005</td>
</tr>
<tr>
<td>user</td>
<td>1.79769e+308</td>
</tr>
<tr>
<td>config_file</td>
<td>/opt/PostgresPlus/9.2AS/data/postgresql.</td>
</tr>
<tr>
<td>postmaster</td>
<td></td>
</tr>
<tr>
<td>constraint_exclusion</td>
<td>partition</td>
</tr>
<tr>
<td>user</td>
<td></td>
</tr>
<tr>
<td>config_file</td>
<td></td>
</tr>
</tbody>
</table>
cpu_operator_cost                      0.0025
user                                  0    1.79769e+308
cpu_tuple_cost                       0.01
user                                  0    1.79769e+308
cursor_tuple_fraction                 0.1
user                                  0    1

data_directory                       /opt/PostgresPlus/9.2AS/data
postmaster                            DateStyle  Redwood, SHOW_TIME
user                                  db_dialect  redwood
user                                  dbms_alert.max_alerts  100
postmaster                            dbsms_pipe.total_message_buffer  30
postmaster                            db_user_namespace  off
sighup                                deadlock_timeout  1000
superuser                              debug_assertions  off
user                                  debug_pretty_print  on
user                                  debug_print_parse  off
user                                  debug_print_plan  off
user                                  debug_print_rewritten  off
user                                  default_heap_fillfactor  100
user                                  default_statistics_target  100
user                                  default_tablespace
user                                  default_text_search_config  pg_catalog.english
user                                  default_transaction_deferrable  off
user                                  default_transaction_isolation  read committed
user                                  default_transaction_read_only  off
user                                  default_with_oids  off
user                                  default_with_rowids  off
user                                  dynamic_library_path  $libdir
superuser                              edb_audit
sighup                                edb_audit_connect  failed
sighup                                edb_audit_directory  edb_audit
sighup                                edb_audit_disconnect  none
sighup                                edb_audit_filename  audit-%Y%m%d_%H%M%S
sighup                                edb_audit_rotation_day  every
<table>
<thead>
<tr>
<th>Configuration</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>edb_audit_rotation_seconds</code></td>
<td>0</td>
</tr>
<tr>
<td><code>sighup</code></td>
<td>0 2147483647</td>
</tr>
<tr>
<td><code>edb_audit_rotation_size</code></td>
<td>0 5000</td>
</tr>
<tr>
<td><code>sighup</code></td>
<td>ddl, error</td>
</tr>
<tr>
<td><code>edb_connectby_order</code></td>
<td>on</td>
</tr>
<tr>
<td><code>user</code></td>
<td></td>
</tr>
<tr>
<td><code>edb_dynatune</code></td>
<td>66</td>
</tr>
<tr>
<td><code>postmaster</code></td>
<td>0 100</td>
</tr>
<tr>
<td><code>edb_dynatune_profile</code></td>
<td>oltp</td>
</tr>
<tr>
<td><code>postmaster</code></td>
<td></td>
</tr>
<tr>
<td><code>edb_enable_icache</code></td>
<td>off</td>
</tr>
<tr>
<td><code>postmaster</code></td>
<td></td>
</tr>
<tr>
<td><code>edb_icache_compression_level</code></td>
<td>6</td>
</tr>
<tr>
<td><code>superuser</code></td>
<td>0 9</td>
</tr>
<tr>
<td><code>edb_icache_servers</code></td>
<td></td>
</tr>
<tr>
<td><code>sighup</code></td>
<td></td>
</tr>
<tr>
<td><code>edb_redwood_date</code></td>
<td>on</td>
</tr>
<tr>
<td><code>user</code></td>
<td></td>
</tr>
<tr>
<td><code>edb_redwood_strings</code></td>
<td>on</td>
</tr>
<tr>
<td><code>user</code></td>
<td></td>
</tr>
<tr>
<td><code>edb_stmt_level_tx</code></td>
<td>off</td>
</tr>
<tr>
<td><code>user</code></td>
<td></td>
</tr>
<tr>
<td><code>effective_cache_size</code></td>
<td>34277</td>
</tr>
<tr>
<td><code>user</code></td>
<td>1 2147483647</td>
</tr>
<tr>
<td><code>effective_io_concurrency</code></td>
<td>1 1000</td>
</tr>
<tr>
<td><code>user</code></td>
<td></td>
</tr>
<tr>
<td><code>enable_bitmapscan</code></td>
<td>on</td>
</tr>
<tr>
<td><code>user</code></td>
<td></td>
</tr>
<tr>
<td><code>enable_hashagg</code></td>
<td>on</td>
</tr>
<tr>
<td><code>user</code></td>
<td></td>
</tr>
<tr>
<td><code>enable_hashjoin</code></td>
<td>on</td>
</tr>
<tr>
<td><code>user</code></td>
<td></td>
</tr>
<tr>
<td><code>enable_hints</code></td>
<td>on</td>
</tr>
<tr>
<td><code>user</code></td>
<td></td>
</tr>
<tr>
<td><code>enable_indexonlyscan</code></td>
<td>on</td>
</tr>
<tr>
<td><code>user</code></td>
<td></td>
</tr>
<tr>
<td><code>enable_indexscan</code></td>
<td>on</td>
</tr>
<tr>
<td><code>user</code></td>
<td></td>
</tr>
<tr>
<td><code>enable_material</code></td>
<td>on</td>
</tr>
<tr>
<td><code>user</code></td>
<td></td>
</tr>
<tr>
<td><code>enable_mergejoin</code></td>
<td>on</td>
</tr>
<tr>
<td><code>user</code></td>
<td></td>
</tr>
<tr>
<td><code>enable_nestloop</code></td>
<td>on</td>
</tr>
<tr>
<td><code>user</code></td>
<td></td>
</tr>
<tr>
<td><code>enable_seqscan</code></td>
<td>on</td>
</tr>
<tr>
<td><code>user</code></td>
<td></td>
</tr>
<tr>
<td><code>enable_sort</code></td>
<td>on</td>
</tr>
<tr>
<td><code>user</code></td>
<td></td>
</tr>
<tr>
<td><code>enable_tidscan</code></td>
<td>on</td>
</tr>
<tr>
<td><code>user</code></td>
<td></td>
</tr>
<tr>
<td><code>escape_string_warning</code></td>
<td>on</td>
</tr>
<tr>
<td><code>user</code></td>
<td></td>
</tr>
<tr>
<td><code>event_source</code></td>
<td>PostgreSQL</td>
</tr>
<tr>
<td><code>postmaster</code></td>
<td></td>
</tr>
<tr>
<td><code>exit_on_error</code></td>
<td>off</td>
</tr>
<tr>
<td><code>user</code></td>
<td></td>
</tr>
<tr>
<td><code>external_pid_file</code></td>
<td></td>
</tr>
<tr>
<td><code>postmaster</code></td>
<td></td>
</tr>
<tr>
<td><code>extra_float_digits</code></td>
<td>-15 3</td>
</tr>
<tr>
<td><code>user</code></td>
<td></td>
</tr>
</tbody>
</table>
fromCollapseLimit 8
user 1 2147483647
fsync on
sighup
fullPageWrites on
sighup
ggeo on
user
ggeoEffort 5
user 1 10
ggeoGenerations 0
user 0 2147483647
ggeoPoolSize 0
user 0 2147483647
ggeoSeed 0
user 0 1
ggeoSelectionBias 2
user 1.5 2
ggeoThreshold 12
user 2 2147483647
ginFuzzySearchLimit 0
user 0 2147483647
hbaFile /opt/PostgresPlus/9.2AS/data/pg_hba.conf
postmaster
hotStandby off
postmaster
hotStandbyFeedback off
sighup
identFile /opt/PostgresPlus/9.2AS/data/pg_ident.conf
postmaster
ignoreSystemIndexes off
backend
integerDatetimes on
internal
IntervalStyle postgres
user
joinCollapseLimit 8
user 1 2147483647
krbCaseinsUsers off
sighup
krbServerKeyfile FILE:/home/edb/AS92/edb-postgres/inst/et
sighup
krbSrvname postgres
sighup
lcCollate en_US.UTF-8
internal
lcType en_US.UTF-8
internal
lcMessages en_US.UTF-8
superuser
lcMonetary en_US.UTF-8
user
lcNumeric en_US.UTF-8
user
lcTime en_US.UTF-8
user
listenAddresses *
postmaster
localPreloadLibraries
backend
loCompatPrivileges off
superuser
<table>
<thead>
<tr>
<th>Configuration</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>log_autovacuum_min_duration</td>
<td>-1</td>
</tr>
<tr>
<td>log_checkpoints</td>
<td>off</td>
</tr>
<tr>
<td>log_connections</td>
<td>off</td>
</tr>
<tr>
<td>log_destination</td>
<td>stderr</td>
</tr>
<tr>
<td>log_directory</td>
<td>pg_log</td>
</tr>
<tr>
<td>log_disconnections</td>
<td>off</td>
</tr>
<tr>
<td>log_duration</td>
<td>off</td>
</tr>
<tr>
<td>log_error_verbosity</td>
<td>default</td>
</tr>
<tr>
<td>log_executor_stats</td>
<td>off</td>
</tr>
<tr>
<td>log_file_mode</td>
<td>0600</td>
</tr>
<tr>
<td>log_filename</td>
<td>enterprisedb-%Y-%m-%d_%H%M%S.log</td>
</tr>
<tr>
<td>logging_collector</td>
<td>on</td>
</tr>
<tr>
<td>log_hostname</td>
<td>off</td>
</tr>
<tr>
<td>log_line_prefix</td>
<td>%t</td>
</tr>
<tr>
<td>log_lock_waits</td>
<td>off</td>
</tr>
<tr>
<td>log_min_duration_statement</td>
<td>-1</td>
</tr>
<tr>
<td>log_min_error_statement</td>
<td>error</td>
</tr>
<tr>
<td>log_min_messages</td>
<td>warning</td>
</tr>
<tr>
<td>log_parser_stats</td>
<td>off</td>
</tr>
<tr>
<td>log_planner_stats</td>
<td>off</td>
</tr>
<tr>
<td>log_rotation_age</td>
<td>1440</td>
</tr>
<tr>
<td>log_rotation_size</td>
<td>35791394</td>
</tr>
<tr>
<td>log_statement</td>
<td>none</td>
</tr>
<tr>
<td>log_statement_stats</td>
<td>off</td>
</tr>
<tr>
<td>log_temp_files</td>
<td>-1</td>
</tr>
<tr>
<td>log_timezone</td>
<td>US/Eastern</td>
</tr>
<tr>
<td>log_truncate_on_rotation</td>
<td>off</td>
</tr>
<tr>
<td>maintenance_work_mem</td>
<td>36871</td>
</tr>
<tr>
<td>max_connections</td>
<td>100</td>
</tr>
<tr>
<td>max_files_per_process</td>
<td>1000</td>
</tr>
<tr>
<td>max_function_args</td>
<td>256</td>
</tr>
<tr>
<td>internal</td>
<td>256</td>
</tr>
<tr>
<td>user</td>
<td>1024</td>
</tr>
<tr>
<td>postmaster</td>
<td>8388607</td>
</tr>
<tr>
<td>postmaster</td>
<td>2147483647</td>
</tr>
<tr>
<td>postmaster</td>
<td>25</td>
</tr>
<tr>
<td>postmaster</td>
<td>2147483647</td>
</tr>
<tr>
<td>postmaster</td>
<td>25</td>
</tr>
<tr>
<td><strong>Parameter</strong></td>
<td><strong>Value</strong></td>
</tr>
<tr>
<td>--------------</td>
<td>-----------</td>
</tr>
<tr>
<td>max_identifier_length</td>
<td>63</td>
</tr>
<tr>
<td>internal</td>
<td>63</td>
</tr>
<tr>
<td>max_index_keys</td>
<td>32</td>
</tr>
<tr>
<td>internal</td>
<td>32</td>
</tr>
<tr>
<td>max_locks_per_transaction</td>
<td>64</td>
</tr>
<tr>
<td>postmaster</td>
<td>10</td>
</tr>
<tr>
<td>2147483647</td>
<td></td>
</tr>
<tr>
<td>max_pred_locks_per_transaction</td>
<td>64</td>
</tr>
<tr>
<td>postmaster</td>
<td>10</td>
</tr>
<tr>
<td>2147483647</td>
<td></td>
</tr>
<tr>
<td>max_prepared_transactions</td>
<td>0</td>
</tr>
<tr>
<td>postmaster</td>
<td>0</td>
</tr>
<tr>
<td>max_stack_depth</td>
<td>2048</td>
</tr>
<tr>
<td>superuser</td>
<td>100</td>
</tr>
<tr>
<td>max_standby_archive_delay</td>
<td>30000</td>
</tr>
<tr>
<td>sighup</td>
<td>-1</td>
</tr>
<tr>
<td>2147483647</td>
<td></td>
</tr>
<tr>
<td>max_standby_streaming_delay</td>
<td>30000</td>
</tr>
<tr>
<td>sighup</td>
<td>-1</td>
</tr>
<tr>
<td>2147483647</td>
<td></td>
</tr>
<tr>
<td>max_wal_senders</td>
<td>0</td>
</tr>
<tr>
<td>postmaster</td>
<td>0</td>
</tr>
<tr>
<td>odbc_lib_path</td>
<td>8388607</td>
</tr>
<tr>
<td>postmaster</td>
<td>optimizer_mode</td>
</tr>
<tr>
<td>user</td>
<td>oracle_home</td>
</tr>
<tr>
<td>password_encryption</td>
<td>on</td>
</tr>
<tr>
<td>user</td>
<td>plpgsql.variable_conflict</td>
</tr>
<tr>
<td>superuser</td>
<td>port</td>
</tr>
<tr>
<td>postmaster</td>
<td>1</td>
</tr>
<tr>
<td>65535</td>
<td></td>
</tr>
<tr>
<td>post_auth_delay</td>
<td>0</td>
</tr>
<tr>
<td>backend</td>
<td>0</td>
</tr>
<tr>
<td>2147483647</td>
<td></td>
</tr>
<tr>
<td>pre_auth_delay</td>
<td>0</td>
</tr>
<tr>
<td>sighup</td>
<td>0</td>
</tr>
<tr>
<td>60</td>
<td></td>
</tr>
<tr>
<td>qreplace_function</td>
<td>superuser</td>
</tr>
<tr>
<td>quote_all_identifiers</td>
<td>off</td>
</tr>
<tr>
<td>user</td>
<td>random_page_cost</td>
</tr>
<tr>
<td>user</td>
<td>0</td>
</tr>
<tr>
<td>replication_timeout</td>
<td>60000</td>
</tr>
<tr>
<td>sighup</td>
<td>0</td>
</tr>
<tr>
<td>restart_after_crash</td>
<td>on</td>
</tr>
<tr>
<td>sighup</td>
<td>search_path</td>
</tr>
<tr>
<td>user</td>
<td>segment_size</td>
</tr>
<tr>
<td>internal</td>
<td>131072</td>
</tr>
<tr>
<td>seq_page_cost</td>
<td>1</td>
</tr>
<tr>
<td>user</td>
<td>0</td>
</tr>
<tr>
<td>server_encoding</td>
<td>UTF8</td>
</tr>
<tr>
<td>internal</td>
<td>server_version</td>
</tr>
<tr>
<td>internal</td>
<td>server_version_num</td>
</tr>
<tr>
<td>internal</td>
<td>90200</td>
</tr>
<tr>
<td>session_replication_role</td>
<td>origin</td>
</tr>
<tr>
<td>superuser</td>
<td>shared_buffers</td>
</tr>
<tr>
<td>postmaster</td>
<td>16</td>
</tr>
<tr>
<td>Configuration</td>
<td>Value</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>------------------------</td>
</tr>
<tr>
<td><code>sql_inheritance</code></td>
<td>on</td>
</tr>
<tr>
<td><code>user ssl</code></td>
<td>off</td>
</tr>
<tr>
<td><code>postmaster ssl_ca_file</code></td>
<td></td>
</tr>
<tr>
<td><code>postmaster ssl_cert_file</code></td>
<td>server.crt</td>
</tr>
<tr>
<td><code>postmaster ssl_ciphers</code></td>
<td><code>ALL:!ADH:!LOW:!EXP:!MD5:@STRENGTH</code></td>
</tr>
<tr>
<td><code>postmaster ssl_key_file</code></td>
<td>server.key</td>
</tr>
<tr>
<td><code>postmaster ssl_renegotiation_limit</code></td>
<td>524288</td>
</tr>
<tr>
<td><code>user statement_timeout</code></td>
<td>0</td>
</tr>
<tr>
<td><code>user stats_temp_directory</code></td>
<td><code>pg_stat_tmp</code></td>
</tr>
<tr>
<td><code>sighup superuser_reserved_connections</code></td>
<td>3</td>
</tr>
<tr>
<td><code>postmaster synchronize_seqscans</code></td>
<td>on</td>
</tr>
<tr>
<td><code>user synchronous_commit</code></td>
<td>on</td>
</tr>
<tr>
<td><code>user synchronous_standby_names</code></td>
<td></td>
</tr>
<tr>
<td><code>sighup syslog_facility</code></td>
<td><code>local0</code></td>
</tr>
<tr>
<td><code>sighup syslog_ident</code></td>
<td><code>postgres</code></td>
</tr>
<tr>
<td><code>sighup tcp_keepalives_count</code></td>
<td>0</td>
</tr>
<tr>
<td><code>user tcp_keepalives_IDLE</code></td>
<td>0</td>
</tr>
<tr>
<td><code>user tcp_keepalives_interval</code></td>
<td>0</td>
</tr>
<tr>
<td><code>user temp_buffers</code></td>
<td>1024</td>
</tr>
<tr>
<td><code>user temp_file_limit</code></td>
<td>1073741823</td>
</tr>
<tr>
<td><code>superuser temp_tablespaces</code></td>
<td>2147483647</td>
</tr>
<tr>
<td><code>user timed_statistics</code></td>
<td>on</td>
</tr>
<tr>
<td><code>user TimeZone</code></td>
<td>US/Eastern</td>
</tr>
<tr>
<td><code>user timezone_abbreviations</code></td>
<td>Default</td>
</tr>
<tr>
<td><code>user trace_hints</code></td>
<td>off</td>
</tr>
<tr>
<td><code>user trace_notify</code></td>
<td>off</td>
</tr>
<tr>
<td><code>user trace_recovery_messages</code></td>
<td>log</td>
</tr>
<tr>
<td><code>sighup trace_sort</code></td>
<td>off</td>
</tr>
<tr>
<td><code>user trace_activities</code></td>
<td>on</td>
</tr>
<tr>
<td><code>superuser</code></td>
<td></td>
</tr>
<tr>
<td>Environment</td>
<td>Value 1</td>
</tr>
<tr>
<td>-------------</td>
<td>---------</td>
</tr>
<tr>
<td>track_activity_query_size</td>
<td>1024</td>
</tr>
<tr>
<td>postmaster</td>
<td>100</td>
</tr>
<tr>
<td>track_counts</td>
<td></td>
</tr>
<tr>
<td>superuser</td>
<td></td>
</tr>
<tr>
<td>track_functions</td>
<td></td>
</tr>
<tr>
<td>superuser</td>
<td></td>
</tr>
<tr>
<td>track_io_timing</td>
<td></td>
</tr>
<tr>
<td>superuser</td>
<td></td>
</tr>
<tr>
<td>transaction_deferrable</td>
<td></td>
</tr>
<tr>
<td>user</td>
<td></td>
</tr>
<tr>
<td>transaction_isolation</td>
<td></td>
</tr>
<tr>
<td>user</td>
<td></td>
</tr>
<tr>
<td>transaction_read_only</td>
<td></td>
</tr>
<tr>
<td>user</td>
<td></td>
</tr>
<tr>
<td>transform_null_equals</td>
<td></td>
</tr>
<tr>
<td>user</td>
<td></td>
</tr>
<tr>
<td>unix_socket_directory</td>
<td></td>
</tr>
<tr>
<td>postmaster</td>
<td></td>
</tr>
<tr>
<td>unix_socket_group</td>
<td></td>
</tr>
<tr>
<td>postmaster</td>
<td></td>
</tr>
<tr>
<td>unix_socket_permissions</td>
<td>0</td>
</tr>
<tr>
<td>postmaster</td>
<td>0</td>
</tr>
<tr>
<td>update_process_title</td>
<td></td>
</tr>
<tr>
<td>superuser</td>
<td></td>
</tr>
<tr>
<td>vacuum_cost_delay</td>
<td></td>
</tr>
<tr>
<td>user</td>
<td>0</td>
</tr>
<tr>
<td>vacuum_cost_limit</td>
<td></td>
</tr>
<tr>
<td>user</td>
<td>1</td>
</tr>
<tr>
<td>vacuum_cost_page_dirty</td>
<td></td>
</tr>
<tr>
<td>user</td>
<td>0</td>
</tr>
<tr>
<td>vacuum_cost_page_hit</td>
<td>1</td>
</tr>
<tr>
<td>user</td>
<td>0</td>
</tr>
<tr>
<td>vacuum_cost_page_miss</td>
<td></td>
</tr>
<tr>
<td>user</td>
<td>0</td>
</tr>
<tr>
<td>vacuum_defer_cleanup_age</td>
<td></td>
</tr>
<tr>
<td>sighup</td>
<td>0</td>
</tr>
<tr>
<td>vacuum_freeze_min_age</td>
<td></td>
</tr>
<tr>
<td>user</td>
<td>0</td>
</tr>
<tr>
<td>vacuum_freeze_table_age</td>
<td>1500000000</td>
</tr>
<tr>
<td>user</td>
<td>0</td>
</tr>
<tr>
<td>wal_block_size</td>
<td></td>
</tr>
<tr>
<td>internal</td>
<td>8192</td>
</tr>
<tr>
<td>wal_buffers</td>
<td></td>
</tr>
<tr>
<td>postmaster</td>
<td>-1</td>
</tr>
<tr>
<td>wal_keep_segments</td>
<td></td>
</tr>
<tr>
<td>sighup</td>
<td>0</td>
</tr>
<tr>
<td>wal_level</td>
<td></td>
</tr>
<tr>
<td>postmaster</td>
<td></td>
</tr>
<tr>
<td>wal_receiver_status_interval</td>
<td>10</td>
</tr>
<tr>
<td>sighup</td>
<td>0</td>
</tr>
<tr>
<td>wal_segment_size</td>
<td></td>
</tr>
<tr>
<td>internal</td>
<td>2048</td>
</tr>
<tr>
<td>wal_sync_method</td>
<td></td>
</tr>
<tr>
<td>sighup</td>
<td></td>
</tr>
<tr>
<td>wal_writer_delay</td>
<td></td>
</tr>
<tr>
<td>sighup</td>
<td>1</td>
</tr>
<tr>
<td>work_mem</td>
<td>64</td>
</tr>
<tr>
<td>user</td>
<td></td>
</tr>
<tr>
<td>xmlbinary</td>
<td>2097151</td>
</tr>
<tr>
<td>user</td>
<td></td>
</tr>
<tr>
<td>xmloption</td>
<td></td>
</tr>
<tr>
<td>user</td>
<td></td>
</tr>
</tbody>
</table>
11.6.2 **stat_db_rpt()**

The signature is:

\[ \text{stat_db_rpt}(\text{beginning_id}, \text{ending_id}) \]

**Parameters**

- **beginning_id**
  
  \( \text{beginning_id} \) is an integer value that represents the beginning session identifier.

- **ending_id**
  
  \( \text{ending_id} \) is an integer value that represents the ending session identifier.

The following example demonstrates the `stat_db_rpt()` function:

```sql
SELECT * FROM stat_db_rpt(9, 10);
```

### DATA from `pg_stat_database`

<table>
<thead>
<tr>
<th>DATABASE</th>
<th>NUMBACKENDS</th>
<th>XACT COMMIT</th>
<th>XACT ROLLBACK</th>
<th>BLKS READ</th>
<th>BLKS HIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>edb</td>
<td>1</td>
<td>21</td>
<td>0</td>
<td>92928</td>
<td>101217</td>
</tr>
</tbody>
</table>

11.6.3 **stat_tables_rpt()**

The signature is:

\[ \text{function_name}(\text{beginning_id}, \text{ending_id}, \text{top}_n, \text{scope}) \]

**Parameters**

- **beginning_id**
  
  \( \text{beginning_id} \) 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 following code sample demonstrates the *stat_tables_rpt()* function:

```sql
SELECT * FROM stat_tables_rpt(18, 19, 10, 'ALL');
```

<table>
<thead>
<tr>
<th>SCHEMA</th>
<th>RELATION</th>
<th>SEQ SCAN</th>
<th>REL TUP</th>
<th>READ</th>
<th>IDX SCAN</th>
<th>IDX TUP</th>
<th>READ</th>
<th>INS</th>
<th>UPD</th>
<th>DEL</th>
</tr>
</thead>
<tbody>
<tr>
<td>pg_catalog</td>
<td>pg_class</td>
<td>8</td>
<td>2952</td>
<td>78</td>
<td>65</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>pg_catalog</td>
<td>pg_index</td>
<td>4</td>
<td>448</td>
<td>23</td>
<td>28</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>pg_catalog</td>
<td>pg_namespace</td>
<td>4</td>
<td>76</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>pg_catalog</td>
<td>pg_database</td>
<td>3</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>pg_catalog</td>
<td>pg_authid</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>sys</td>
<td>edb$snap</td>
<td>1</td>
<td>15</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>public</td>
<td>accounts</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>public</td>
<td>branches</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>sys</td>
<td>edb$session_wait_history</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>25</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sys</td>
<td>edb$session_waits</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

DATA from pg_stat_all_tables ordered by rel tup read
### 11.6.4 statio_tables_rpt()

The signature is:

````sql
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 following example demonstrates the `statio_tables_rpt()` function:

```sql
edb=# SELECT * FROM statio_tables_rpt(9, 10, 10, 'SYS');
```

<table>
<thead>
<tr>
<th>SCHEMA</th>
<th>RELATION</th>
<th>HEAP</th>
<th>HEAP</th>
<th>HEAP</th>
<th>IDX</th>
<th>IDX</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>READ</td>
<td>HIT</td>
<td>ICACHE</td>
<td>READ</td>
<td>HIT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>IDX</td>
<td>TOAST</td>
<td>TOAST</td>
<td>TIDX</td>
<td>TIDX</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ICACHE</td>
<td>READ</td>
<td>ICACHE</td>
<td>READ</td>
<td>HIT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HIT</td>
<td>HIT</td>
<td>HIT</td>
<td>HIT</td>
<td></td>
</tr>
<tr>
<td>public</td>
<td>pgbench_accounts</td>
<td>92766</td>
<td>67215</td>
<td>288</td>
<td>59</td>
<td>32126</td>
</tr>
<tr>
<td></td>
<td></td>
<td>9</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>pg_class</td>
<td>0</td>
<td>296</td>
<td>0</td>
<td>3</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sys</td>
<td>edb$stat_all_indexes</td>
<td>8</td>
<td>125</td>
<td>0</td>
<td>4</td>
<td>233</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sys</td>
<td>edb$statio_all_index</td>
<td>8</td>
<td>125</td>
<td>0</td>
<td>4</td>
<td>233</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sys</td>
<td>edb$stat_all_tables</td>
<td>6</td>
<td>91</td>
<td>0</td>
<td>2</td>
<td>174</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sys</td>
<td>edb$statio_all_table</td>
<td>6</td>
<td>91</td>
<td>0</td>
<td>2</td>
<td>174</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>pg_namespace</td>
<td>3</td>
<td>72</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sys</td>
<td>edb$session_wait_his</td>
<td>1</td>
<td>24</td>
<td>0</td>
<td>4</td>
<td>47</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>pg_opclass</td>
<td>3</td>
<td>13</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>pg_trigger</td>
<td>0</td>
<td>12</td>
<td>0</td>
<td>1</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

(16 rows)

### 11.6.5 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 following code sample demonstrates the stat_indexes_rpt() function:

```
edb=# SELECT * FROM stat_indexes_rpt(9, 10, 10, 'ALL');

<table>
<thead>
<tr>
<th>SCHEMA</th>
<th>RELATION</th>
<th>INDEX</th>
</tr>
</thead>
<tbody>
<tr>
<td>pg_catalog</td>
<td>pg_cast</td>
<td>pg_cast_source_target_index</td>
</tr>
<tr>
<td></td>
<td></td>
<td>30</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>pg_class</td>
<td>pg_class_oid_index</td>
</tr>
<tr>
<td></td>
<td></td>
<td>15</td>
</tr>
<tr>
<td></td>
<td></td>
<td>15</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>pg_trigger</td>
<td>pg_trigger_tgrelid_tgname_index</td>
</tr>
<tr>
<td></td>
<td></td>
<td>12</td>
</tr>
<tr>
<td></td>
<td></td>
<td>12</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>pg_attribute</td>
<td>pg_attribute_relid_attnum_index</td>
</tr>
<tr>
<td></td>
<td></td>
<td>31</td>
</tr>
<tr>
<td></td>
<td></td>
<td>31</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>pg_statistic</td>
<td>pg_statistic_relid_att_index</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>
```
11.6.6  

**statio_indexes_rpt()**

The signature is:

\[
\text{statio_indexes_rpt}(\text{beginning_id}, \text{ending_id}, \text{top_n}, \text{scope})
\]

**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 following example demonstrates the `statio_indexes_rpt()` function:
```
edb=# SELECT * FROM statio_indexes_rpt(9, 10, 10, 'SYS');

DATA from pg_statio_all_indexes

<table>
<thead>
<tr>
<th>SCHEMA</th>
<th>RELATION</th>
<th>INDEX</th>
<th>IDX BLKS READ</th>
<th>IDX BLKS HIT</th>
<th>IDX BLKS ICACHE HIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>pgbench_accounts</td>
<td>pgbench_accounts_pkey</td>
<td>59</td>
<td>32126</td>
<td>9</td>
</tr>
<tr>
<td>sys</td>
<td>edb$stat_all_indexes</td>
<td>edb$stat_idx_pk</td>
<td>4</td>
<td>233</td>
<td>0</td>
</tr>
<tr>
<td>sys</td>
<td>edb$statio_all_indexes</td>
<td>edb$statio_idx_pk</td>
<td>4</td>
<td>233</td>
<td>0</td>
</tr>
<tr>
<td>sys</td>
<td>edb$stat_all_tables</td>
<td>edb$stat_tab_pk</td>
<td>2</td>
<td>174</td>
<td>0</td>
</tr>
<tr>
<td>sys</td>
<td>edb$statio_all_tables</td>
<td>edb$statio_tab_pk</td>
<td>2</td>
<td>174</td>
<td>0</td>
</tr>
<tr>
<td>sys</td>
<td>edb$session_wait_history</td>
<td>session_waits_hist_pk</td>
<td>4</td>
<td>47</td>
<td>0</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>pg_cast</td>
<td>pg_cast_source_target_index</td>
<td>1</td>
<td>29</td>
<td>0</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>pg_trigger</td>
<td>pg_trig_tgrelid_tgname_index</td>
<td>1</td>
<td>15</td>
<td>0</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>pg_class</td>
<td>pg_class_oid_index</td>
<td>1</td>
<td>14</td>
<td>0</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>pg_statistic</td>
<td>pg_statistic_relid_att_index</td>
<td>2</td>
<td>12</td>
<td>0</td>
</tr>
</tbody>
</table>
(14 rows)
```
11.7 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:

<table>
<thead>
<tr>
<th>Event type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Checkpoint waits</td>
<td>Checkpoint waits may indicate that checkpoint parameters need to be adjusted, (checkpoint_segments and checkpoint_timeout).</td>
</tr>
<tr>
<td>WAL-related waits</td>
<td>WAL-related waits may indicate wal_buffers are under-sized.</td>
</tr>
<tr>
<td>SQL Parse waits</td>
<td>If the number of waits is high, try to use prepared statements.</td>
</tr>
<tr>
<td>db file random reads</td>
<td>If high, check that appropriate indexes and statistics exist.</td>
</tr>
<tr>
<td>db file random writes</td>
<td>If high, may need to decrease bgwriter_delay.</td>
</tr>
<tr>
<td>btree random lock acquires</td>
<td>May indicate indexes are being rebuilt. Schedule index builds during less active time.</td>
</tr>
</tbody>
</table>

Performance reviews should also include careful scrutiny of the hardware, the operating system, the network and the application SQL statements.
### Event Descriptions

<table>
<thead>
<tr>
<th>Event Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>add in shmem lock acquire</td>
<td>Obsolete/unused</td>
</tr>
<tr>
<td>bgwriter communication lock acquire</td>
<td>The bgwriter (background writer) process has waited for the short-term lock that synchronizes messages between the bgwriter and a backend process.</td>
</tr>
<tr>
<td>btree vacuum lock acquire</td>
<td>The server has waited for the short-term lock that synchronizes access to the next available vacuum cycle ID.</td>
</tr>
<tr>
<td>buffer free list lock acquire</td>
<td>The server has waited for the short-term lock that synchronizes access to the list of free buffers (in shared memory).</td>
</tr>
<tr>
<td>checkpoint lock acquire</td>
<td>A server process has waited for the short-term lock that prevents simultaneous checkpoints.</td>
</tr>
<tr>
<td>checkpoint start lock acquire</td>
<td>The server has waited for the short-term lock that synchronizes access to the bgwriter checkpoint schedule.</td>
</tr>
<tr>
<td>clog control lock acquire</td>
<td>The server has waited for the short-term lock that synchronizes access to the commit log.</td>
</tr>
<tr>
<td>control file lock acquire</td>
<td>The server has waited for the short-term lock that synchronizes write access to the control file (this should usually be a low number).</td>
</tr>
<tr>
<td>db file extend</td>
<td>A server process has waited for the operating system while adding a new page to the end of a file.</td>
</tr>
<tr>
<td>db file read</td>
<td>A server process has waited for the completion of a read (from disk).</td>
</tr>
<tr>
<td>db file write</td>
<td>A server process has waited for the completion of a write (to disk).</td>
</tr>
<tr>
<td>db file sync</td>
<td>A server process has waited for the operating system to flush all changes to disk.</td>
</tr>
<tr>
<td>first buf mapping lock acquire</td>
<td>The server has waited for a short-term lock that synchronizes access to the shared-buffer mapping table.</td>
</tr>
<tr>
<td>freespace lock acquire</td>
<td>The server has waited for the short-term lock that synchronizes access to the freespace map.</td>
</tr>
<tr>
<td>Infinite Cache read</td>
<td>The server has waited for an Infinite Cache read request.</td>
</tr>
<tr>
<td>Infinite Cache write</td>
<td>The server has waited for an Infinite Cache write request.</td>
</tr>
<tr>
<td>lwlock acquire</td>
<td>The server has waited for a short-term lock that has not been described elsewhere in this section.</td>
</tr>
<tr>
<td>multi xact gen lock acquire</td>
<td>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).</td>
</tr>
<tr>
<td>multi xact member lock acquire</td>
<td>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).</td>
</tr>
<tr>
<td>multi xact offset lock acquire</td>
<td>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).</td>
</tr>
<tr>
<td>oid gen lock acquire</td>
<td>The server has waited for the short-term lock that synchronizes access to the next available OID (object ID).</td>
</tr>
<tr>
<td>query plan</td>
<td>The server has computed the execution plan for a SQL statement.</td>
</tr>
<tr>
<td>rel cache init lock acquire</td>
<td>The server has waited for the short-term lock that prevents simultaneous relation-cache loads/unloads.</td>
</tr>
<tr>
<td>shmem index lock acquire</td>
<td>The server has waited for the short-term lock that synchronizes access to the shared-memory map.</td>
</tr>
<tr>
<td>sINVAL lock acquire</td>
<td>The server has waited for the short-term lock that synchronizes access to the cache invalidation state.</td>
</tr>
<tr>
<td>sql parse</td>
<td>The server has parsed a SQL statement.</td>
</tr>
<tr>
<td>Lock Type</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>subtrans control lock</td>
<td>The server has waited for the short-term lock that synchronizes access to</td>
</tr>
<tr>
<td>acquire</td>
<td>the subtransaction log.</td>
</tr>
<tr>
<td>tablespace create lock</td>
<td>The server has waited for the short-term lock that prevents simultaneous</td>
</tr>
<tr>
<td>acquire</td>
<td>CREATE TABLESPACE or DROP TABLESPACE commands.</td>
</tr>
<tr>
<td>two phase state lock</td>
<td>The server has waited for the short-term lock that synchronizes access to</td>
</tr>
<tr>
<td>acquire</td>
<td>the list of prepared transactions.</td>
</tr>
<tr>
<td>wal insert lock acquire</td>
<td>The server has waited for the short-term lock that synchronizes write access</td>
</tr>
<tr>
<td></td>
<td>to the write-ahead log. A high number may indicate that WAL buffers are sized</td>
</tr>
<tr>
<td></td>
<td>too small.</td>
</tr>
<tr>
<td>wal write lock acquire</td>
<td>The server has waited for the short-term lock that synchronizes write-</td>
</tr>
<tr>
<td></td>
<td>ahead log flushes.</td>
</tr>
<tr>
<td>wal file sync</td>
<td>The server has waited for the write-ahead log to sync to disk (related to</td>
</tr>
<tr>
<td></td>
<td>the wal_sync_method parameter which, by default, is 'fsync' - better</td>
</tr>
<tr>
<td></td>
<td>performance can be gained by changing this parameter to open_sync).</td>
</tr>
<tr>
<td>wal flush</td>
<td>The server has waited for the write-ahead log to flush to disk.</td>
</tr>
<tr>
<td>wal write</td>
<td>The server has waited for a write to the write-ahead log buffer (expect</td>
</tr>
<tr>
<td></td>
<td>this value to be high).</td>
</tr>
<tr>
<td>xid gen lock acquire</td>
<td>The server has waited for the short-term lock that synchronizes access to</td>
</tr>
<tr>
<td></td>
<td>the next available transaction ID.</td>
</tr>
</tbody>
</table>
11.9 Catalog Views

The following DRITA catalog views provide access to performance information relating to system waits.

11.9.1  edb$system_waits

The edb$system_waits view summarizes the number of waits and the total wait time per session for each wait named. It also displays the average and max wait times. edb$system_waits summarizes the following information:

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Modifiers</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>edb_id</td>
<td>numeric</td>
<td></td>
<td>identifier</td>
</tr>
<tr>
<td>dbname</td>
<td>text</td>
<td></td>
<td>database name</td>
</tr>
<tr>
<td>wait_name</td>
<td>text</td>
<td></td>
<td>name of the event</td>
</tr>
<tr>
<td>wait_count</td>
<td>numeric</td>
<td></td>
<td>number of times the event occurs</td>
</tr>
<tr>
<td>avg_wait</td>
<td>numeric(50,6)</td>
<td></td>
<td>average wait time in microseconds</td>
</tr>
<tr>
<td>max_wait</td>
<td>numeric</td>
<td></td>
<td>maximum wait time in microseconds</td>
</tr>
<tr>
<td>total_time</td>
<td>numeric</td>
<td></td>
<td>total wait time in microseconds</td>
</tr>
<tr>
<td>wait_name</td>
<td>text</td>
<td></td>
<td>name of the event</td>
</tr>
</tbody>
</table>

The following example shows the result of a SELECT statement on the edb$system_waits view:

```
select * from sys.edb$system_waits;
```

<table>
<thead>
<tr>
<th>edb_id</th>
<th>dbname</th>
<th>wait_name</th>
<th>wait_count</th>
<th>avg_wait</th>
<th>max_wait</th>
<th>total_wait</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>edb</td>
<td>db fileread</td>
<td>301</td>
<td>0.011516</td>
<td>0.629986</td>
<td>2.742500</td>
</tr>
<tr>
<td>1</td>
<td>edb</td>
<td>wal flush</td>
<td>26</td>
<td>0.010364</td>
<td>0.085380</td>
<td>0.269452</td>
</tr>
<tr>
<td>1</td>
<td>edb</td>
<td>wal write</td>
<td>26</td>
<td>0.010355</td>
<td>0.085371</td>
<td>0.269232</td>
</tr>
<tr>
<td>1</td>
<td>edb</td>
<td>query plan</td>
<td>277</td>
<td>0.001367</td>
<td>0.049425</td>
<td>0.262596</td>
</tr>
<tr>
<td>2</td>
<td>edb</td>
<td>wal flush</td>
<td>28</td>
<td>0.040443</td>
<td>0.095150</td>
<td>0.431984</td>
</tr>
<tr>
<td>2</td>
<td>edb</td>
<td>wal write</td>
<td>28</td>
<td>0.040434</td>
<td>0.095093</td>
<td>0.431698</td>
</tr>
<tr>
<td>2</td>
<td>edb</td>
<td>query plan</td>
<td>299</td>
<td>0.001479</td>
<td>0.049425</td>
<td>0.262596</td>
</tr>
</tbody>
</table>

11.9.2  edb$session_waits

The edb$session_waits view summarizes the number of waits and the total wait time per session for each wait named and identified by backend ID. It also displays the average and max wait times. edb$session_waits summarizes the following information:

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Modifiers</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>backend_id</td>
<td>bigint</td>
<td></td>
<td>session identifier</td>
</tr>
<tr>
<td>wait_count</td>
<td>bigint</td>
<td></td>
<td>number of times the event occurs</td>
</tr>
<tr>
<td>avg_wait_time</td>
<td>numeric</td>
<td></td>
<td>average wait time in microseconds</td>
</tr>
<tr>
<td>max_wait_time</td>
<td>numeric(50,6)</td>
<td></td>
<td>maximum wait time in microseconds</td>
</tr>
</tbody>
</table>
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        | enterpriseb | <IDLE>
 1 | edb   | 22988 | db file read   | 116        | 0.009556     | 0.040627      | 1.108438        | enterpriseb | select * from edbsnap();
 1 | edb   | 22988 | wal flush      | 26         | 0.010364     | 0.085380      | 0.269452        | enterpriseb | select * from edbsnap();
```

(3 rows)

### 11.9.3  `edb$session_wait_history`

The `edb$session_wait_history` view contains the last 25 wait events for each backend ID active during the session. The `edb$session_wait_history` view includes the following information:

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Modifiers</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>edb_id</td>
<td>numeric</td>
<td></td>
<td>identifier</td>
</tr>
<tr>
<td>dbname</td>
<td>text</td>
<td></td>
<td>database name</td>
</tr>
<tr>
<td>backend_id</td>
<td>bigint</td>
<td></td>
<td>session identifier</td>
</tr>
<tr>
<td>seq</td>
<td>bigint</td>
<td></td>
<td>number between 1 and 25</td>
</tr>
<tr>
<td>wait_name</td>
<td>text</td>
<td></td>
<td>name of the event</td>
</tr>
<tr>
<td>elapsed</td>
<td>bigint</td>
<td></td>
<td>elapsed time in microseconds</td>
</tr>
<tr>
<td>p1</td>
<td>bigint</td>
<td></td>
<td>variable #1- meaning dependent on event</td>
</tr>
<tr>
<td>p2</td>
<td>bigint</td>
<td></td>
<td>variable #2- meaning dependent on event</td>
</tr>
<tr>
<td>p3</td>
<td>bigint</td>
<td></td>
<td>variable #3- meaning dependent on event</td>
</tr>
</tbody>
</table>

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    | 8  | 1  | 
 1 | edb   | 22935 | 3   | db file read    | 983     | 32 | 1  | 
 1 | edb   | 22935 | 4   | db file read    | 13717   | 19 | 1  | 
 1 | edb   | 22935 | 5   | query plan      | 75      | 0  | 0  | 0
 1 | edb   | 22935 | 6   | db file read    | 11053   | 7  | 1  | 
 1 | edb   | 22935 | 7   | db file read    | 404     | 4  | 1  |
```

(7 rows)
12 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-expensive (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.

LIST partitioning

In a LIST partitioned table, each partition stores data that is described by a partitioning rule. For example:

```sql
PARTITION BY LIST(country)
(
    PARTITION europe VALUES('ITALY', 'FRANCE'),
    PARTITION asia VALUES('INDIA', 'PAKISTAN')
...)
```

When you INSERT a new row into the table, Advanced Server decides which partition will hold the row based on the value of the country column. If country is ITALY or FRANCE, Advanced Server stores the row in the partition named europe. If country is INDIA or PAKISTAN, Advanced Server stores the row in the partition named asia. If
country contains any other value, Advanced Server reports an error (you can specify a *DEFAULT* partition to catch other values).

**RANGE partitioning**

In a *RANGE* partitioned table, each partition specifies a range of values. For example:

```sql
PARTITION BY RANGE(order_date)
(
  PARTITION q1_2011 VALUES LESS THAN('2011-Apr-01'),
  PARTITION q2_2011 VALUES LESS THAN('2011-Jul-01'),
  PARTITION q3_2011 VALUES LESS THAN('2011-Oct-01'),
  PARTITION q4_2011 VALUES LESS THAN('2012-Jan-01'),
  PARTITION others VALUES LESS THAN(MAXVALUE)
);
```

When you *INSERT* a row into this table, Advanced Server chooses which partition will hold the row based on the value of the *order_date* column. Advanced Server compares the value of *order_date* to 2011-Apr-01; if it is less than 2011-Apr-01, it stores the row in the q1_2011 partition. If not, it continues to the next rule, and so on until it finds a partition that describes the value of *order_date*.

The *others* partition catches any orders with an *order_date* that does not fit the rules that define the other partition. The *MAXVALUE* keyword matches any value that does not fit into one of the specified ranges.

Please note: range values are non-inclusive (only those values *less than* the value specified will satisfy the rule), and must be specified in ascending order.

**Subpartitioning**

You can define a *subpartition* to further divide the data stored in a table partition into smaller (more manageable) subsets. For example:

```sql
PARTITION BY RANGE(order_date) SUBPARTITION BY LIST(country)
(
  PARTITION q1_2011 VALUES LESS THAN('2011-Apr-01')
  (  
    SUBPARTITION q1_2011_europe VALUES('ITALY', 'GERMANY'),
    SUBPARTITION q1_2011_asia VALUES('INDIA', 'PAKISTAN')
  ),
  PARTITION q2_2011 VALUE LESS THAN '2011-Jul-01')
  (  
    SUBPARTITION q2_2011_europe VALUES('ITALY', 'GERMANY'),
    SUBPARTITION q2_2011_asia VALUES('INDIA', 'PAKISTAN')
  ),
  ...)
```
When you INSERT a row into this table, Advanced Server chooses which subpartition will hold the row first based on the value of the order_date column, and then based on the location specified in the country column. An entry with an order_date of 2011-May-01, with a country value of INDIA, will be stored in the q2_2011_asia subpartition.

You can divide a RANGE partitioned table by LIST subpartitions, or a LIST partitioned table by RANGE subpartitions.

You cannot add partitions or subpartitions to an existing table that does not have a partition (or subpartitions); to create a partitioned table, you must use the CREATE TABLE...PARTITION BY command.
12.1 CREATE TABLE...PARTITION BY

Use the CREATE TABLE...PARTITION BY command to create a table with data distributed amongst one or more partitions. The syntax is:

```
CREATE TABLE [ schema. ]table_name
    PARTITION BY {RANGE|LIST}(column[, column ]...)
    [SUBPARTITION BY {RANGE | LIST} (column[, column ]...)]
    (partition_definition[, 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}
```

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 `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 records to the given partition.

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, 2010 will include only those date values that fall on or before December 31, 2009.

Range partition rules must be listed in ascending order to ensure that a given row ends up in the correct partition. `INSERT` commands that store records with values that exceed the top boundary of a `RANGE` partitioned table will fail. Include a boundary rule that specifies a value of `MAXVALUE` to direct any record that exceeds the values specified in the partitioning rules to a given partition; if you do not include a `MAXVALUE` rule, any record that exceeds the maximum limit specified by the boundary rules will result in an error.

If you use `CREATE TABLE` syntax to create an index on a partitioned table, the index will also be created in each partition or subpartition. If you add an index to a partitioned table using the `CREATE INDEX` command, indexes are not automatically created on partitions or subpartitions.

Parameters

`table_name`

The name (optionally schema-qualified) of the table to be created.

`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 records 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 record that does not match the specified rules of at least one partition will fail, and return an error.

When specifying rules for a RANGE partitioned table, include the MAXVALUE keyword in the last partition rule to direct any un-categorized records to the given partition. If you do not include a rule that includes a value of MAXVALUE, any INSERT statement that attempts to add a record with a value greater than the highest value specified will fail, and return an error.

**Example - PARTITION BY LIST**

The following example creates a partitioned table (employees) using the PARTITION BY LIST clause. The *employees* table stores information in three partitions (mgmt, sales, and support):

```sql
CREATE TABLE employees
  (empno numeric(4,0),
   ename varchar(10),
   job varchar(9),
   hiredate timestamp
  )
PARTITION BY LIST (job)
  (PARTITION mgmt VALUES ('MANAGER', 'PRESIDENT'),
   PARTITION sales VALUES ('SALESMAN', 'CLERK'),
   PARTITION support VALUES ('ANALYST', 'DEFAULT'));
```

The partition that stores a specific entry is determined by the value entered in the job field. Administrative staff (with a job description of MANAGER or PRESIDENT) are stored in a partition named mgmt; employees with a job description of SALESMAN or CLERK are stored in a partition named sales.

**Example - PARTITION BY RANGE**

The following example creates a partitioned table named employees; the table stores employee information in three partitions (one_year_staff, five_year_staff and ten_year_staff) created using the PARTITION BY RANGE clause:

```
CREATE TABLE employees
(
    empno     numeric(4,0),
    ename     varchar(10),
    job       varchar(9),
    hiredate  timestamp
)
PARTITION BY RANGE (hiredate)
(
    PARTITION ten_year_staff VALUES LESS THAN('01-JAN-2001'),
    PARTITION five_year_staff VALUES LESS THAN('01-JAN-2006'),
    PARTITION one_year_staff VALUES LESS THAN('01-JAN-2011')
);
```

- The first partition, named ten_year_staff contains employee data for employees that have a hiredate before January 1, 2001.
- The second partition, named five_year_staff, contains employee data for those employees that have a hiredate before January 1, 2006.
- The third partition, named one_year_staff, contains employee data for those employees that have a hiredate before January 1, 2011.

**Example - PARTITION BY RANGE, SUBPARTITION BY LIST**

The following example combines the syntax of the previous examples to create a partitioned table that is first partitioned by RANGE; the RANGE partitions (one_year_staff, five_year_staff and ten_year_staff) are then subpartitioned by LIST subpartitioning rules. The LIST subpartition rules divide each RANGE partition into three partitions (mgmt, sales and support):

```
CREATE TABLE employees
(
    empno     numeric(4,0),
    ename     varchar(10),
    job       varchar(9),
```
hiredate  timestamp
)
PARTITION BY RANGE (hiredate)
    SUBPARTITION BY LIST (job)
    {
        PARTITION ten_year_staff VALUES LESS THAN ('01-JAN-2001')
            {
                SUBPARTITION sr_mgm VALUES ('MANAGER', 'PRESIDENT'),
                SUBPARTITION sr_sales VALUES ('SALESMAN', 'CLERK'),
                SUBPARTITION sr_support VALUES ('ANALYST')
            },
        PARTITION five_year_staff VALUES LESS THAN ('01-JAN-2006')
            {
                SUBPARTITION mgmt VALUES ('MANAGER', 'PRESIDENT'),
                SUBPARTITION sales VALUES ('SALESMAN', 'CLERK'),
                SUBPARTITION support VALUES ('ANALYST')
            },
        PARTITION one_year_staff VALUES LESS THAN ('01-JAN-2011')
            {
                SUBPARTITION new_mgm VALUES ('MANAGER', 'PRESIDENT'),
                SUBPARTITION new_sales VALUES ('SALESMAN', 'CLERK'),
                SUBPARTITION new_support VALUES ('ANALYST')
            }
    }
12.2 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 }
```

and `list_subpartition` is:

```
SUBPARTITION partition_name
VALUES (value[, value]...)
[TABLESPACE tablespace_name]
```

and `range_subpartition` is:

```
SUBPARTITION partition_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.
The new partition rules will reference the column specified in the rules that define the existing partition(s). If the existing partitions are defined using LIST rules, any new partitions added with the ADD PARTITION clause must also be defined using LIST rules. If the existing partitions use RANGE boundaries, any new partitions must also specify RANGE boundaries.

If the partitioned table is indexed, the index will be replicated 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 modified 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.

`column`

The name of a column on which the partitioning rules are based. The `value` stored in the specified column will determine which partition that the record will be stored in.

`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 records 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 record that does not match the specified rules of at least one partition will fail, and return an error.

When specifying rules for a RANGE partitioned table, include the MAXVALUE keyword in the last partition rule to direct any un-categorized records to the given partition. If you do not include a rule that includes a value of MAXVALUE, any INSERT statement that attempts to add a record with a value 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 - Adding a partition to a LIST partitioned table.

The example that follows adds a partition to the LIST partitioned version of the employees table. The table was created using the command:

```sql
CREATE TABLE employees
(
    empno     numeric(4,0),
    ename     varchar(10),
    job       varchar(9),
    mgr       numeric(4,0),
    hiredate  timestamp,
    sal       numeric(7,2),
    comm      numeric(7,2),
    deptno    numeric(7,2)
)
PARTITION BY LIST (job)
(
    PARTITION sales VALUES ('SALESMAN', 'CLERK'),
    PARTITION support VALUES ('ANALYST', 'DEFAULT')
);
```

The following command demonstrates adding a partition named mgmt to the employees table:

```sql
ALTER TABLE employees ADD PARTITION mgmt VALUES ('MANAGER', 'PRESIDENT');
```

After executing the command, the table will have three partitions - sales, support and mgmt.

Example - Adding a partition to a RANGE partitioned table.

The example adds a partition to the RANGE partitioned version of the employees table. The table was created using the command:
CREATE TABLE employees
(
    empno numeric(4,0),
    ename varchar(10),
    job varchar(9),
    mgr numeric(4,0),
    hiredate timestamp,
    sal numeric(7,2),
    comm numeric(7,2),
    deptno numeric(7,2)
)
PARTITION BY RANGE (hiredate)
(
    PARTITION ten_year_staff VALUES LESS THAN('01-JAN-2001'),
    PARTITION five_year_staff VALUES LESS THAN('01-JAN-2006')
);

Use the following command to add a third partition (one_year_staff) that stores employee records for those employees with a hiredate that falls between January 1, 2006 and December 31, 2010:

    ALTER TABLE employees ADD PARTITION one_year_staff
    VALUES LESS THAN('01-JAN-2011');

After executing the command, the employees table will have three partitions - one_year_staff, five_year_staff and ten_year_staff.
12.3 ALTER TABLE... ADD SUBPARTITION

The ALTER TABLE... ADD SUBPARTITION command adds a subpartition to an existing subpartitioned partition. The syntax is:

```sql
ALTER TABLE table_name MODIFY PARTITION partition_name
    ADD SUBPARTITION subpartition_definition;
```

Where `subpartition_definition` is:

```sql
{list_subpartition | range_subpartition }

and list_subpartition is:

```
SUBPARTITION subpartition_name
VALUES (value[, value]...)
[TABLESPACE tablespace_name]
```

and range_subpartition is:

```sql
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.

The new subpartition rules must reference the column specified in the rules that define the existing subpartition(s). If the existing subpartitions are defined using LIST rules, any new subpartitions added with the ADD SUBPARTITION clause will also be defined using LIST rules. If the existing subpartitions use RANGE boundaries, any new subpartitions will also specify RANGE boundaries.

If the subpartitioned is indexed, the index will be replicated 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 modified 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.

column

The name of a column on which the partitioning rules are based. The value stored in the specified column will determine which partition that the record will be stored in.

(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 records 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 record that does not match the specified rules of at least one partition will fail, and return an error.

When specifying rules for a RANGE partitioned table, include the MAXVALUE keyword in the last partition rule to direct any un-categorized records to the given partition. If you do not include a rule that includes a value of MAXVALUE, any INSERT statement that attempts to add a record with a value 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 - Adding a LIST subpartition to a RANGE partitioned table.

This example adds a LIST partition to the RANGE partitioned employees table. The employees table was created using the command:
CREATE TABLE employees
(
    empno numeric(4,0),
    ename varchar(10),
    job varchar(9),
    hiredate timestamp
)
PARTITION BY RANGE (hiredate)
SUBPARTITION BY LIST (job)
(
    PARTITION ten_year_staff VALUES LESS THAN ('01-JAN-2001')
    (
        SUBPARTITION mgmt VALUES ('MANAGER', 'PRESIDENT'),
        SUBPARTITION sales VALUES ('SALESMAN', 'CLERK')
    ),
    PARTITION five_year_staff VALUES LESS THAN ('01-JAN-2006'),
    PARTITION one_year_staff VALUES LESS THAN ('01-JAN-2011')
);

After executing the above command, the employees table will have three partitions - ten_year_staff, five_year_staff and one_year_staff. The ten_year_staff partition has two subpartitions; mgmt and sales.

The following command adds a third subpartition to the ten_year_staff partition, named support:

    ALTER TABLE employees MODIFY PARTITION ten_year_staff
    ADD SUBPARTITION support VALUES ('ANALYST');
12.4 ALTER TABLE...DROP PARTITION

Use the ALTER TABLE... DROP PARTITION command to delete a partition and the data stored in that partition. The syntax is:

```
ALTER TABLE table_name DROP PARTITION partition_name;
```

Description

The ALTER TABLE... DROP PARTITION deletes a partition and the data contained within that partition.

To use the DROP PARTITION clause, you must be the table owner, a member of a group that owns the table with DROP TABLE privileges, or have superuser or administrative privileges.

Parameters

- **table_name**
  
  The name (optionally schema-qualified) of the modified table.

- **partition_name**
  
  The name of the partition to be deleted.

Example

The following example deletes the partition, sales from the employees table:

```
ALTER TABLE employees DROP PARTITION sales;
```
12.5 ALTER TABLE... DROP SUBPARTITION

Use the ALTER TABLE... DROP SUBPARTITION command to delete a subpartition and the data stored in that subpartition. The syntax is:

    ALTER TABLE table_name DROP SUBPARTITION subpartition_name;

Description

The ALTER TABLE... DROP SUBPARTITION deletes a subpartition and the data contained within that partition. To use the DROP SUBPARTITION clause, you must be the table owner, or have superuser or administrative privileges.

Parameters

*table_name*

The name (optionally schema-qualified) of the modified table.

*subpartition_name*

The name of the subpartition to be deleted.

Example

The following example deletes the subpartition, sales from the employees table:

    ALTER TABLE employees DROP SUBPARTITION salesman;
12.6 ALTER TABLE...SPLIT PARTITION

Use the ALTER TABLE... SPLIT PARTITION command to divide a single partition into two partitions. The command comes in two variations; the first variation 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 variation 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 partitioned table. There is no upper limit to the number of defined partitions. When you execute an ALTER TABLE... SPLIT PARTITION command, Advanced Server creates two new partitions, moving any rows that contain the specified values (in the case of a LIST partition) or which are less than the specified values (in the case of a RANGE partition) into new_part1, and any remaining rows into new_part2.

If the existing partitions are defined using LIST rules, any new partitions added with the SPLIT PARTITION clause must also be defined using LIST rules. If the existing partitions use RANGE boundaries, any new partitions must also specify RANGE boundaries.

If the partitioned table is indexed, the index will be replicated 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 modified 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.

**column**

The name of a column on which the partitioning rules are based. The value stored in the specified column will determine which partition that the record will be stored in.

**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.

When specifying rules for a RANGE partitioned table, include the MAXVALUE keyword in the last partition rule to direct any un-categorized records to the given partition. If you do not include a rule that includes a value of MAXVALUE, any INSERT statement that attempts to add a record with a value greater than the highest value specified will fail, and return an error.

(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 records 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 record that does not match the specified rules of at least one partition will fail, and return an error.

`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 `employees` table (created with the following statement) into two partitions:

```sql
CREATE TABLE employees
(
  empno numeric(4,0),
  ename varchar(10),
  job varchar(9),
  hiredate timestamp
)
PARTITION BY LIST (job)
(
  PARTITION mgmt VALUES ('MANAGER', 'PRESIDENT'),
  PARTITION staff VALUES ('SALESMAN', 'CLERK', 'ANALYST')
);
```

The table definition creates two partitions; one partition contains management employees, and the other, non-management employees. The following command uses the `VALUES` keyword to divide the `staff` partition into two LIST partitions, `sales` and `support`:

```sql
ALTER TABLE employees SPLIT PARTITION staff
  VALUES ('SALESMAN', 'CLERK')
  INTO (PARTITION sales, PARTITION support);
```

**Example - Splitting a RANGE partition.**

Our example will divide one of the partitions in the `employees` table (created with the following statement) into two partitions:

```sql
CREATE TABLE employees
(
  empno numeric(4,0),
  ename varchar(10),
  job varchar(9),
  hiredate timestamp
)
PARTITION BY RANGE (hiredate)
(
    PARTITION ten_year_staff VALUES LESS THAN('01-JAN-2001'),
    PARTITION five_year_staff VALUES LESS THAN('01-JAN-2006')
);

ALTER TABLE employees SPLIT PARTITION five_year_staff
AT ('01-JAN-2010')
INTO (PARTITION one_year_staff, PARTITION five_year_staff);
12.7 ALTER TABLE...SPLIT SUBPARTITION

Use the ALTER TABLE... SPLIT SUBPARTITION command to divide a single partition into two partitions. The command comes in two variations; the first variation splits a RANGE subpartition into two subpartitions:

```sql
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 partition into two subpartitions:

```sql
ALTER TABLE table_name SPLIT SUBPARTITION subpartition_name
   VALUES (value[, value]...)
   INTO
   { PARTITION new_subpart1 [TABLESPACE tablespace_name],
     PARTITION 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 the specified values (in the case of a LIST subpartition) or which are less than the specified values (in the case of a RANGE subpartition) into `new_subpart1`, and any remaining rows into `new_subpart2`.

The new subpartition rules will reference the column specified in the rules that define the existing subpartition(s). If the existing subpartitions are defined using LIST rules, any new subpartitions added with the SPLIT SUBPARTITION clause must also be defined using LIST rules. If the existing subpartitions use RANGE boundaries, any new subpartitions must also specify RANGE boundaries.

If the partitioned table is indexed, the index will be replicated 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

\textit{table\_name}

The name (optionally schema-qualified) of the modified table.

\textit{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.

\textit{column}

The name of a column on which the partitioning rules are based. The value stored in the specified column will determine which partition that the record will be stored in.

\textit{(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 \texttt{LIST} partition), or MAXVALUE (if specifying a \texttt{RANGE} partition).

When specifying rules for a \texttt{LIST} partitioned table, include the \texttt{DEFAULT} keyword in the last partition rule to direct any un-matched records to the given partition. If you do not include a rule that includes a value of \texttt{DEFAULT}, any INSERT statement that attempts to add a record that does not match the specified rules of at least one partition will fail, and return an error.

When specifying rules for a \texttt{RANGE} partitioned table, include the \texttt{MAXVALUE} keyword in the last partition rule to direct any un-categorized records to the given partition. If you do not include a rule that includes a value of \texttt{MAXVALUE}, any INSERT statement that attempts to add a record with a value greater than the highest value specified will fail, and return an error.

\textit{tablespace\_name}

The name of the tablespace in which the partition or subpartition resides.

Example

This example will split the \texttt{europe\_2000} subpartition in the \texttt{sales\_hist} table (created with the following statement) into two subpartitions:
CREATE TABLE salesHist
(sales_year int, country varchar2(30), amount numeric)
PARTITION BY RANGE(sales_year) SUBPARTITION BY LIST(country)
(
    PARTITION sales_2000 VALUES LESS THAN(2001)
    (
        SUBPARTITION fareast_2000 VALUES ('JAPAN', 'VIETNAM'),
        SUBPARTITION europe_2000 VALUES ('GERMANY', 'ITALY'),
        SUBPARTITION west_2000 VALUES ('AMERICA', 'CANADA'),
        SUBPARTITION east_2000 VALUES ('INDIA', 'PAKISTAN'),
        SUBPARTITION rest_2000 VALUES (DEFAULT)
    ),
    PARTITION sales_2001 VALUES LESS THAN(2002)
    (
        SUBPARTITION fareast_2001 VALUES ('JAPAN', 'VIETNAM'),
        SUBPARTITION europe_2001 VALUES ('GERMANY', 'ITALY'),
        SUBPARTITION west_2001 VALUES ('AMERICA', 'CANADA'),
        SUBPARTITION east_2001 VALUES ('INDIA', 'PAKISTAN'),
        SUBPARTITION rest_2001 VALUES (DEFAULT)
    )
);

The following statement splits the europe_2000 subpartition into two subpartitions (europe_south_2000 and europe_north_2000):

ALTER TABLE salesHist
    SPLIT SUBPARTITION europe_2000 VALUES('ITALY') INTO
    (
        SUBPARTITION europe_south_2000,
        SUBPARTITION europe_north_2000
    );
12.8 ALTER TABLE... TRUNCATE PARTITION

Use the ALTER TABLE... TRUNCATE PARTITION command to remove all of 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 all of 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.

For more information about restrictions that apply to the ALTER TABLE... TRUNCATE PARTITION, please refer to the TRUNCATE command, in the PostgreSQL Core documentation at:


Parameters

table_name

The name (optionally schema-qualified) of the modified table.

partition_name

The name of the partition to be truncated.

DROP STORAGE and REUSE STORAGE are included for compatibility only; the clauses are parsed and ignored.

Example
To remove all of the data from the staff partition of the employees table, invoke the following command:

```
ALTER TABLE employees TRUNCATE PARTITION staff;
```
12.9 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

Use the ALTER TABLE... TRUNCATE SUBPARTITION command to remove all of the data from the 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.

For more information about restrictions that apply to the ALTER TABLE... TRUNCATE SUBPARTITION, please refer to the TRUNCATE command, in the PostgreSQL Core documentation at:

```
```

Parameters

`table_name`

The name (optionally schema-qualified) of the modified table.

`subpartition_name`

The name of the subpartition to be truncated.

DROP STORAGE and REUSE STORAGE are included for compatibility only; the clauses are parsed and ignored.

Example
To remove all of the data from the staff subpartition of the employees table, invoke the following command:

```
ALTER TABLE employees TRUNCATE PARTITION staff;
```
12.10 ALTER TABLE... EXCHANGE PARTITION

Use the ALTER TABLE... EXCHANGE PARTITION command to exchange an existing table with a partition or subpartition. The command syntax is available in two forms:

```
ALTER TABLE target_table
    EXCHANGE PARTITION target_partition
    WITH TABLE source_table
    [(WITH | WITHOUT) VALIDATION];
```

and

```
ALTER TABLE target_table
    EXCHANGE SUBPARTITION target_partition
    WITH TABLE source_table
    [(WITH | WITHOUT) VALIDATION];
```

Description

The ALTER TABLE... EXCHANGE PARTITION command swaps an existing table with a partition or subpartition. The structure of the source_table must match the structure of target_table (that is, both tables must have matching columns and data types). When this command completes, the data originally found in the target_partition can now be found in the source_table and the data originally found in the source_table can be found in the target_partition.

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.

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.

\textit{target\_partition}

The name of the partition or subpartition to be replaced.

\textit{source\_name}

The name of the table that will replace the \textit{target\_partition}.

\textbf{Example}

The following command replaces the 1st\_qtr partition of the \textit{sales} table with the contents of the \textit{ytd\_sales} table:

\begin{verbatim}
ALTER TABLE sales
 EXCHANGE PARTITION 1st_qtr
 WITH TABLE ytd_sales;
\end{verbatim}
12.11 ALTER TABLE... MOVE PARTITION

Use the ALTER TABLE... MOVE PARTITION command to move a partition to a different tablespace. Two forms of the command are supported; the syntax is:

```
ALTER TABLE table_name
  MOVE PARTITION partition_name
  TABLESPACE tablespace_name;
```

and

```
ALTER TABLE table_name
  MOVE SUBPARTITION partition_name
  TABLESPACE tablespace_name;
```

Description

This command 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.

You must own a table to invoke ALTER TABLE... MOVE PARTITION or ALTER TABLE... MOVE SUBPARTITION against that table.

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.

- **tablespace_name**
  - The name of the tablespace to which the partition or subpartition will be moved.

Example
The following command moves the 4th_quarter partition of the sales table into a tablespace named active_sales:

```sql
ALTER TABLE sales
    MOVE PARTITION 4th_quarter
    TABLESPACE active_sales;
```
12.12 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 partition_name
    TO new_name;
```

Description

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.

You must own a table to invoke ALTER TABLE... RENAME PARTITION or ALTER TABLE... RENAME SUBPARTITION against that table.

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
The following command renames a partition named (q4) that resides in the sales table to 4th_quarter:

```
ALTER TABLE sales
    RENAME PARTITION q4
    TO 4th_quarter;
```
12.13 Table Partitioning Views

The following Oracle-compatible catalog views provide access to information relating to table partitioning.

12.13.1 ALL_PART_TABLES

The following table lists the information available in the ALL_PART_TABLES view:

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>owner</td>
<td>name</td>
<td>The owner of the table.</td>
</tr>
<tr>
<td>table_name</td>
<td>name</td>
<td>The name of the table.</td>
</tr>
<tr>
<td>schema_name</td>
<td>name</td>
<td>The schema in which the table resides.</td>
</tr>
<tr>
<td>partitioning_type</td>
<td>text</td>
<td>RANGE or LIST</td>
</tr>
<tr>
<td>subpartitioning_type</td>
<td>text</td>
<td>RANGE, LIST or NONE</td>
</tr>
<tr>
<td>partition_count</td>
<td>bigint</td>
<td>The number of partitions.</td>
</tr>
<tr>
<td>def_subpartition_count</td>
<td>integer</td>
<td>The default subpartition count - this will always be 0.</td>
</tr>
<tr>
<td>partitioning_key_count</td>
<td>integer</td>
<td>The number of columns listed in the partition by clause.</td>
</tr>
<tr>
<td>subpartitioning_key_count</td>
<td>integer</td>
<td>The number of columns in the subpartition by clause.</td>
</tr>
<tr>
<td>status</td>
<td>character varying(8)</td>
<td>Provided for Oracle Compatibility - this column will always be VALID.</td>
</tr>
<tr>
<td>def_tablespace_name</td>
<td>character varying(30)</td>
<td>Provided for Oracle Compatibility - this column will always be NULL.</td>
</tr>
<tr>
<td>def_pct_free</td>
<td>numeric</td>
<td>Provided for Oracle Compatibility - this column will always be NULL.</td>
</tr>
<tr>
<td>def_pct_used</td>
<td>numeric</td>
<td>Provided for Oracle Compatibility - this column will always be NULL.</td>
</tr>
<tr>
<td>def_ini_trans</td>
<td>numeric</td>
<td>Provided for Oracle Compatibility - this column will always be NULL.</td>
</tr>
<tr>
<td>def_max_trans</td>
<td>numeric</td>
<td>Provided for Oracle Compatibility - this column will always be NULL.</td>
</tr>
<tr>
<td>def_initial_extent</td>
<td>character varying(40)</td>
<td>Provided for Oracle Compatibility - this column will always be NULL.</td>
</tr>
<tr>
<td>def_next_extent</td>
<td>character varying(40)</td>
<td>Provided for Oracle Compatibility - this column will always be NULL.</td>
</tr>
<tr>
<td>def_min_extents</td>
<td>character varying(40)</td>
<td>Provided for Oracle Compatibility - this column will always be NULL.</td>
</tr>
<tr>
<td>def_max_extents</td>
<td>character varying(40)</td>
<td>Provided for Oracle Compatibility - this column will always be NULL.</td>
</tr>
<tr>
<td>def_pct_increase</td>
<td>character varying(40)</td>
<td>Provided for Oracle Compatibility - this column will always be NULL.</td>
</tr>
<tr>
<td>def_freelists</td>
<td>numeric</td>
<td>Provided for Oracle Compatibility - this column will always be NULL.</td>
</tr>
<tr>
<td>def_freelist_groups</td>
<td>numeric</td>
<td>Provided for Oracle Compatibility - this column will always be NULL.</td>
</tr>
<tr>
<td>def_logging</td>
<td>character varying(7)</td>
<td>Provided for Oracle Compatibility - this column will always be YES</td>
</tr>
<tr>
<td>Column Name</td>
<td>Data Type</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------</td>
<td>----------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>def_compression</td>
<td>character varying(8)</td>
<td>Provided for Oracle Compatibility - this column will always be NONE</td>
</tr>
<tr>
<td>def_buffer_pool</td>
<td>character varying(7)</td>
<td>Provided for Oracle Compatibility - this column will always be DEFAULT</td>
</tr>
<tr>
<td>ref_ptn_constraint_name</td>
<td>character varying(30)</td>
<td>Provided for Oracle Compatibility - this column will always be NULL</td>
</tr>
<tr>
<td>interval</td>
<td>character varying(1000)</td>
<td>Provided for Oracle Compatibility - this column will always be NULL</td>
</tr>
</tbody>
</table>
### 12.13.2 ALL_TAB_PARTITIONS

The following table lists the information available in the ALL_TAB_PARTITIONS view:

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>table_owner</td>
<td>name</td>
<td>The owner of the table.</td>
</tr>
<tr>
<td>table_name</td>
<td>name</td>
<td>The name of the table.</td>
</tr>
<tr>
<td>schema_name</td>
<td>name</td>
<td>The schema in which the table resides.</td>
</tr>
<tr>
<td>composite</td>
<td>text</td>
<td>YES if the table is subpartitioned; NO if it is not subpartitioned.</td>
</tr>
<tr>
<td>partition_name</td>
<td>name</td>
<td>The name of the partition.</td>
</tr>
<tr>
<td>subpartition_count</td>
<td>bigint</td>
<td>The number of subpartitions for this partition.</td>
</tr>
<tr>
<td>high_value</td>
<td>text</td>
<td>The partition limit for RANGE partitions, or the partition value for LIST partitions.</td>
</tr>
<tr>
<td>high_value_length</td>
<td>integer</td>
<td>The length of high_value.</td>
</tr>
<tr>
<td>partition_position</td>
<td>integer</td>
<td>The ordinal position of this partition.</td>
</tr>
<tr>
<td>tablespace_name</td>
<td>name</td>
<td>The tablespace in which this partition resides.</td>
</tr>
<tr>
<td>pct_free</td>
<td>numeric</td>
<td>Provided for Oracle Compatibility - this column will always be 0.</td>
</tr>
<tr>
<td>pct_used</td>
<td>numeric</td>
<td>Provided for Oracle Compatibility - this column will always be 0.</td>
</tr>
<tr>
<td>ini_trans</td>
<td>numeric</td>
<td>Provided for Oracle Compatibility - this column will always be 0.</td>
</tr>
<tr>
<td>max_trans</td>
<td>numeric</td>
<td>Provided for Oracle Compatibility - this column will always be 0.</td>
</tr>
<tr>
<td>initial_extent</td>
<td>numeric</td>
<td>Provided for Oracle Compatibility - this column will always be NULL.</td>
</tr>
<tr>
<td>next_extent</td>
<td>numeric</td>
<td>Provided for Oracle Compatibility - this column will always be NULL.</td>
</tr>
<tr>
<td>min_extent</td>
<td>numeric</td>
<td>Provided for Oracle Compatibility - this column will always be 0.</td>
</tr>
<tr>
<td>max_extent</td>
<td>numeric</td>
<td>Provided for Oracle Compatibility - this column will always be 0.</td>
</tr>
<tr>
<td>pct_increase</td>
<td>numeric</td>
<td>Provided for Oracle Compatibility - this column will always be 0.</td>
</tr>
<tr>
<td>freelists</td>
<td>numeric</td>
<td>Provided for Oracle Compatibility - this column will always be NULL.</td>
</tr>
<tr>
<td>freelist_groups</td>
<td>numeric</td>
<td>Provided for Oracle Compatibility - this column will always be NULL.</td>
</tr>
<tr>
<td>logging</td>
<td>character varying(7)</td>
<td>Provided for Oracle Compatibility - this column will always be YES.</td>
</tr>
<tr>
<td>compression</td>
<td>character varying(8)</td>
<td>Provided for Oracle Compatibility - this column will always be NONE.</td>
</tr>
<tr>
<td>num_rows</td>
<td>numeric</td>
<td>The approx. number of rows in this partition.</td>
</tr>
<tr>
<td>blocks</td>
<td>integer</td>
<td>The approx. number of blocks in this partition.</td>
</tr>
<tr>
<td>empty_blocks</td>
<td>numeric</td>
<td>Provided for Oracle Compatibility - this column will always be NULL.</td>
</tr>
<tr>
<td>avg_space</td>
<td>numeric</td>
<td>Provided for Oracle Compatibility - this column will always be NULL.</td>
</tr>
<tr>
<td>chain_cnt</td>
<td>numeric</td>
<td>Provided for Oracle Compatibility - this column will always be NULL.</td>
</tr>
<tr>
<td>Column</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------</td>
<td>-----------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>avg_row_len</td>
<td>numeric</td>
<td>Provided for Oracle Compatibility - this column will always be NULL.</td>
</tr>
<tr>
<td>sample_size</td>
<td>numeric</td>
<td>Provided for Oracle Compatibility - this column will always be NULL.</td>
</tr>
<tr>
<td>last_analyzed</td>
<td>timestamp without time zone</td>
<td>Provided for Oracle Compatibility - this column will always be NULL.</td>
</tr>
<tr>
<td>buffer_pool</td>
<td>character varying(7)</td>
<td>Provided for Oracle Compatibility - this column will always be NULL.</td>
</tr>
<tr>
<td>global_stats</td>
<td>character varying(3)</td>
<td>Provided for Oracle Compatibility - this column will always be YES.</td>
</tr>
<tr>
<td>user_stats</td>
<td>character varying(3)</td>
<td>Provided for Oracle Compatibility - this column will always be NO.</td>
</tr>
<tr>
<td>backing_table</td>
<td>regclass</td>
<td>OID of the backing table for this partition.</td>
</tr>
</tbody>
</table>
### 12.13.3 ALL_TAB_SUBPARTITIONS

The following table lists the information available in the `ALL_TAB_SUBPARTITIONS` view:

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>table_owner</td>
<td>name</td>
<td>The name of the owner of the table.</td>
</tr>
<tr>
<td>table_name</td>
<td>name</td>
<td>The name of the table.</td>
</tr>
<tr>
<td>schema_name</td>
<td>name</td>
<td>The name of the schema in which the table resides.</td>
</tr>
<tr>
<td>partition_name</td>
<td>name</td>
<td>The name of the partition.</td>
</tr>
<tr>
<td>high_value</td>
<td>text</td>
<td>The subpartition limit for RANGE subpartitions, or the subpartition value for LIST subpartitions.</td>
</tr>
<tr>
<td>high_value_length</td>
<td>integer</td>
<td>The length of high_value.</td>
</tr>
<tr>
<td>subpartition_name</td>
<td>name</td>
<td>The name of the subpartition.</td>
</tr>
<tr>
<td>subpartition_position</td>
<td>integer</td>
<td>The ordinal position of this subpartition.</td>
</tr>
<tr>
<td>tablespace_name</td>
<td>name</td>
<td>The tablespace in which this subpartition resides.</td>
</tr>
<tr>
<td>pct_free</td>
<td>numeric</td>
<td>Provided for Oracle Compatibility - this column will always be 0.</td>
</tr>
<tr>
<td>pct_used</td>
<td>numeric</td>
<td>Provided for Oracle Compatibility - this column will always be 0.</td>
</tr>
<tr>
<td>ini_trans</td>
<td>numeric</td>
<td>Provided for Oracle Compatibility - this column will always be 0.</td>
</tr>
<tr>
<td>max_trans</td>
<td>numeric</td>
<td>Provided for Oracle Compatibility - this column will always be 0.</td>
</tr>
<tr>
<td>initial_extent</td>
<td>numeric</td>
<td>Provided for Oracle Compatibility - this column will always be NULL.</td>
</tr>
<tr>
<td>next_extent</td>
<td>numeric</td>
<td>Provided for Oracle Compatibility - this column will always be NULL.</td>
</tr>
<tr>
<td>min_extent</td>
<td>numeric</td>
<td>Provided for Oracle Compatibility - this column will always be 0.</td>
</tr>
<tr>
<td>max_extent</td>
<td>numeric</td>
<td>Provided for Oracle Compatibility - this column will always be 0.</td>
</tr>
<tr>
<td>pct_increase</td>
<td>numeric</td>
<td>Provided for Oracle Compatibility - this column will always be 0.</td>
</tr>
<tr>
<td>freelists</td>
<td>numeric</td>
<td>Provided for Oracle Compatibility - this column will always be NULL.</td>
</tr>
<tr>
<td>freelists_groups</td>
<td>numeric</td>
<td>Provided for Oracle Compatibility - this column will always be NULL.</td>
</tr>
<tr>
<td>logging</td>
<td>character varying(7)</td>
<td>Provided for Oracle Compatibility - this column will always be YES.</td>
</tr>
<tr>
<td>compression</td>
<td>character varying(8)</td>
<td>Provided for Oracle Compatibility - this column will always be NONE.</td>
</tr>
<tr>
<td>num_rows</td>
<td>numeric</td>
<td>The approx. number of rows in this subpartition.</td>
</tr>
<tr>
<td>blocks</td>
<td>integer</td>
<td>The approx. number of blocks in this subpartition.</td>
</tr>
<tr>
<td>empty_blocks</td>
<td>numeric</td>
<td>Provided for Oracle Compatibility - this column will always be NULL.</td>
</tr>
</tbody>
</table>
### Database Compatibility for Oracle® Developer’s Guide

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>avg_space</code></td>
<td>numeric</td>
<td>Provided for Oracle Compatibility - this column will always be NULL.</td>
</tr>
<tr>
<td><code>chain_cnt</code></td>
<td>numeric</td>
<td>Provided for Oracle Compatibility - this column will always be NULL.</td>
</tr>
<tr>
<td><code>avg_row_len</code></td>
<td>numeric</td>
<td>Provided for Oracle Compatibility - this column will always be NULL.</td>
</tr>
<tr>
<td><code>sample_size</code></td>
<td>numeric</td>
<td>Provided for Oracle Compatibility - this column will always be NULL.</td>
</tr>
<tr>
<td><code>last_analyzed</code></td>
<td>timestamp</td>
<td>Provided for Oracle Compatibility - this column will always be NULL.</td>
</tr>
<tr>
<td><code>buffer_pool</code></td>
<td>character varying(7)</td>
<td>Provided for Oracle Compatibility - this column will always be NULL.</td>
</tr>
<tr>
<td><code>global_stats</code></td>
<td>character varying(3)</td>
<td>Provided for Oracle Compatibility - this column will always be YES.</td>
</tr>
<tr>
<td><code>user_stats</code></td>
<td>character varying(3)</td>
<td>Provided for Oracle Compatibility - this column will always be NO.</td>
</tr>
<tr>
<td><code>backing_table</code></td>
<td>regclass</td>
<td>OID of the backing table for this subpartition.</td>
</tr>
</tbody>
</table>
12.13.4 ALL_PART_KEY_COLUMNS

The following table lists the information available in the `ALL_PART_KEY_COLUMNS` view:

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>owner</td>
<td>name</td>
<td>The name of the table owner.</td>
</tr>
<tr>
<td>name</td>
<td>name</td>
<td>The name of the table.</td>
</tr>
<tr>
<td>schema</td>
<td>name</td>
<td>The name of the schema on which the table resides.</td>
</tr>
<tr>
<td>object_type</td>
<td>character(5)</td>
<td>Provided for Oracle Compatibility - this column will always be TABLE.</td>
</tr>
<tr>
<td>column_name</td>
<td>name</td>
<td>The name of the partitioning key column.</td>
</tr>
<tr>
<td>column_position</td>
<td>integer</td>
<td>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...)</td>
</tr>
</tbody>
</table>

12.13.5 ALL_SUBPART_KEY_COLUMNS

The following table lists the information available in the `ALL_SUBPART_KEY_COLUMNS` view:

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>owner</td>
<td>name</td>
<td>The name of the table owner.</td>
</tr>
<tr>
<td>name</td>
<td>name</td>
<td>The name of the table.</td>
</tr>
<tr>
<td>schema</td>
<td>name</td>
<td>The name of the schema on which the table resides.</td>
</tr>
<tr>
<td>object_type</td>
<td>character(5)</td>
<td>Provided for Oracle Compatibility - this column will always be TABLE.</td>
</tr>
<tr>
<td>column_name</td>
<td>name</td>
<td>The name of the partitioning key column.</td>
</tr>
<tr>
<td>column_position</td>
<td>integer</td>
<td>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...)</td>
</tr>
</tbody>
</table>
**13 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:

- Pro*C compatible anonymous blocks.
- An Oracle-compatible **CALL** statement.

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 Postgres Plus Advanced Server ECPG Connector Guide, available from the EnterpriseDB website at:

13.1 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
#else
#endif
#elif expression
#endif
#define symbolName expansion
#define symbolName([macro arguments]) expansion
#undef symbolName
#if 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:

```c
#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 -D HAVE_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:

13.2 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.

### 13.3 Type Codes

The following table contains the type codes for external data 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:

<table>
<thead>
<tr>
<th>Type Code</th>
<th>Host Variable Type (C Data Type)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1, 2, 8, 11, 12, 15, 23, 24, 91, 94, 95, 96, 97</td>
<td><code>char[]</code></td>
</tr>
<tr>
<td>3</td>
<td><code>int</code></td>
</tr>
<tr>
<td>4, 7, 21</td>
<td><code>float</code></td>
</tr>
<tr>
<td>5, 6</td>
<td>null-terminated string (char[length+1])</td>
</tr>
<tr>
<td>9</td>
<td><code>varchar</code></td>
</tr>
<tr>
<td>22</td>
<td><code>double</code></td>
</tr>
<tr>
<td>68</td>
<td><code>unsigned int</code></td>
</tr>
</tbody>
</table>

The following table contains the type codes for internal data 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:

<table>
<thead>
<tr>
<th>Internal Type Code</th>
<th>Server Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>VARCHAR2</td>
</tr>
<tr>
<td>2</td>
<td>NUMBER</td>
</tr>
<tr>
<td>8</td>
<td>LONG</td>
</tr>
<tr>
<td>11</td>
<td>ROWID</td>
</tr>
<tr>
<td>12</td>
<td>DATE</td>
</tr>
<tr>
<td>23</td>
<td>RAW</td>
</tr>
<tr>
<td>24</td>
<td>LONG RAW</td>
</tr>
<tr>
<td>96</td>
<td>CHAR</td>
</tr>
<tr>
<td>100</td>
<td>BINARY FLOAT</td>
</tr>
<tr>
<td>101</td>
<td>BINARY DOUBLE</td>
</tr>
<tr>
<td>104</td>
<td>UROWID</td>
</tr>
<tr>
<td>187</td>
<td>TIMESTAMP</td>
</tr>
<tr>
<td>188</td>
<td>TIMESTAMP W/TIMEZONE</td>
</tr>
<tr>
<td>189</td>
<td>INTERVAL YEAR TO MONTH</td>
</tr>
<tr>
<td>190</td>
<td>INTERVAL DAY TO SECOND</td>
</tr>
<tr>
<td>232</td>
<td>TIMESTAMP LOCAL_TZ</td>
</tr>
</tbody>
</table>
13.4 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:

```c
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 actual number 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. See the displayResultSet() function for an example of how to allocate space for SELECT-list values (Section 5.4, Executing a Query with an Unknown Number of Variables).
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 (see Section 7.3 for a list of type codes). 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 and a 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:

```c
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:

```c
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 maximum number 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.
### 13.5 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:

```sql
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. For more information about substituting host variables for value expressions, please see Section 3.1.2, Declaring Host Variables.

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 see the PostgreSQL Core Documentation, available from the EnterpriseDB website at:


#### 13.5.1 ALLOCATE DESCRIPTOR

Use the **ALLOCATE DESCRIPTOR** statement to allocate an SQL descriptor area:

```sql
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;

13.5.2 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_variable` specifies 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;
```
13.5.3  **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.

13.5.4  **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

13.5.5 CONNECT

Use the CONNECT statement to establish a connection to a database. The CONNECT statement is available in two forms - one form is Oracle-compatible, the other is not.

The first form is Oracle-compatible:

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 password
USER 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 \texttt{DEFAULT} to establish a connection with the default database, using the default role name. If you specify \texttt{DEFAULT} as the target database, do not include a \texttt{connection\_name} or \texttt{credentials}.

\texttt{connection\_name} is the name of the connection to the database. \texttt{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 \texttt{connection\_name} for each connection.

If you do not specify a name for a connection, \texttt{ecpglib} assigns a name of \texttt{DEFAULT} to the connection. You can refer to the connection by name (\texttt{DEFAULT}) in any \texttt{EXEC SQL} statement.

\texttt{CURRENT} is the most recently opened or the connection mentioned in the most-recent \texttt{SET CONNECTION TO} statement. If you do not refer to a connection by name in an \texttt{EXEC SQL} statement, ECPG assumes the name of the connection to be \texttt{CURRENT}.

\texttt{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.

\texttt{password} is the password associated with the specified \texttt{user\_name}.

The following code fragment uses the second form of the \texttt{CONNECT} statement to establish a connection to a database named \texttt{edb}, using the role \texttt{alice} and the password associated with that role, \texttt{1safepwd}:

\begin{verbatim}
EXEC SQL CONNECT TO edb AS acctg_conn
    USER 'alice' IDENTIFIED BY '1safepwd';
\end{verbatim}

The name of the connection is \texttt{acctg\_conn}; you can use the connection name when changing the connection name using the \texttt{SET CONNECTION} statement.

\section*{13.5.6 DEALLOCATE DESCRIPTOR}

Use the \texttt{DEALLOCATE DESCRIPTOR} statement to free memory in use by an allocated descriptor. The syntax of the statement is:

\begin{verbatim}
EXEC SQL DEALLOCATE DESCRIPTOR \texttt{descriptor\_name}
\end{verbatim}

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;

13.5.7 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.
13.5.8 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.

13.5.9 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.

13.5.10 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:

```
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:

- `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 see the PostgreSQL Core documentation, available from EnterpriseDB at:


13.5.11 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_name INTO 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 input variable *required* by the statement. For example, given two descriptors:

```sql
SQLDA *query_values_in;
SQLDA *query_values_out;
```

You might prepare a query that returns information from the `emp` table:

```sql
EXEC SQL PREPARE get_emp FROM
  "SELECT ename, empno, sal FROM emp WHERE empno = ?";
```

The command requires one input variable (for the parameter marker (?)�).

```sql
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:

```sql
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 *executing* the 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.

### 13.5.12 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] DESCRIPTR 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 input variable 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 DESCRIPTR 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.
13.5.13 DISCONNECT

Use the DISCONNECT statement to close the connection to the server. The syntax is:

```sql
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:

```c
/* client.pgc*/
int main()
{
    EXEC SQL CONNECT TO hr AS connection_name;
    EXEC SQL DISCONNECT connection_name;
    return(0);
}
```

13.5.14 EXECUTE

Use the EXECUTE statement to execute a statement previously prepared using an EXEC SQL PREPARE statement. The syntax is:

```sql
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 DESCRIPTOR SQLDA_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;

13.5.15 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;
```

**13.5.16 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 Postgres Plus Advanced Server.

13.5.17 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;
```

13.5.18 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:

\textit{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 \texttt{FOR} clause, the statement is executed once for each member of the array.

\textit{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 \texttt{USING} clause, the \texttt{FETCH} statement will populate the specified SQLDA descriptor with the values returned by the server.

If you include an \texttt{INTO} clause, the \texttt{FETCH} statement will populate the host variables (and optional indicator variables) specified in the target list.

The following code fragment declares a cursor named \texttt{employees} that retrieves the employee number, name and salary from the \texttt{emp} table:

\begin{verbatim}
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;
\end{verbatim}

\section{13.5.19 FETCH DESCRIPTOR}

Use the \texttt{FETCH DESCRIPTOR} statement to retrieve rows from a cursor into an SQL descriptor. The syntax is:

\begin{verbatim}
EXEC SQL [FOR array\_size] FETCH cursor
   INTO [SQL] DESCRIPTOR descriptor\_name;
\end{verbatim}

Where:

\textit{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 \texttt{FOR} clause, the statement is executed once for each member of the array.

\textit{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 \texttt{DECLARE} and \texttt{OPEN} the cursor before calling the \texttt{FETCH DESCRIPTOR} statement.

Include the \texttt{INTO} clause to specify an SQL descriptor into which the \texttt{EXECUTE} statement will write the results returned by the prepared statement. \texttt{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:

```sql
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';
}
```

13.5.20 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.

```sql
EXEC SQL GET DESCRIPTOR descriptor_name:
  host_variable = COUNT;
```

The second form returns information about a specific value (specified by the VALUE
column_number clause).

```sql
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:

```sql
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.

### 13.5.21 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:
### INSERT INTO Table

```sql
INSERT INTO table [(column [, ...])]  
{DEFAULT VALUES |  
VALUES ( {expression | DEFAULT} [, ...]) [, ...] | query}  
[RETURNING * | output_expression [ [AS] output_name ] [, ...] ]
```

Where:

- **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:

```sql
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 see the PostgreSQL Core documentation, available from EnterpriseDB at:


### 13.5.22 OPEN

Use the **OPEN** statement to open a cursor. The syntax is:

```sql
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.

### 13.5.23 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;

13.5.24 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.
\textit{sql\_statement} may take the form of a \texttt{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 \texttt{PREPARE} the statement. When you \texttt{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 \texttt{add\_emp}) that inserts a record into the \texttt{emp} table:

\begin{verbatim}
EXEC SQL PREPARE add_emp (int, text, text, numeric) AS
    INSERT INTO emp VALUES($1, $2, $3, $4);
\end{verbatim}

Each time you invoke the statement, provide fresh parameter values for the statement:

\begin{verbatim}
EXEC SQL EXECUTE add_emp(8003, 'Davis', 'CLERK', 2000.00);
EXEC SQL EXECUTE add_emp(8004, 'Myer', 'CLERK', 2000.00);
\end{verbatim}

Please note: A client application must issue a \texttt{PREPARE} statement within each session in which a statement will be executed; prepared statements persist only for the duration of the current session.

\section*{13.5.25 ROLLBACK}

Use the \texttt{ROLLBACK} statement to abort the current transaction, and discard any updates made by the transaction. The syntax is:

\begin{verbatim}
EXEC SQL [AT \texttt{database\_name}] ROLLBACK [WORK]
    [ { TO [SAVEPOINT] \texttt{savepoint} } | RELEASE ]
\end{verbatim}

Where:

\texttt{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 \texttt{AT} clause, the statement will execute against the current default database.

Include the \texttt{TO} clause to abort any commands that were executed after the specified \texttt{savepoint}; use the \texttt{SAVEPOINT} statement to define the \texttt{savepoint}. If you omit the \texttt{TO} clause, the \texttt{ROLLBACK} statement will abort the transaction, discarding all updates.

Include the \texttt{RELEASE} clause to cause the application to execute an \texttt{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.

### 13.5.26 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;
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 from EnterpriseDB at:


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:

```sql
SELECT * FROM emp WHERE dept_no = ?;
```

This `SELECT` statement allows you to provide a value at run-time for the `dept_no` parameter marker.

### 13.5.28 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:

```sql
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:

```sql
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.

13.5.29 

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 describe the input parameters 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.

### 13.5.30 UPDATE

Use an `UPDATE` statement to modify the data stored in a table. The syntax is:

```sql
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 from EnterpriseDB at:


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 employees salary, by multiplying the current salary by the value in the `:increase` host variable:

```sql
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.

13.5.31 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:

<table>
<thead>
<tr>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOT FOUND</td>
<td>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.</td>
</tr>
<tr>
<td>SQLERROR</td>
<td>The server returns an SQLERROR condition when it encounters a serious error returned by an SQL statement.</td>
</tr>
<tr>
<td>SQLWARNING</td>
<td>The server returns an SQLWARNING condition when it encounters a non-fatal warning returned by an SQL statement.</td>
</tr>
</tbody>
</table>

The following table describes the actions that result from a client encountering a condition:

<table>
<thead>
<tr>
<th>Action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALL function[]</td>
<td>Instructs the client application to call the named function.</td>
</tr>
<tr>
<td>CONTINUE</td>
<td>Instructs the client application to proceed to the next statement.</td>
</tr>
<tr>
<td>DO BREAK</td>
<td>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.</td>
</tr>
<tr>
<td>DO CONTINUE</td>
<td>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.</td>
</tr>
<tr>
<td>DO function[]</td>
<td>Instructs the client application to call the named function.</td>
</tr>
<tr>
<td>GOTO label</td>
<td>Instructs the client application to proceed to the statement that contains the label.</td>
</tr>
<tr>
<td>GO TO label</td>
<td>Instructs the client application to proceed to the statement that contains the label.</td>
</tr>
<tr>
<td>SQLPRINT</td>
<td>Instructs the client application to print a message to standard error.</td>
</tr>
<tr>
<td>STOP</td>
<td>Instructs the client application to stop execution.</td>
</tr>
</tbody>
</table>
The following code fragment prints a message if the client application encounters a warning, and aborts the application if it encounters an error:

```sql
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:

```sql
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:

```sql
EXEC SQL WHENEVER NOT FOUND CALL error_handler(__LINE__);
14 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.

14.1.1 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.

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Modifiers</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dirname</td>
<td>&quot;name&quot;</td>
<td>not null</td>
<td>The name of the alias.</td>
</tr>
<tr>
<td>dirowner</td>
<td>oid</td>
<td>not null</td>
<td>The OID of the user that owns the alias.</td>
</tr>
<tr>
<td>dirpath</td>
<td>text</td>
<td></td>
<td>The directory name to which access is granted.</td>
</tr>
<tr>
<td>diracl</td>
<td>aclitem[]</td>
<td></td>
<td>The access control list that determines which users may access the alias.</td>
</tr>
</tbody>
</table>

14.1.2 pg_synonym

The pg_synonym table contains one row for each synonym created with the CREATE SYNONYM command or CREATE PUBLIC SYNONYM command.

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Modifiers</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>synname</td>
<td>&quot;name&quot;</td>
<td>not null</td>
<td>The name of the synonym.</td>
</tr>
<tr>
<td>synnamespace</td>
<td>oid</td>
<td>not null</td>
<td>Replaces synowner. Contains the OID of the pg_namespace row where the synonym is stored</td>
</tr>
<tr>
<td>synowner</td>
<td>oid</td>
<td>not null</td>
<td>The OID of the user that owns the synonym.</td>
</tr>
<tr>
<td>synobjschema</td>
<td>&quot;name&quot;</td>
<td>not null</td>
<td>The schema in which the referenced object is defined.</td>
</tr>
<tr>
<td>synobjname</td>
<td>&quot;name&quot;</td>
<td>not null</td>
<td>The name of the referenced object.</td>
</tr>
<tr>
<td>synlink</td>
<td>text</td>
<td></td>
<td>The (optional) name of the database link in which the referenced object is defined.</td>
</tr>
</tbody>
</table>
14.1.3  **edb_variable**

The `edb_variable` table contains one row for each package level variable (each variable declared within a package).

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Modifiers</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>varname</td>
<td>&quot;name&quot;</td>
<td>not null</td>
<td>The name of the variable.</td>
</tr>
<tr>
<td>varpackage</td>
<td>oid</td>
<td>not null</td>
<td>The OID of the <code>pg_namespace</code> row that stores the package.</td>
</tr>
<tr>
<td>vartype</td>
<td>oid</td>
<td>not null</td>
<td>The OID of the <code>pg_type</code> row that defines the type of the variable.</td>
</tr>
<tr>
<td>varaccess</td>
<td>&quot;char&quot;</td>
<td>not null</td>
<td>+ if the variable is visible outside of the package.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- if the variable is only visible within the package.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Note: Public variables are declared within the package header; private variables are declared within the package body.</td>
</tr>
<tr>
<td>varsrc</td>
<td>text</td>
<td></td>
<td>Contains the source of the variable declaration, including any default value expressions for the variable.</td>
</tr>
<tr>
<td>varseq</td>
<td>smallint</td>
<td>not null</td>
<td>The order in which the variable was declared in the package.</td>
</tr>
</tbody>
</table>

14.1.4  **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.

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>product</td>
<td>character varying (74)</td>
<td>The name of the product.</td>
</tr>
<tr>
<td>version</td>
<td>character varying (74)</td>
<td>The version number of the product.</td>
</tr>
<tr>
<td>status</td>
<td>character varying (74)</td>
<td>The status of the release.</td>
</tr>
</tbody>
</table>
15 Acknowledgements

The PostgreSQL 8.3, 8.4, 9.0, 9.1 and 9.2 Documentation provided the baseline for portions of this Oracle Compatibility Developer’s Guide that is common to PostgreSQL, and is hereby acknowledged:

Portions of this EnterpriseDB™ Software and Documentation may utilize the following copyrighted material, the use of which is hereby acknowledged.

PostgreSQL Documentation, Database Management System

PostgreSQL is Copyright © 1996-2012 by the PostgreSQL Global Development Group and is distributed under the terms of the license of the University of California below.

Postgres95 is Copyright © 1994-5 by the Regents of the University of California.

Permission to use, copy, modify, and distribute this software and its documentation for any purpose, without fee, and without a written agreement is hereby granted, provided that the above copyright notice and this paragraph and the following two paragraphs appear in all copies.

IN NO EVENT SHALL THE UNIVERSITY OF CALIFORNIA BE LIABLE TO ANY PARTY FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES, INCLUDING LOST PROFITS, ARISING OUT OF THE USE OF THIS SOFTWARE AND ITS DOCUMENTATION, EVEN IF THE UNIVERSITY OF CALIFORNIA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

THE UNIVERSITY OF CALIFORNIA SPECIFICALLY DISCLAIMS ANY WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE SOFTWARE PROVIDED HEREUNDER IS ON AN "AS-IS" BASIS, AND THE UNIVERSITY OF CALIFORNIA HAS NO OBLIGATIONS TO PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS.