



EDB

EDB Postgres™ Advanced Server

Release 13

Database Compatibility for Oracle® Developers SQL Guide

Nov 29, 2020

CONTENTS

1	Introduction	1
2	ALTER DIRECTORY	3
3	ALTER INDEX	5
4	ALTER PROCEDURE	7
5	ALTER PROFILE	9
6	ALTER QUEUE	13
7	ALTER QUEUE TABLE	17
8	ALTER ROLE... IDENTIFIED BY	19
9	ALTER ROLE - Managing Database Link and DBMS_RLS Privileges	21
10	ALTER SEQUENCE	25
11	ALTER SESSION	27
12	ALTER TABLE	29
13	ALTER TRIGGER	33
14	ALTER TABLESPACE	35
15	ALTER USER... IDENTIFIED BY	37
16	ALTER USER/ROLE... PROFILE MANAGEMENT CLAUSES	39
17	CALL	41
18	COMMENT	43
19	COMMIT	45
20	CREATE DATABASE	47
21	CREATE PUBLIC DATABASE LINK	49
22	CREATE DIRECTORY	61

23 CREATE FUNCTION	63
24 CREATE INDEX	69
25 CREATE MATERIALIZED VIEW	73
26 CREATE PACKAGE	75
27 CREATE PACKAGE BODY	77
28 CREATE PROCEDURE	83
29 CREATE PROFILE	89
30 CREATE QUEUE	93
31 CREATE QUEUE TABLE	95
32 CREATE ROLE	97
33 CREATE SCHEMA	99
34 CREATE SEQUENCE	101
35 CREATE SYNONYM	105
36 CREATE TABLE	107
37 CREATE TABLE AS	115
38 CREATE TRIGGER	117
39 CREATE TYPE	127
40 CREATE TYPE BODY	135
41 CREATE USER	139
42 CREATE USER ROLE... PROFILE MANAGEMENT CLAUSES	141
43 CREATE VIEW	143
44 DELETE	145
45 DROP DATABASE LINK	147
46 DROP DIRECTORY	149
47 DROP FUNCTION	151
48 DROP INDEX	153
49 DROP PACKAGE	155
50 DROP PROCEDURE	157
51 DROP PROFILE	159
52 DROP QUEUE	161

53 DROP QUEUE TABLE	163
54 DROP SYNONYM	165
55 DROP ROLE	167
56 DROP SEQUENCE	169
57 DROP TABLE	171
58 DROP TABLESPACE	173
59 DROP TRIGGER	175
60 DROP TYPE	177
61 DROP USER	179
62 DROP VIEW	181
63 EXEC	183
64 GRANT	185
64.1 GRANT on Database Objects	187
64.2 GRANT on Roles	189
64.3 GRANT on System Privileges	191
65 INSERT	193
66 LOCK	197
67 REVOKE	199
68 ROLLBACK	203
69 ROLLBACK TO SAVEPOINT	205
70 SAVEPOINT	207
71 SELECT	209
71.1 FROM Clause	211
71.2 WHERE Clause	212
71.3 GROUP BY Clause	213
71.4 HAVING Clause	214
71.5 SELECT List	214
71.6 UNION Clause	215
71.7 INTERSECT Clause	215
71.8 MINUS Clause	215
71.9 CONNECT BY Clause	216
71.10 ORDER BY Clause	216
71.11 DISTINCT UNIQUE Clause	218
71.12 FOR UPDATE Clause	218
72 SET CONSTRAINTS	221
73 SET ROLE	223

74 SET TRANSACTION	225
75 TRUNCATE	227
76 UPDATE	229
77 Conclusion	233
Index	235

INTRODUCTION

This guide provides a summary of the SQL commands compatible with Oracle databases that are supported by Advanced Server. The SQL commands in this section will work on both an Oracle database and an Advanced Server database.

Note the following points:

- 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 for each command. In most cases, syntax, options, and functionality that are not compatible with Oracle databases have been omitted from the command description and syntax.
- The Advanced Server documentation set documents command functionality that may not be compatible with Oracle databases.

ALTER DIRECTORY

Name

ALTER DIRECTORY -- change the owner of a directory created using *CREATE DIRECTORY* command.

Synopsis

```
ALTER DIRECTORY <name> OWNER TO <rolename>
```

Description

The ALTER DIRECTORY ... OWNER TO command changes the owner of a directory. You must have the superuser privilege to execute this command; the new owner of the directory must also have the superuser privilege.

Parameters

name

The name of the directory to be altered.

rolename

The name of an owner that will own the directory.

Examples

The following example demonstrates changing ownership; bob and carol are superusers. bob is a current owner of directory EMPDIR:

```
SELECT * FROM all_directories where directory_name = 'EMPDIR' order
by 1,2,3;
  owner | directory_name | directory_path
-----+-----+-----
  bob   | EMPDIR          | /path
(1 row)
```

To alter the ownership of directory EMPDIR to carol:

```
ALTER DIRECTORY EMPDIR OWNER TO carol;
ALTER DIRECTORY

SELECT * FROM all_directories where directory_name = 'EMPDIR' order by
1,2,3;
  owner | directory_name | directory_path
-----+-----+-----
  carol | EMPDIR          | /path
(1 row)
```

The ownership of a directory is altered and granted to `carol`.

See Also

CREATE DIRECTORY, DROP DIRECTORY

ALTER INDEX

Name

ALTER INDEX – modify an existing index.

Synopsis

Advanced Server supports three variations of the ALTER INDEX command compatible with Oracle databases. Use the first variation to rename an index:

```
ALTER INDEX <name> RENAME TO <new_name>
```

Use the second variation of the ALTER INDEX command to rebuild an index:

```
ALTER INDEX <name> REBUILD
```

Use the third variation of the ALTER INDEX command to set the PARALLEL or NOPARALLEL clause:

```
ALTER INDEX <name> { NOPARALLEL | PARALLEL [ <integer> ] }
```

Description

ALTER INDEX changes the definition of an existing index. The RENAME clause changes the name of the index. The REBUILD clause reconstructs an index, replacing the old copy of the index with an updated version based on the index's table.

The REBUILD clause invokes the PostgreSQL REINDEX command; for more information about using the REBUILD clause, see the PostgreSQL core documentation at:

<https://www.postgresql.org/docs/current/static/sql-reindex.html>

The PARALLEL clause sets the degree of parallelism for an index that can be used to parallelize the rebuilding of an index.

The NOPARALLEL clause resets parallelism to use default values; reloptions will show the parallel_workers parameter as 0.

ALTER INDEX has no effect on stored data.

Parameters

name

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

new_name

New name for the index.

PARALLEL

Include the `PARALLEL` clause to specify a degree of parallelism; set the `parallel_workers` parameter equal to the degree of parallelism for the rebuilding of an index. If you specify `PARALLEL` but no degree of parallelism is provided, the server enforces default parallelism.

`NOPARALLEL`

Specify `NOPARALLEL` to reset parallelism to default values.

`integer`

The `integer` indicates the degree of parallelism (the number of `parallel_workers` used when rebuilding an index).

Examples

To change the name of an index from `name_idx` to `empname_idx`:

```
ALTER INDEX name_idx RENAME TO empname_idx;
```

To rebuild an index named `empname_idx`:

```
ALTER INDEX empname_idx REBUILD;
```

The following example sets the degree of parallelism on an `empname_idx` index to 7:

```
ALTER INDEX empname_idx PARALLEL 7;
```

See Also

[CREATE INDEX](#), [DROP INDEX](#)

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;
```

See Also

CREATE PROCEDURE, DROP PROCEDURE

ALTER PROFILE

Name

ALTER PROFILE – alter an existing profile

Synopsis

```
ALTER PROFILE <profile_name> RENAME TO <new_name>;

ALTER PROFILE <profile_name>
    LIMIT {<parameter value>}[...];
```

Description

Use the ALTER PROFILE command to modify a user-defined profile; Advanced Server supports two forms of the command:

- Use ALTER PROFILE...RENAME TO to change the name of a profile.
- Use ALTER PROFILE...LIMIT to modify the limits associated with a profile.

Include the LIMIT clause and one or more space-delimited parameter/value pairs to specify the rules enforced by Advanced Server, or use ALTER PROFILE...RENAME TO to change the name of a profile.

Parameters

profile_name

The name of the profile.

new_name

new_name specifies the new name of the profile.

parameter

parameter specifies the attribute limited by the profile.

value

value specifies the parameter limit.

Advanced Server supports the value shown below for each parameter:

FAILED_LOGIN_ATTEMPTS specifies the number of failed login attempts that a user may make before the server locks the user out of their account for the length of time specified by PASSWORD_LOCK_TIME. Supported values are:

- An INTEGER value greater than 0.
- DEFAULT- the value of FAILED_LOGIN_ATTEMPTS specified in the DEFAULT profile.

- **DEFAULT** - the value of `FAILED_LOGIN_ATTEMPTS` specified in the **DEFAULT** profile.
- **UNLIMITED** – the connecting user may make an unlimited number of failed login attempts.

`PASSWORD_LOCK_TIME` specifies the length of time that must pass before the server unlocks an account that has been locked because of `FAILED_LOGIN_ATTEMPTS`. Supported values are:

- A **NUMERIC** value greater than or equal to 0. To specify a fractional portion of a day, specify a decimal value. For example, use the value 4.5 to specify 4 days, 12 hours.
- **DEFAULT** - the value of `PASSWORD_LOCK_TIME` specified in the **DEFAULT** profile.
- **UNLIMITED** – the account is locked until it is manually unlocked by a database superuser.

`PASSWORD_LIFE_TIME` specifies the number of days that the current password may be used before the user is prompted to provide a new password. Include the `PASSWORD_GRACE_TIME` clause when using the `PASSWORD_LIFE_TIME` clause to specify the number of days that will pass after the password expires before connections by the role are rejected. If `PASSWORD_GRACE_TIME` is not specified, the password will expire on the day specified by the default value of `PASSWORD_GRACE_TIME`, and the user will not be allowed to execute any command until a new password is provided. Supported values are:

- A **NUMERIC** value greater than or equal to 0. To specify a fractional portion of a day, specify a decimal value. For example, use the value 4.5 to specify 4 days, 12 hours.
- **DEFAULT** - the value of `PASSWORD_LIFE_TIME` specified in the **DEFAULT** profile.
- **UNLIMITED** – The password does not have an expiration date.

`PASSWORD_GRACE_TIME` specifies the length of the grace period after a password expires until the user is forced to change their password. When the grace period expires, a user will be allowed to connect, but will not be allowed to execute any command until they update their expired password. Supported values are:

- A **NUMERIC** value greater than or equal to 0. To specify a fractional portion of a day, specify a decimal value. For example, use the value 4.5 to specify 4 days, 12 hours.
- **DEFAULT** - the value of `PASSWORD_GRACE_TIME` specified in the **DEFAULT** profile.
- **UNLIMITED** – The grace period is infinite.

`PASSWORD_REUSE_TIME` specifies the number of days a user must wait before re-using a password. The `PASSWORD_REUSE_TIME` and `PASSWORD_REUSE_MAX` parameters are intended to be used together. If you specify a finite value for one of these parameters while the other is **UNLIMITED**, old passwords can never be reused. If both parameters are set to **UNLIMITED** there are no restrictions on password reuse. Supported values are:

- A **NUMERIC** value greater than or equal to 0. To specify a fractional portion of a day, specify a decimal value. For example, use the value 4.5 to specify 4 days, 12 hours.
- **DEFAULT** - the value of `PASSWORD_REUSE_TIME` specified in the **DEFAULT** profile.
- **UNLIMITED** – The password can be re-used without restrictions.

`PASSWORD_REUSE_MAX` specifies the number of password changes that must occur before a password can be reused. The `PASSWORD_REUSE_TIME` and `PASSWORD_REUSE_MAX` parameters are intended to be used together. If you specify a finite value for one of these parameters while the other is **UNLIMITED**, old passwords can never be reused. If both parameters are set to **UNLIMITED** there are no restrictions on password reuse. Supported values are:

- An **INTEGER** value greater than or equal to 0.
- **DEFAULT** - the value of `PASSWORD_REUSE_MAX` specified in the **DEFAULT** profile.
- **UNLIMITED** – The password can be re-used without restrictions.

PASSWORD_VERIFY_FUNCTION specifies password complexity. Supported values are:

- The name of a PL/SQL function.
- DEFAULT - the value of PASSWORD_VERIFY_FUNCTION specified in the DEFAULT profile.
- NULL

PASSWORD_ALLOW_HASHED specifies whether an encrypted password to be allowed for use or not. If you specify the value as TRUE, the system allows a user to change the password by specifying a hash computed encrypted password on the client side. However, if you specify the value as FALSE, then a password must be specified in a plain-text form in order to be validated effectively, else an error will be thrown if a server receives an encrypted password. Supported values are:

- A BOOLEAN value TRUE/ON/YES/1 or FALSE/OFF/NO/0.
- DEFAULT – the value of PASSWORD_ALLOW_HASHED specified in the DEFAULT profile.

Note: The PASSWORD_ALLOW_HASHED is not an Oracle-compatible parameter.

Examples

The following example modifies a profile named acctg_profile:

```
ALTER PROFILE acctg_profile
    LIMIT FAILED_LOGIN_ATTEMPTS 3 PASSWORD_LOCK_TIME 1;
```

acctg_profile will count failed connection attempts when a login role attempts to connect to the server. The profile specifies that if a user has not authenticated with the correct password in three attempts, the account will be locked for one day.

The following example changes the name of acctg_profile to payables_profile:

```
ALTER PROFILE acctg_profile RENAME TO payables_profile;
```

See Also

CREATE PROFILE, DROP PROFILE

ALTER QUEUE

Advanced Server includes extra syntax (not offered by Oracle) with the `ALTER QUEUE SQL` command. This syntax can be used in association with the `DBMS_AQADM` package.

Name

`ALTER QUEUE` -- allows a superuser or a user with the `aq_administrator_role` privilege to modify the attributes of a queue.

Synopsis

This command is available in four forms. The first form of this command changes the name of a queue.

```
ALTER QUEUE <queue_name> RENAME TO <new_name>
```

Parameters

`queue_name`

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

`RENAME TO`

Include the `RENAME TO` clause and a new name for the queue to rename the queue.

`new_name`

New name for the queue.

The second form of the `ALTER QUEUE` command modifies the attributes of the queue:

```
ALTER QUEUE <queue_name> SET [ ( { <option_name option_value> } [, SET  
<option_name>
```

Parameters

`queue_name`

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

Include the `SET` clause and `option_name/option_value` pairs to modify the attributes of the queue:

`option_name option_value`

The name of the option or options to be associated with the new queue and the corresponding value of the option. If you provide duplicate option names, the server will return an error.

- If `option_name` is `retries`, provide an integer that represents the number of times a dequeue may be attempted.

- If `option_name` is `retrydelay`, provide a double-precision value that represents the delay in seconds.
- If `option_name` is `retention`, provide a double-precision value that represents the retention time in seconds.

Use the third form of the `ALTER QUEUE` command to enable or disable enqueueing and/or dequeueing on a particular queue:

```
ALTER QUEUE <queue_name> ACCESS { START | STOP } [ FOR { enqueue | dequeue }  
] [ NOWAIT ]
```

Parameters

`queue_name`

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

`ACCESS`

Include the `ACCESS` keyword to enable or disable enqueueing and/or dequeueing on a particular queue.

`START | STOP`

Use the `START` and `STOP` keywords to indicate the desired state of the queue.

`FOR enqueue | dequeue`

Use the `FOR` clause to indicate if you are specifying the state of enqueueing or dequeueing activity on the specified queue.

`NOWAIT`

Include the `NOWAIT` keyword to specify that the server should not wait for the completion of outstanding transactions before changing the state of the queue. The `NOWAIT` keyword can only be used when specifying an `ACCESS` value of `STOP`. The server will return an error if `NOWAIT` is specified with an `ACCESS` value of `START`.

Use the fourth form to `ADD` or `DROP` callback details for a particular queue.

```
ALTER QUEUE <queue_name> { ADD | DROP } CALL TO <location_name> [ WITH  
<callback_option> ]
```

Parameters

`queue_name`

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

`ADD | DROP`

Include the `ADD` or `DROP` keywords to enable add or remove callback details for a queue.

`location_name`

`location_name` specifies the name of the callback procedure.

`callback_option`

`callback_option` can be `context`; specify a `RAW` value when including this clause.

Examples

The following example changes the name of a queue from `work_queue_east` to `work_order`:

```
ALTER QUEUE work_queue_east RENAME TO work_order;
```

The following example modifies a queue named `work_order`, setting the number of retries to 100, the delay between retries to 2 seconds, and the length of time that the queue will retain dequeued messages to 10 seconds:

```
ALTER QUEUE work_order SET (retries 100, retrydelay 2, retention 10);
```

The following commands enable enqueueing and dequeueing in a queue named `work_order`:

```
ALTER QUEUE work_order ACCESS START;  
ALTER QUEUE work_order ACCESS START FOR enqueue;  
ALTER QUEUE work_order ACCESS START FOR dequeue;
```

The following commands disable enqueueing and dequeueing in a queue named `work_order`:

```
ALTER QUEUE work_order ACCESS STOP NOWAIT;  
ALTER QUEUE work_order ACCESS STOP FOR enqueue;  
ALTER QUEUE work_order ACCESS STOP FOR dequeue;
```

See Also

CREATE QUEUE, DROP QUEUE

ALTER QUEUE TABLE

Advanced Server includes extra syntax (not offered by Oracle) with the `ALTER QUEUE SQL` command. This syntax can be used in association with the `DBMS_AQADM` package.

Name

`ALTER QUEUE TABLE`-- modify an existing queue table.

Synopsis

Use `ALTER QUEUE TABLE` to change the name of an existing queue table:

```
ALTER QUEUE TABLE <name> RENAME TO <new_name>
```

Description

`ALTER QUEUE TABLE` allows a superuser or a user with the `aq_administrator_role` privilege to change the name of an existing queue table.

Parameters

`name`

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

`new_name`

New name for the queue table.

Examples

To change the name of a queue table from `wo_table_east` to `work_order_table`:

```
ALTER QUEUE TABLE wo_queue_east RENAME TO work_order_table;
```

See Also

CREATE QUEUE TABLE, DROP QUEUE TABLE

ALTER ROLE... IDENTIFIED BY

Name

ALTER ROLE - change the password associated with a database role

Synopsis

```
ALTER ROLE <role_name> IDENTIFIED BY <password>
        [REPLACE <prev_password>]
```

Description

A role without the `CREATEROLE` privilege may use this command to change their own password. An unprivileged role must include the `REPLACE` clause and their previous password if `PASSWORD_VERIFY_FUNCTION` is not `NULL` in their profile. When the `REPLACE` clause is used by a non-superuser, the server will compare the password provided to the existing password and raise an error if the passwords do not match.

A database superuser can use this command to change the password associated with any role. If a superuser includes the `REPLACE` clause, the clause is ignored; a non-matching value for the previous password will not throw an error.

If the role for which the password is being changed has the `SUPERUSER` attribute, then a superuser must issue this command. A role with the `CREATEROLE` attribute can use this command to change the password associated with a role that is not a superuser.

Parameters

`role_name`

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

`password`

The role's new password.

`prev_password`

The role's previous password.

Examples

To change a role's password:

```
ALTER ROLE john IDENTIFIED BY xyRP35z REPLACE 23PJ74a;
```

ALTER ROLE - MANAGING DATABASE LINK AND DBMS_RLS PRIVILEGES

Advanced Server 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 database links compatible with Oracle databases, 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]
```

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]
```

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]
```

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]
```

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

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 key words, 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 compatible with Oracle databases.

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 key words, 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 compatible with Oracle databases.

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). 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 compatible with Oracle databases.

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](#)

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 compatibility with Oracle syntax and have no effect whatsoever on the runtime behavior of Advanced Server. Others will alter a corresponding 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 Advanced Server `datestyle` runtime configuration parameter.

NLS_LANGUAGE (string)

Sets the language in which messages are displayed. Has the same effect as setting the 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 the 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 Advanced Server.

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 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 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 Advanced Server. This form is not compatible with Oracle databases.

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

Set the date display format.

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

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>
ALTER TABLE <name>
    { NOPARALLEL | PARALLEL [ <integer> ] }
```

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; for details, see [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.

The `PARALLEL` clause sets the degree of parallelism for a table. The `NOPARALLEL` clause resets the values to their defaults; `reloptions` will show the `parallel_workers` parameter as 0.

A superuser has permission to create a trigger on any user's table but a user can create a trigger only on the table they own. However, when the ownership of a table is changed, the ownership of the trigger's implicit objects is updated when they are matched with a table owner owning a trigger.

You can use `ALTER TRIGGER ...ON AUTHORIZATION` command to alter a trigger's implicit object owner, for information, see [ALTER TRIGGER](#).

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.

`PARALLEL`

Specify `PARALLEL` to select a degree of parallelism; you can also specify the degree of parallelism by setting the `parallel_workers` parameter when performing a parallel scan on a table. If you specify `PARALLEL` without including a degree of parallelism, the index will use default parallelism.

`NOPARALLEL`

Specify `NOPARALLEL` to reset parallelism to default values.

`integer`

The `integer` indicates the degree of parallelism, which is the number of `parallel_workers` used in the parallel operation to perform a parallel scan on a table.

Notes

When you invoke `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;
```

To reset the degree of parallelism to 0 on the `emp` table:

```
ALTER TABLE emp NOPARALLEL;
```

The following example creates a table named `dept`, and then alters the `dept` table to define and enable a unique key on the `dname` column. The constraint `dept_dname_uq` identifies the `dname` column as a unique key. The `USING_INDEX` clause creates an index on a table `dept` with the index statement specified to enable the unique constraint.

```
CREATE TABLE dept (
  deptno      NUMBER(2),
  dname       VARCHAR2(14),
  loc         VARCHAR2(13)
);
```

```
ALTER TABLE dept
  ADD CONSTRAINT dept_dname_uq UNIQUE(dname)
  USING INDEX (CREATE UNIQUE INDEX idx_dept_dname_uq ON dept (dname));
```

The following example creates a table named `emp`, and then alters the `emp` table to define and enable a primary key on the `ename` column. The `emp_ename_pk` constraint identifies the column `ename` as a primary key of the `emp` table. The `USING_INDEX` clause creates an index on a table `emp` with the index statement specified to enable the primary constraint.

```
CREATE TABLE emp (  
    empno          NUMBER(4) NOT NULL,  
    ename          VARCHAR2(10),  
    job            VARCHAR2(9),  
    sal            NUMBER(7,2),  
    deptno         NUMBER(2)  
);
```

```
ALTER TABLE emp  
    ADD CONSTRAINT emp_ename_pk PRIMARY KEY (ename)  
        USING INDEX (CREATE INDEX idx_emp_ename_pk ON emp (ename));
```

See Also

CREATE TABLE, DROP TABLE

ALTER TRIGGER

Name

ALTER TRIGGER -- change the definition of a trigger

Synopsis

Advanced Server supports three variations of the ALTER TRIGGER command. Use the first variation to change the name of a given trigger without changing the trigger definition.

```
ALTER TRIGGER <name> ON <table_name> RENAME TO <new_name>
```

Use the second variation of the ALTER TRIGGER command if the trigger is dependent on an extension; if the extension is dropped the trigger will automatically be dropped as well.

```
ALTER TRIGGER <name> ON <table_name> DEPENDS ON EXTENSION <extension_name>
```

Use the third variation of the ALTER TRIGGER command to change the ownership of a trigger's object.

```
ALTER TRIGGER <name> ON <table_name> AUTHORIZATION <rolespec>
```

For information about using non-compatible implementations of the ALTER TRIGGER command that are supported by Advanced Server, see the PostgreSQL core documentation at:

<https://www.postgresql.org/docs/current/sql-altertrigger.html>

Description

ALTER TRIGGER changes the properties of existing trigger. You must own the table on which the trigger acts to be allowed to change its properties.

To alter an owner of the trigger's implicit object you can use the ALTER TRIGGER ...ON AUTHORIZATION command. You must have the privilege to execute ALTER TRIGGER ...ON AUTHORIZATION command to assign the trigger's implicit object ownership to a user after authorization.

Parameters

name

The name of the trigger to be altered.

table_name

The name of a table on which trigger acts.

rolespec

The rolespec determines an owner of trigger objects.

Examples

The following example includes user `bob` and `carol` as superusers. The user `bob` owns a table `emp` and user `carol` owns a trigger named `emp_sal_trig`, which is created on table `emp`:

```
SELECT relname, relowner::regrole FROM pg_class WHERE relname = 'emp';
 relname | relowner
-----+-----
 emp     | bob
(1 row)
```

```
SELECT pronomame, proowner::regrole FROM pg_proc WHERE oid = (SELECT tgfooid
FROM pg_trigger WHERE tname = 'emp_sal_trig') ORDER BY oid;
 pronomame | proowner
-----+-----
 emp_sal_trig_emp | carol
(1 row)
```

To alter the ownership of table `emp` from user `bob` to a new owner `edb`:

```
ALTER TABLE emp OWNER TO edb;
ALTER TABLE

SELECT relname, relowner::regrole FROM pg_class WHERE relname = 'emp';
 relname | relowner
-----+-----
 emp     | edb
(1 row)
```

The table ownership is changed from user `bob` to an owner `edb` but the trigger ownership of `emp_sal_trig` is not altered and owned by user `carol`. Now alter the trigger `emp_sal_trig` on table `emp` and grant authorization to an owner `edb`:

```
ALTER TRIGGER emp_sal_trig ON emp AUTHORIZATION edb;
ALTER TRIGGER

SELECT pronomame, proowner::regrole FROM pg_proc WHERE oid = (SELECT tgfooid
FROM pg_trigger WHERE tname = 'emp_sal_trig') ORDER BY oid;
 pronomame | proowner
-----+-----
 emp_sal_trig_emp | edb
(1 row)
```

The trigger ownership `emp_sal_trig` on table `emp` is altered and granted to an owner `edb`.

See Also

CREATE TRIGGER, DROP TRIGGER

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

ALTER USER... IDENTIFIED BY

Name

ALTER USER -- change a database user account

Synopsis

```
ALTER USER <role_name> IDENTIFIED BY <password> REPLACE <prev_password>
```

Description

A role without the `CREATEROLE` privilege may use this command to change their own password. An unprivileged role must include the `REPLACE` clause and their previous password if `PASSWORD_VERIFY_FUNCTION` is not `NULL` in their profile. When the `REPLACE` clause is used by a non-superuser, the server will compare the password provided to the existing password and raise an error if the passwords do not match.

A database superuser can use this command to change the password associated with any role. If a superuser includes the `REPLACE` clause, the clause is ignored; a non-matching value for the previous password will not throw an error.

If the role for which the password is being changed has the `SUPERUSER` attribute, then a superuser must issue this command. A role with the `CREATEROLE` attribute can use this command to change the password associated with a role that is not a superuser.

Parameters

`role_name`

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

`password`

The role's new password.

`prev_password`

The role's previous password.

Examples

Change a user password:

```
ALTER USER john IDENTIFIED BY xyRP35z REPLACE 23PJ74a;
```

See Also

CREATE USER, DROP USER

ALTER USER|ROLE... PROFILE MANAGEMENT CLAUSES

Name

ALTER USER|ROLE

Synopsis

```
ALTER USER|ROLE <name> [[WITH] option[...]]
```

where *option* can be the following compatible clauses:

```
PROFILE <profile_name>
| ACCOUNT {LOCK|UNLOCK}
| PASSWORD EXPIRE [AT '<timestamp>']
```

or *option* can be the following non-compatible clauses:

```
| PASSWORD SET AT '<timestamp>'
| LOCK TIME '<timestamp>'
| STORE PRIOR PASSWORD {'<password>' '<timestamp>'} [, ...]
```

For information about the administrative clauses of the ALTER USER or ALTER ROLE command that are supported by Advanced Server, see the PostgreSQL core documentation available at:

<https://www.postgresql.org/docs/current/static/sql-commands.html>

Only a database superuser can use the ALTER USER|ROLE clauses that enforce profile management. The clauses enforce the following behaviors:

Include the PROFILE clause and a *profile_name* to associate a pre-defined profile with a role, or to change which pre-defined profile is associated with a user.

Include the ACCOUNT clause and the LOCK or UNLOCK keyword to specify that the user account should be placed in a locked or unlocked state.

Include the LOCK TIME '*timestamp*' clause and a date/time value to lock the role at the specified time, and unlock the role at the time indicated by the PASSWORD_LOCK_TIME parameter of the profile assigned to this role. If LOCK TIME is used with the ACCOUNT LOCK clause, the role can only be unlocked by a database superuser with the ACCOUNT UNLOCK clause.

Include the PASSWORD EXPIRE clause with the AT '*timestamp*' keywords to specify a date/time when the password associated with the role will expire. If you omit the AT '*timestamp*' keywords, the password will expire immediately.

Include the PASSWORD SET AT '*timestamp*' keywords to set the password modification date to the time specified.

Include the `STORE PRIOR PASSWORD {'password' 'timestamp' [, ...]}` clause to modify the password history, adding the new password and the time the password was set.

Each login role may only have one profile. To discover the profile that is currently associated with a login role, query the `profile` column of the `DBA_USERS` view.

Parameters

`name`

The name of the role with which the specified profile will be associated.

`password`

The password associated with the role.

`profile_name`

The name of the profile that will be associated with the role.

`timestamp`

The date and time at which the clause will be enforced. When specifying a value for `timestamp`, enclose the value in single-quotes.

Notes

For information about the Postgres-compatible clauses of the `ALTER USER` or `ALTER ROLE` command, see the PostgreSQL core documentation available at:

<https://www.postgresql.org/docs/current/static/sql-alterrole.html>

Examples

The following command uses the `ALTER USER... PROFILE` command to associate a profile named `acctg` with a user named `john`:

```
ALTER USER john PROFILE acctg_profile;
```

The following command uses the `ALTER ROLE... PROFILE` command to associate a profile named `acctg` with a user named `john`:

```
ALTER ROLE john PROFILE acctg_profile;
```

See Also

CREATE USER|ROLE... PROFILE MANAGEMENT CLAUSES

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.

Note: You must specify an OUT parameter in the CALL statement when calling a package function; the OUT parameter will act as an INOUT parameter during package overloading.

Examples

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

```
CALL update_balance();  
CALL update_balance(1,2,3);
```


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 a comment.

Examples

Attach a comment to the table `emp`:

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

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

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

Remove these comments:

```
COMMENT ON TABLE emp IS '';
COMMENT ON COLUMN emp.empno IS '';
```

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.

Note: Executing a COMMIT in a plpgsql procedure will throw an error if there is an Oracle-style SPL procedure on the runtime stack.

Examples

To commit the current transaction and make all changes permanent:

```
COMMIT;
```

See Also

ROLLBACK, ROLLBACK TO SAVEPOINT

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;
```


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 { postgres_fdw '<fdw_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 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.

Note: A database link cannot be used to access a remote database within a standby database server. Standby database servers are used for high availability, load balancing, and replication.

For information about high availability, load balancing, and replication for Postgres database servers, see the PostgreSQL core documentation available at:

<https://www.postgresql.org/docs/current/static/high-availability.html>

Note: For Advanced Server 12, the CREATE DATABASE LINK command is tested against and certified for use with Oracle version 10g Release 2 (10.2), Oracle version 11g Release 2 (11.2), and Oracle version 12c Release 1 (12.1).

Note: The edb_dblink_oci.rescans GUC can be set to SCROLL or SERIALIZABLE at the server level in postgresql.conf file. It can also be set at session level using the SET command, but the setting will not be applied to existing dblink connections due to dblink connection caching.

The `edb_dblink_oci` supports both types of rescans: `SCROLL` and `SERIALIZABLE`. By default it is set to `SERIALIZABLE`. When set to `SERIALIZABLE`, `edb_dblink_oci` uses the `SERIALIZABLE` transaction isolation level on the Oracle side, which corresponds to PostgreSQL's `REPEATABLE READ`:

- This is necessary as a single PostgreSQL statement can lead to multiple Oracle queries and thereby uses a serializable isolation level to provide consistent results.
- A serialization failure may occur due to a table modification concurrent with long-running DML transactions (for example `ADD`, `UPDATE`, or `DELETE` statements). If such a failure occurs, the OCI reports `ORA-08177: can't serialize access for this transaction`, and the application must retry the transaction.
- A `SCROLL` rescan will be quick, but with each iteration will reset the current row position to 1. A `SERIALIZABLE` rescan has performance benefits over a `SCROLL` rescan.

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`.

`postgres_fdw`

Specifies foreign data wrapper `postgres_fdw` as the connection to a remote Advanced Server database. If `postgres_fdw` has not been installed on the database, use the `CREATE EXTENSION` command to install `postgres_fdw`. For more information, see the `CREATE EXTENSION` command in the PostgreSQL Core documentation at: <https://www.postgresql.org/docs/current/static/sql-createextension.html>

`fdw_connection_string`

Specify the connection information for the `postgres_fdw` foreign data wrapper.

`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.

Setting up an Oracle Instant Client for `oci-dblink`

In order to use `oci-dblink`, an Oracle instant client must be downloaded and installed on the host running the Advanced Server database in which the database link is to be created.

An instant client can be downloaded from the following site:

<http://www.oracle.com/technetwork/database/features/instant-client/index-097480.html>

Oracle Instant Client for Linux

The following instructions apply to Linux hosts running Advanced Server.

Be sure the `libaio` library (the Linux-native asynchronous I/O facility) has already been installed on the Linux host running Advanced Server.

The `libaio` library can be installed with the following command:

```
yum install libaio
```

If the Oracle instant client that you've downloaded does not include the file specifically named `libclntsh.so` without a version number suffix, you must create a symbolic link named `libclntsh.so` that points to the downloaded version of the library file. Navigate to the instant client directory and execute the following command:

```
ln -s libclntsh.so.version libclntsh.so
```

Where `version` is the version number of the `libclntsh.so` library. For example:

```
ln -s libclntsh.so.12.1 libclntsh.so
```

When you are executing a SQL command that references a database link to a remote Oracle database, Advanced Server must know where the Oracle instant client library resides on the Advanced Server host.

The `LD_LIBRARY_PATH` environment variable must include the path to the Oracle client installation directory containing the `libclntsh.so` file. For example, assuming the installation directory containing `libclntsh.so` is `/tmp/instantclient`:

```
export LD_LIBRARY_PATH=/tmp/instantclient:$LD_LIBRARY_PATH
```

Note: The `LD_LIBRARY_PATH` environment variable setting must be in effect when the `pg_ctl` utility is executed to start or restart Advanced Server.

If you are running the current session as the user account (for example, `enterprisedb`) that will directly invoke `pg_ctl` to start or restart Advanced Server, then be sure to set `LD_LIBRARY_PATH` before invoking `pg_ctl`.

You can set `LD_LIBRARY_PATH` within the `.bash_profile` file under the home directory of the `enterprisedb` user account (that is, set `LD_LIBRARY_PATH` within file `~enterprisedb/.bash_profile`). This ensures that `LD_LIBRARY_PATH` will be set when you log in as `enterprisedb`.

If you are using a Linux service script with the `systemctl` or `service` command to start or restart Advanced Server, you must set `LD_LIBRARY_PATH` so it is in effect when the script invokes the `pg_ctl` utility.

For example, to set an environment variable for Advanced Server, you can create a file named `/etc/systemd/system/edb-as-13.service`; include `/lib/systemd/system/edb-as-13.service` within the file.

Assuming the `LD_LIBRARY_PATH=/tmp/instantclient` you can now include the environment variable by specifying:

```
[Service]
Environment=LD_LIBRARY_PATH=/tmp/instantclient:$LD_LIBRARY_PATH
Environment=ORACLE_HOME=/tmp/instantclient
```

Then, use the following command to reload `systemd`:

```
systemctl daemon-reload
```

Then, restart the Advanced Server service with the following command:

```
systemctl restart edb-as-13
```

The particular script file that needs to be modified to include the `LD_LIBRARY_PATH` setting depends upon the Advanced Server version, the Linux system on which it was installed, and whether it was installed with the graphical installer or an RPM package.

See the appropriate version of the *EDB Postgres Advanced Server Installation Guide* to determine the service script that affects the startup environment. The installation guides can be found at the following location:

<https://www.enterprisedb.com/edb-docs>

Oracle Instant Client for Windows

The following instructions apply to Windows hosts running Advanced Server.

When you are executing a SQL command that references a database link to a remote Oracle database, Advanced Server must know where the Oracle instant client library resides on the Advanced Server host.

Set the Windows `PATH` system environment variable to include the Oracle client installation directory that contains the `oci.dll` file.

As an alternative you, can set the value of the `oracle_home` configuration parameter in the `postgresql.conf` file. The value specified in the `oracle_home` configuration parameter will override the Windows `PATH` environment variable.

To set the `oracle_home` configuration parameter in the `postgresql.conf` file, edit the file, adding the following line:

```
oracle_home = 'lib_directory'
```

Substitute the name of the Windows directory that contains `oci.dll` for `lib_directory`. For example:

```
oracle_home = 'C:/tmp/instantclient_10_2'
```

After setting the `PATH` environment variable or the `oracle_home` configuration parameter, you must restart the server for the changes to take effect. Restart the server from the Windows Services console.

Note: If `tnsnames.ora` is configured in failover mode, and a client:server failure occurs, the client connection will be established with a secondary server (usually a backup server). Later, when the primary server resumes, the client will retain their connection to a secondary server until a new session is established. The new client connections will automatically be established with the primary server. If the primary and secondary servers are out-of-sync, then there is a possibility that the clients that have established a connection to the secondary server and the clients which later connected to the primary server can see a different database view.

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.

```
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 postgres_fdw Database Link

The following example demonstrates using the CREATE DATABASE LINK command to create a database link (named bedford) that connects an instance of Advanced Server to another Advanced Server instance via a postgres_fdw foreign data wrapper connection. The connection information tells Advanced Server to log in as user admin, whose password is mypassword. Including the postgres_fdw option tells Advanced Server that this is a postgres_fdw connection; the connection string, 'host=127.0.0.1 port=5444 dbname=marketing' specifies the server address and name of the database.

```
CREATE DATABASE LINK bedford
CONNECT TO admin IDENTIFIED BY 'mypassword'
USING postgres_fdw 'host=127.0.0.1 port=5444 dbname=marketing';
```

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 Advanced Server sample application's emp table has been created in an Oracle database and a second 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;
```

empno	ename	job	mgr	hiredate	sal	comm	deptno
7369	SMITH	CLERK	7902	17-DEC-80 00:00:00	800		20
7499	ALLEN	SALESMAN	7698	20-FEB-81 00:00:00	1600	300	30
7521	WARD	SALESMAN	7698	22-FEB-81 00:00:00	1250	500	30
7566	JONES	MANAGER	7839	02-APR-81 00:00:00	2975		20
7654	MARTIN	SALESMAN	7698	28-SEP-81 00:00:00	1250	1400	30
7698	BLAKE	MANAGER	7839	01-MAY-81 00:00:00	2850		30
7782	CLARK	MANAGER	7839	09-JUN-81 00:00:00	2450		10
7788	SCOTT	ANALYST	7566	19-APR-87 00:00:00	3000		20
7839	KING	PRESIDENT		17-NOV-81 00:00:00	5000		10
7844	TURNER	SALESMAN	7698	08-SEP-81 00:00:00	1500	0	30
7876	ADAMS	CLERK	7788	23-MAY-87 00:00:00	1100		20
7900	JAMES	CLERK	7698	03-DEC-81 00:00:00	950		30
7902	FORD	ANALYST	7566	03-DEC-81 00:00:00	3000		20

(continues on next page)

(continued from previous page)

```

7934 | MILLER | CLERK          | 7782 | 23-JAN-82 00:00:00 | 1300 |          | 10
(14 rows)

```

Create a private database link named, `fdwlink`, to the Advanced Server database named, `edb`, located on host `192.168.2.22` running on port `5444`. Connect to the Advanced Server database with username, `enterprisedb`, and password, `password`.

```

CREATE DATABASE LINK fdwlink CONNECT TO enterprisedb IDENTIFIED BY
'password' USING postgres_fdw 'host=192.168.2.22 port=5444 dbname=edb';

```

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

```

SELECT lnkname, lnkuser, lnkconnstr FROM pg_catalog.edb_dblink;

 lnkname |   lnkuser   |          lnkconnstr
-----+-----+-----
 oralink | edb         | //127.0.0.1:1521/xe
 fdwlink | enterprisedb |
(2 rows)

```

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

```

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

 deptno |   dname   | empno | ename  |   job   |  sal  | comm
-----+-----+-----+-----+-----+-----+-----
    10 | ACCOUNTING |   7782 | CLARK  | MANAGER |  2450 |
    10 | ACCOUNTING |   7839 | KING   | PRESIDENT | 5000 |
    10 | ACCOUNTING |   7934 | MILLER | CLERK   |  1300 |
    20 | RESEARCH   |   7369 | SMITH  | CLERK   |   800 |
    20 | RESEARCH   |   7566 | JONES  | MANAGER |  2975 |
    20 | RESEARCH   |   7788 | SCOTT  | ANALYST |  3000 |
    20 | RESEARCH   |   7876 | ADAMS  | CLERK   |  1100 |
    20 | RESEARCH   |   7902 | FORD   | ANALYST |  3000 |
    30 | SALES      |   7499 | ALLEN  | SALESMAN |  1600 |   300
    30 | SALES      |   7521 | WARD   | SALESMAN |  1250 |   500
    30 | SALES      |   7654 | MARTIN | SALESMAN |  1250 |  1400
    30 | SALES      |   7698 | BLAKE  | MANAGER |  2850 |
    30 | SALES      |   7844 | TURNER | SALESMAN |  1500 |    0
    30 | SALES      |   7900 | JAMES  | CLERK   |   950 |
(14 rows)

```

Pushdown for an oci Database Link

When the `oci-dblink` is used to execute SQL statements on a remote Oracle database, there are certain circumstances where pushdown of the processing occurs on the foreign server.

Pushdown refers to the occurrence of processing on the foreign (that is, the remote) server instead of the local client where the SQL statement was issued. Pushdown can result in performance improvement since the data is processed on the remote server before being returned to the local client.

Pushdown applies to statements with the standard SQL join operations (inner join, left outer join, right outer join, and full outer join). Pushdown still occurs even when a sort is specified on the resulting data set.

In order for pushdown to occur, certain basic conditions must be met. The tables involved in the join operation must belong to the same foreign server and use the identical connection information to the foreign server (that is, the same database link defined with the `CREATE DATABASE LINK` command).

In order to determine if pushdown is to be used for a SQL statement, display the execution plan by using the `EXPLAIN` command.

For information about the `EXPLAIN` command, see the PostgreSQL Core documentation at:

<https://www.postgresql.org/docs/current/static/sql-explain.html>

The following examples use the database link created as shown by the following:

```
CREATE PUBLIC DATABASE LINK oralink CONNECT TO edb IDENTIFIED BY 'password'
USING '//192.168.2.23:1521/xe';
```

The following example shows the execution plan of an inner join:

```
EXPLAIN (verbose, costs off) SELECT d.deptno, d.dname, e.empno, e.ename FROM
dept@oralink d, emp@oralink e WHERE d.deptno = e.deptno ORDER BY 1, 3;
```

QUERY PLAN

```
-----
Foreign Scan
  Output: d.deptno, d.dname, e.empno, e.ename
  Relations: (_dblink_dept_1 d) INNER JOIN (_dblink_emp_2 e)
  Remote Query: SELECT r1.deptno, r1.dname, r2.empno, r2.ename FROM (dept
r1 INNER JOIN emp r2 ON ((r1.deptno = r2.deptno))) ORDER BY r1.deptno
ASC NULLS LAST, r2.empno ASC NULLS LAST
(4 rows)
```

Note that the `INNER JOIN` operation occurs under the Foreign Scan section. The output of this join is the following:

deptno	dname	empno	ename
10	ACCOUNTING	7782	CLARK
10	ACCOUNTING	7839	KING
10	ACCOUNTING	7934	MILLER
20	RESEARCH	7369	SMITH
20	RESEARCH	7566	JONES
20	RESEARCH	7788	SCOTT
20	RESEARCH	7876	ADAMS
20	RESEARCH	7902	FORD
30	SALES	7499	ALLEN
30	SALES	7521	WARD
30	SALES	7654	MARTIN
30	SALES	7698	BLAKE
30	SALES	7844	TURNER
30	SALES	7900	JAMES

(14 rows)

The following shows the execution plan of a left outer join:

```
EXPLAIN (verbose, costs off) SELECT d.deptno, d.dname, e.empno, e.ename FROM
dept@oralink d LEFT OUTER JOIN emp@oralink e ON d.deptno = e.deptno ORDER
BY 1, 3;
```

QUERY PLAN

```
-----
Foreign Scan
  Output: d.deptno, d.dname, e.empno, e.ename
```

(continues on next page)

(continued from previous page)

```

Relations: (_dblink_dept_1 d) LEFT JOIN (_dblink_emp_2 e)
Remote Query: SELECT r1.deptno, r1.dname, r2.empno, r2.ename FROM (dept
r1 LEFT JOIN emp r2 ON ((r1.deptno = r2.deptno))) ORDER BY r1.deptno ASC
NULLS LAST, r2.empno ASC NULLS LAST
(4 rows)

```

The output of this join is the following:

deptno	dname	empno	ename
10	ACCOUNTING	7782	CLARK
10	ACCOUNTING	7839	KING
10	ACCOUNTING	7934	MILLER
20	RESEARCH	7369	SMITH
20	RESEARCH	7566	JONES
20	RESEARCH	7788	SCOTT
20	RESEARCH	7876	ADAMS
20	RESEARCH	7902	FORD
30	SALES	7499	ALLEN
30	SALES	7521	WARD
30	SALES	7654	MARTIN
30	SALES	7698	BLAKE
30	SALES	7844	TURNER
30	SALES	7900	JAMES
40	OPERATIONS		

(15 rows)

The following example shows a case where the entire processing is not pushed down because the emp joined table resides locally instead of on the same foreign server.

```

EXPLAIN (verbose, costs off) SELECT d.deptno, d.dname, e.empno, e.ename FROM
dept@oralink d LEFT OUTER JOIN emp e ON d.deptno = e.deptno ORDER BY 1, 3;

```

QUERY PLAN

```

-----
Sort
  Output: d.deptno, d.dname, e.empno, e.ename
  Sort Key: d.deptno, e.empno
  -> Hash Left Join
    Output: d.deptno, d.dname, e.empno, e.ename
    Hash Cond: (d.deptno = e.deptno)
    -> Foreign Scan on _dblink_dept_1 d
      Output: d.deptno, d.dname, d.loc
      Remote Query: SELECT deptno, dname, NULL FROM dept
    -> Hash
      Output: e.empno, e.ename, e.deptno
      -> Seq Scan on public.emp e
        Output: e.empno, e.ename, e.deptno

```

(13 rows)

The output of this join is the same as the previous left outer join example.

Creating a Foreign Table from a Database Link

Note: The procedure described in this section is not compatible with Oracle databases.

After you have created a database link, you can create a foreign table based upon this database link. The foreign table can then be used to access the remote table referencing it with the foreign table name instead of using the database link syntax. Using the database link requires appending `@dblink` to the table or view name referenced in the SQL command where `dblink` is the name of the database link.

This technique can be used for either an `oci-dblink` connection for remote Oracle access, or a `postgres_fdw` connection for remote Postgres access.

The following example shows the creation of a foreign table to access a remote Oracle table.

First, create a database link as previously described. The following is the creation of a database link named `oralink` for connecting to the Oracle database.

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

The following query shows the database link:

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

lnkname	lnkuser	lnkconnstr
oralink	edb	//127.0.0.1:1521/xe

(1 row)

When you create the database link, Advanced Server creates a corresponding foreign server. The following query displays the foreign server:

```
SELECT srvname, srvowner, srvfdw, srvtype, srvoptions FROM
pg_foreign_server;
```

srvname	srvowner	srvfdw	srvtype	srvoptions
oralink	10	14005		{connstr=//127.0.0.1:1521/xe}

(1 row)

For more information about foreign servers, see the `CREATE SERVER` command in the PostgreSQL Core documentation at:

<https://www.postgresql.org/docs/current/static/sql-createserver.html>

Create the foreign table as shown by the following:

```
CREATE FOREIGN TABLE emp_ora (
    empno          NUMERIC(4),
    ename          VARCHAR(10),
    job            VARCHAR(9),
    mgr            NUMERIC(4),
    hiredate       TIMESTAMP WITHOUT TIME ZONE,
    sal            NUMERIC(7,2),
    comm           NUMERIC(7,2),
    deptno        NUMERIC(2)
)
SERVER oralink
OPTIONS (table_name 'emp', schema_name 'edb'
);
```

Note the following in the `CREATE FOREIGN TABLE` command:

- The name specified in the `SERVER` clause at the end of the `CREATE FOREIGN TABLE` command is the name of the foreign server, which is `oralink` in this example as displayed in the `srvname` column from the query on `pg_foreign_server`.
- The table name and schema name are specified in the `OPTIONS` clause by the `table` and `schema` options.
- The column names specified in the `CREATE FOREIGN TABLE` command must match the column names in the remote table.
- Generally, `CONSTRAINT` clauses may not be accepted or enforced on the foreign table as they are assumed to have been defined on the remote table.

For more information about the `CREATE FOREIGN TABLE` command, see the PostgreSQL Core documentation at:

<https://www.postgresql.org/docs/current/static/sql-createforeigntable.html>

The following is a query on the foreign table:

```
SELECT * FROM emp_ora;
```

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

(14 rows)

In contrast, the following is a query on the same remote table, but using the database link instead of the foreign table:

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

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

(continues on next page)

(continued from previous page)

```

---
7369 | SMITH   | CLERK       | 7902 | 17-DEC-80 00:00:00 | 800 |      |      | 20
7499 | ALLEN   | SALESMAN    | 7698 | 20-FEB-81 00:00:00 | 1600 | 300 |      | 30
7521 | WARD    | SALESMAN    | 7698 | 22-FEB-81 00:00:00 | 1250 | 500 |      | 30
7566 | JONES   | MANAGER     | 7839 | 02-APR-81 00:00:00 | 2975 |      |      | 20
7654 | MARTIN  | SALESMAN    | 7698 | 28-SEP-81 00:00:00 | 1250 | 1400 |      | 30
7698 | BLAKE   | MANAGER     | 7839 | 01-MAY-81 00:00:00 | 2850 |      |      | 30
7782 | CLARK   | MANAGER     | 7839 | 09-JUN-81 00:00:00 | 2450 |      |      | 10
7788 | SCOTT   | ANALYST     | 7566 | 19-APR-87 00:00:00 | 3000 |      |      | 20
7839 | KING    | PRESIDENT   |      | 17-NOV-81 00:00:00 | 5000 |      |      | 10
7844 | TURNER  | SALESMAN    | 7698 | 08-SEP-81 00:00:00 | 1500 | 0 |      | 30
7876 | ADAMS   | CLERK       | 7788 | 23-MAY-87 00:00:00 | 1100 |      |      | 20
7900 | JAMES   | CLERK       | 7698 | 03-DEC-81 00:00:00 | 950 |      |      | 30
7902 | FORD    | ANALYST     | 7566 | 03-DEC-81 00:00:00 | 3000 |      |      | 20
7934 | MILLER  | CLERK       | 7782 | 23-JAN-82 00:00:00 | 1300 |      |      | 10
(14 rows)

```

Note: For backward compatibility reasons, it is still possible to write `USING libpq` rather than `USING postgres_fdw`. However, the `libpq` connector is missing many important optimizations which are present in the `postgres_fdw` connector. Therefore, the `postgres_fdw` connector should be used whenever possible. The `libpq` option is deprecated and may be removed entirely in a future Advanced Server release.

See Also

[*DROP DATABASE LINK*](#)

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.

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 a table compatible with Oracle databases.

The directory alias can also be viewed from the Oracle catalog views SYS.ALL_DIRECTORIES and SYS.DBA_DIRECTORIES, which are compatible with Oracle databases.

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;

dirname | diowner |   dirpath   | diracl
-----+-----+-----+-----
empdir  |      10 | C:/TEMP/EMPDIR |
(1 row)
```

View the directory aliases using a view compatible with Oracle databases:

```
SELECT * FROM SYS.ALL_DIRECTORIES;

   owner   | directory_name | directory_path
-----+-----+-----
ENTERPRISEDDB | EMPDIR        | C:/TEMP/EMPDIR
(1 row)
```

See Also

ALTER DIRECTORY, DROP DIRECTORY

CREATE FUNCTION

Name

CREATE FUNCTION -- define a new function

Synopsis

```
CREATE [ OR REPLACE ] FUNCTION <name> [ (<parameters>) ]
  RETURN <data_type>
  [
    IMMUTABLE
  | STABLE
  | VOLATILE
  | DETERMINISTIC
  | [ NOT ] LEAKPROOF
  | CALLED ON NULL INPUT
  | RETURNS NULL ON NULL INPUT
  | STRICT
  | [ EXTERNAL ] SECURITY INVOKER
  | [ EXTERNAL ] SECURITY DEFINER
  | AUTHID DEFINER
  | AUTHID CURRENT_USER
  | PARALLEL { UNSAFE | RESTRICTED | SAFE }
  | COST <execution_cost>
  | ROWS <result_rows>
  | SET configuration_parameter
    { TO <value> | = <value> | FROM CURRENT }
  ...]
{ IS | AS }
  [ PRAGMA AUTONOMOUS_TRANSACTION; ]
  [ <declarations> ]
  BEGIN
    <statements>
  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 input 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 an Advanced Server feature - overloading of stored, standalone functions is not compatible with Oracle databases.)

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. Also when using `OUT` parameters, you cannot change the types of any `OUT` parameters except by dropping the function.

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

Parameters

`name`

`name` is the identifier of the function.

`parameters`

`parameters` is a list of formal parameters.

`data_type`

`data_type` is the data type of the value returned by the function's `RETURN` statement.

`declarations`

`declarations` are variable, cursor, type, or subprogram declarations. If subprogram declarations are included, they must be declared after all other variable, cursor, and type declarations.

`statements`

`statements` are SPL program statements (the `BEGIN` – `END` block may contain an `EXCEPTION` section).

`IMMUTABLE`

`STABLE`

`VOLATILE`

These attributes inform the query optimizer about the behavior of the function; you can specify only one choice. `VOLATILE` is the default behavior.

`IMMUTABLE` indicates that the function cannot modify the database and always reaches the same result when given the same argument values; it does not do database lookups or otherwise use information not directly present in its argument list. If you include this clause, any call of the function with all-constant arguments can be immediately replaced with the function value.

`STABLE` indicates that the function cannot modify the database, and that within a single table scan, it will consistently return the same result for the same argument values, but that its result could change across SQL statements. This is the appropriate selection for function that depend on database lookups, parameter variables (such as the current time zone), etc.

`VOLATILE` indicates that the function value can change even within a single table scan, so no optimizations can be made. Please note that any function that has side-effects must be classified volatile, even if its result is quite predictable, to prevent calls from being optimized away.

`DETERMINISTIC`

`DETERMINISTIC` is a synonym for `IMMUTABLE`. A `DETERMINISTIC` function cannot modify the database and always reaches the same result when given the same argument values; it does not do database lookups or otherwise use information not directly present in its argument list. If you include this clause, any call of the function with all-constant arguments can be immediately replaced with the function value.

[`NOT`] `LEAKPROOF`

A LEAKPROOF function has no side effects, and reveals no information about the values used to call the function.

CALLED ON NULL INPUT

RETURNS NULL ON NULL INPUT

STRICT

CALLED ON NULL INPUT (the default) indicates that the procedure will be called normally when some of its arguments are NULL. It is the author's responsibility to check for NULL values if necessary and respond appropriately.

RETURNS NULL ON NULL INPUT or STRICT indicates that the procedure always returns NULL whenever any of its arguments are NULL. If these clauses are specified, the procedure is not executed when there are NULL arguments; instead a NULL result is assumed automatically.

[EXTERNAL] SECURITY DEFINER

SECURITY DEFINER specifies that the function will execute with the privileges of the user that created it; this is the default. The key word EXTERNAL is allowed for SQL conformance, but is optional.

[EXTERNAL] SECURITY INVOKER

The SECURITY INVOKER clause indicates that the function will execute with the privileges of the user that calls it. The key word EXTERNAL is allowed for SQL conformance, but is optional.

AUTHID DEFINER

AUTHID CURRENT_USER

The AUTHID DEFINER clause is a synonym for [EXTERNAL] SECURITY DEFINER. If the AUTHID clause is omitted or if AUTHID DEFINER is specified, the rights of the function owner are used to determine access privileges to database objects.

The AUTHID CURRENT_USER clause is a synonym for [EXTERNAL] SECURITY INVOKER. If AUTHID CURRENT_USER is specified, the rights of the current user executing the function are used to determine access privileges.

PARALLEL { UNSAFE | RESTRICTED | SAFE }

The PARALLEL clause enables the use of parallel sequential scans (parallel mode). A parallel sequential scan uses multiple workers to scan a relation in parallel during a query in contrast to a serial sequential scan.

When set to UNSAFE, the function cannot be executed in parallel mode. The presence of such a function in a SQL statement forces a serial execution plan. This is the default setting if the PARALLEL clause is omitted.

When set to RESTRICTED, the function can be executed in parallel mode, but the execution is restricted to the parallel group leader. If the qualification for any particular relation has anything that is parallel restricted, that relation won't be chosen for parallelism.

When set to SAFE, the function can be executed in parallel mode with no restriction.

COST execution_cost

execution_cost is a positive number giving the estimated execution cost for the function, in units of cpu_operator_cost. If the function returns a set, this is the cost per returned row. Larger values cause the planner to try to avoid evaluating the function more often than necessary.

ROWS result_rows

`result_rows` is a positive number giving the estimated number of rows that the planner should expect the function to return. This is only allowed when the function is declared to return a set. The default assumption is 1000 rows.

```
SET configuration_parameter { TO value | = value | FROM CURRENT }
```

The `SET` clause causes the specified configuration parameter to be set to the specified value when the function is entered, and then restored to its prior value when the function exits. `SET FROM CURRENT` saves the session's current value of the parameter as the value to be applied when the function is entered.

If a `SET` clause is attached to a function, then the effects of a `SET LOCAL` command executed inside the function for the same variable are restricted to the function; the configuration parameter's prior value is restored at function exit. An ordinary `SET` command (without `LOCAL`) overrides the `SET` clause, much as it would do for a previous `SET LOCAL` command, with the effects of such a command persisting after procedure exit, unless the current transaction is rolled back.

```
PRAGMA AUTONOMOUS_TRANSACTION
```

`PRAGMA AUTONOMOUS_TRANSACTION` is the directive that sets the function as an autonomous transaction.

Note: The `STRICT`, `LEAKPROOF`, `PARALLEL`, `COST`, `ROWS` and `SET` keywords provide extended functionality for Advanced Server and are not supported by Oracle.

Notes

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;
```

Name	Salary	Commission	Total Compensation
SMITH	800.00		19200.00
ALLEN	1600.00	300.00	45600.00
WARD	1250.00	500.00	42000.00
JONES	2975.00		71400.00
MARTIN	1250.00	1400.00	63600.00
BLAKE	2850.00		68400.00
CLARK	2450.00		58800.00
SCOTT	3000.00		72000.00
KING	5000.00		120000.00
TURNER	1500.00	0.00	36000.00

(continues on next page)

(continued from previous page)

ADAMS		1100.00				26400.00
JAMES		950.00				22800.00
FORD		3000.00				72000.00
MILLER		1300.00				31200.00

(14 rows)

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.

```
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

The following example demonstrates using the `AUTHID CURRENT_USER` clause and `STRICT` keyword in a function declaration:

```
CREATE OR REPLACE FUNCTION dept_salaries(dept_id int) RETURN NUMBER
    STRICT
    AUTHID CURRENT_USER
BEGIN
    RETURN QUERY (SELECT sum(salary) FROM emp WHERE deptno = id);
END;
```

Include the `STRICT` keyword to instruct the server to return `NULL` if any input parameter passed is `NULL`; if a `NULL` value is passed, the function will not execute.

The `dept_salaries` function executes with the privileges of the role that is calling the function. If the current user does not have sufficient privileges to perform the `SELECT` statement querying the `emp` table (to display employee salaries), the function will report an error. To instruct the server to use the privileges associated with the role that defined the function, replace the `AUTHID CURRENT_USER` clause with the `AUTHID DEFINER` clause.

Other Pragmas (declared within a package specification)

```
PRAGMA RESTRICT_REFERENCES
```

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

See Also

DROP FUNCTION

CREATE INDEX

Name

`CREATE INDEX` -- define a new index

Synopsis

```
CREATE [ UNIQUE ] INDEX <name> ON <table>
( { <column> | ( <expression> ) | <constant> } )
[ TABLESPACE <tablespace> ]
( { NOPARALLEL | PARALLEL [ <integer> ] } )
```

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).

The key field(s) for the index are specified as column names, constants, or 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 `UPPER(col)` would allow the clause `WHERE UPPER(col) = 'JIM'` to use an index.

Advanced Server provides the B-tree index method. The B-tree index method is an implementation of Lehman-Yao high-concurrency B-trees.

Indexes are not used for `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.

If you create an index on a partitioned table, the `CREATE INDEX` command does propagate indexes to the table’s subpartitions.

The `PARALLEL` clause specifies the degree of parallelism used during the creation of an index. The `NOPARALLEL` clause resets the parallelism to the default value; `reloptions` will show the `parallel_workers` parameter as 0.

Note: If you use the `CREATE INDEX . . . PARALLEL` command to create an index on a table whose definition included the `PARALLEL` clause (at creation), the server will use the `PARALLEL` clause provided with the `CREATE INDEX` command when building a parallel index.

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.

constant

A constant value that can be used as an index key.

Normally, a row where all indexed columns are NULL is not included in an index. That means that the optimizer cannot use that index for certain queries. To overcome this limitation, you can add a constant to the index, thereby forcing the index to never contain a row where all index columns are NULL.

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.

PARALLEL

Specify **PARALLEL** to select a degree of parallelism; set `parallel_workers` parameter equal to the degree of parallelism to create a parallelized index. Alternatively, if you specify **PARALLEL** but no degree of parallelism is listed, an index accepts default parallelism.

NOPARALLEL

Specify **NOPARALLEL** for default execution.

integer

The `integer` indicates the degree of parallelism, which is a number of `parallel_workers` used in the parallel operation to perform a parallel scan on an index.

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`:

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

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

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

You can include a constant value in the index definition (1) to create an index that never contains a row where all of the indexed columns are NULL:

```
CREATE INDEX emp_dob_idx on emp (emp_dob, 1);
```

To create an index on name_idx in the table emp with degree of parallelism set to 7:

```
CREATE UNIQUE INDEX name_idx ON emp (ename) PARALLEL 7;
```

See Also

ALTER INDEX, DROP INDEX

CREATE MATERIALIZED VIEW

Name

CREATE MATERIALIZED VIEW -- define a new materialized view

Synopsis

```
CREATE MATERIALIZED VIEW <name>
    [<build_clause>][<create_mv_refresh>] AS subquery

    Where <build_clause> is:

    BUILD {IMMEDIATE | DEFERRED}

    Where <create_mv_refresh> is:

    REFRESH [COMPLETE] [ON DEMAND]``
```

Description

CREATE MATERIALIZED VIEW defines a view of a query that is not updated each time the view is referenced in a query. By default, the view is populated when the view is created; you can include the BUILD DEFERRED keywords to delay the population of the view.

A materialized view may be schema-qualified; if you specify a schema name when invoking the CREATE MATERIALIZED VIEW command, the view will be created in the specified 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.

subquery

A SELECT statement that specifies the contents of the view. Refer to SELECT for more information about valid queries.

build_clause

Include a build_clause to specify when the view should be populated. Specify BUILD IMMEDIATE, or BUILD DEFERRED:

- BUILD IMMEDIATE instructs the server to populate the view immediately. This is the default behavior.
- BUILD DEFERRED instructs the server to populate the view at a later time (during a REFRESH operation).

`create_mv_refresh`

Include the `create_mv_refresh` clause to specify when the contents of a materialized view should be updated. The clause contains the `REFRESH` keyword followed by `COMPLETE` and/or `ON DEMAND`, where:

- `COMPLETE` instructs the server to discard the current content and reload the materialized view by executing the view's defining query when the materialized view is refreshed.
- `ON DEMAND` instructs the server to refresh the materialized view on demand by calling the `DBMS_MVIEW` package or by calling the Postgres `REFRESH MATERIALIZED VIEW` statement. This is the default behavior.

Notes

Materialized views are read only - the server will not allow an `INSERT`, `UPDATE`, or `DELETE` on a view.

Access to tables referenced in the view is determined by permissions of the view owner; the user of a view must have permissions to call all functions used by the view.

For more information about the Postgres `REFRESH MATERIALIZED VIEW` command, see the PostgreSQL Core Documentation available at:

<https://www.postgresql.org/docs/current/static/sql-refreshmaterializedview.html>

Examples

The following statement creates a materialized view named `dept_30`:

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

The view contains information retrieved from the `emp` table about any employee that works in department 30.

CREATE PACKAGE

Name

CREATE PACKAGE -- define a new package specification

Synopsis

```
CREATE [ OR REPLACE ] PACKAGE <name>
[ AUTHID { DEFINER | CURRENT_USER } ]
{ IS | AS }
[ <declaration>; ] [, ...]
[ { PROCEDURE <proc_name>
    [ (<argname> [ IN | IN OUT | OUT ] <argtype> [ DEFAULT value ]
      [, ...]) ];
  [ PRAGMA RESTRICT_REFERENCES(<name>,
    { RNDS | RNPS | TRUST | WNDS | WNPS } [, ... ] ); ]
  |
  FUNCTION <func_name>
    [ (<argname> [ IN | IN OUT | OUT ] <argtype> [ DEFAULT value ]
      [, ...]) ]
    RETURN <rettype> [ DETERMINISTIC ];
  [ PRAGMA RESTRICT_REFERENCES(<name>,
    { RNDS | RNPS | TRUST | WNDS | WNPS } [, ... ] ); ]
  }
] [, ...]
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.

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.

DETERMINISTIC

DETERMINISTIC is a synonym for IMMUTABLE. A DETERMINISTIC procedure cannot modify the database and always reaches the same result when given the same argument values; it does not do database lookups or otherwise use information not directly present in its argument list. If you include this clause, any call of the procedure with all-constant arguments can be immediately replaced with the procedure value.

RNDS | RNPS | TRUST | WNDS | WNPS

The keywords are accepted for compatibility and ignored.

Examples

The package specification, `empinfo`, contains three public components - a public variable, a public procedure, and a public function.

```
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 PACKAGE BODY -- define a new package body

Synopsis

```
CREATE [ OR REPLACE ] PACKAGE BODY <name>
{ IS | AS }
[ declaration; ] | [ forward_declaration ] [, ...]
[ { PROCEDURE <proc_name>
    [ (<argname> [ IN | IN OUT | OUT ] <argtype> [ DEFAULT <value> ] [,
    ...]) ]
    [ STRICT ]
    [ LEAKPROOF ]
    [ PARALLEL { UNSAFE | RESTRICTED | SAFE } ]
    [ COST <execution_cost> ]
    [ ROWS <result_rows> ]
    [ SET <config_param> { TO <value> | = <value> | FROM CURRENT } ]
  { IS | AS }
    <program_body>
  END [ <proc_name> ];
|
  FUNCTION <func_name>
  [ (<argname> [ IN | IN OUT | OUT ] <argtype> [ DEFAULT <value> ] [,
  ...]) ]
  RETURN <rettype> [ DETERMINISTIC ]
  [ STRICT ]
  [ LEAKPROOF ]
  [ PARALLEL { UNSAFE | RESTRICTED | SAFE } ]
  [ COST <execution_cost> ]
  [ ROWS <result_rows> ]
  [ SET <config_param> { TO <value> | = <value> | FROM CURRENT } ]
  { IS | AS }
    <program_body>
  END [ <func_name> ];
}
[ [, ...]
[ BEGIN
    <statement>; [, ...] ]
END [ <name> ]
```

Where forward_declaration:=

```
[ { PROCEDURE <proc_name>
    [ (<argname> [ IN | IN OUT | OUT ] <argtype> [ DEFAULT <value> ] [, ...])
```

(continues on next page)

(continued from previous page)

```

] ; ]
|
[ { FUNCTION <func_name>
  [ (<argname> [ IN | IN OUT | OUT ] <argtype> [ DEFAULT <value> ] [, ...])
  ]
  RETURN <rettype> [ DETERMINISTIC ]; ]

```

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.

Parameters

name

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

declaration

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

forward_declaration

The forward declaration of a procedure or function appears within a package body and is declared in advance of the actual body definition. In a block, you can create multiple subprograms; if they invoke each other, each one requires a forward declaration. A subprogram must be declared before it can be invoked. You can use a forward declaration to declare a subprogram without defining it. The forward declaration and its corresponding definition must reside in the same block.

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.

STRICT

The STRICT keyword specifies that the function will not be executed if called with a NULL argument; instead the function will return NULL.

LEAKPROOF

The LEAKPROOF keyword specifies that the function will not reveal any information about arguments, other than through a return value.

PARALLEL { UNSAFE | RESTRICTED | SAFE }

The `PARALLEL` clause enables the use of parallel sequential scans (parallel mode). A parallel sequential scan uses multiple workers to scan a relation in parallel during a query in contrast to a serial sequential scan.

When set to `UNSAFE`, the procedure or function cannot be executed in parallel mode. The presence of such a procedure or function forces a serial execution plan. This is the default setting if the `PARALLEL` clause is omitted.

When set to `RESTRICTED`, the procedure or function can be executed in parallel mode, but the execution is restricted to the parallel group leader. If the qualification for any particular relation has anything that is parallel restricted, that relation won't be chosen for parallelism.

When set to `SAFE`, the procedure or function can be executed in parallel mode with no restriction.

`execution_cost`

`execution_cost` specifies a positive number giving the estimated execution cost for the function, in units of `cpu_operator_cost`. If the function returns a set, this is the cost per returned row. The default is 0.0025.

`result_rows`

`result_rows` is the estimated number of rows that the query planner should expect the function to return. The default is 1000.

`SET`

Use the `SET` clause to specify a parameter value for the duration of the function:

`config_param` specifies the parameter name.

`value` specifies the parameter value.

`FROM CURRENT` guarantees that the parameter value is restored when the function ends.

`program_body`

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

The pragma may be `PRAGMA AUTONOMOUS_TRANSACTION` to set the function or procedure as an autonomous transaction.

The declarations may include variable, type, `REF CURSOR`, or subprogram declarations. If subprogram declarations are included, they must be declared after all other variable, type, and `REF CURSOR` declarations.

`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.

`DETERMINISTIC`

Include `DETERMINISTIC` to specify that the function will always return the same result when given the same argument values. A `DETERMINISTIC` function must not modify the database.

Note: The `DETERMINISTIC` keyword is equivalent to the PostgreSQL `IMMUTABLE` option. If `DETERMINISTIC` is specified for a public function in the package body, it must also be specified for

the function declaration in the package specification. For private functions, there is no function declaration in the package specification.

statement

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

Note: The `STRICT`, `LEAKPROOF`, `PARALLEL`, `COST`, `ROWS` and `SET` keywords provide extended functionality for Advanced Server and are not supported by Oracle.

Examples

The following is the package body for the `empinfo` package.

```
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;
```

The following two anonymous blocks execute the procedure and function in the `empinfo` package and display the public variable.

```
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;

Initialized counter
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;
```

(continues on next page)

(continued from previous page)

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

The following example demonstrates the use of a forward declaration within a package body. The example displays the name and number of employees whose salary falls in the specified range.

```
CREATE OR REPLACE PACKAGE empinfo IS
    FUNCTION emp_comp (p_sal_range INTEGER) RETURN INTEGER;
END empinfo;

CREATE OR REPLACE PACKAGE BODY empinfo IS
    FUNCTION sal_range (p_sal_range INTEGER) RETURN INTEGER;
    PROCEDURE list_emp (p_sal_range INTEGER);

    FUNCTION emp_comp (p_sal_range INTEGER) RETURN INTEGER IS
    BEGIN
        dbms_output.put_line ('Employee details');
        return sal_range(p_sal_range);
    END;

    FUNCTION sal_range (p_sal_range INTEGER) RETURN INTEGER IS
        emp_cnt INTEGER;
    BEGIN
        select count(*) into emp_cnt from emp where sal <= p_sal_range;
        dbms_output.put_line('Number of employees in the salary range ' ||
p_sal_range|| ' is :'|| emp_cnt);
        list_emp(p_sal_range);
        return emp_cnt;
    END;

    PROCEDURE list_emp (p_sal_range IN INTEGER) IS
    BEGIN
        FOR i IN select ename from emp where sal <= p_sal_range
        LOOP
            dbms_output.put_line (i);
        END LOOP;
    END;
END empinfo;

SELECT empinfo.emp_comp(1500);
Employee details
Number of employees in the salary range 1500 is :7
(SMITH)
(WARD)
(MARTIN)
(TURNER)
(ADAMS)
(JAMES)
(MILLER)
emp_comp
-----
          7
(1 row)
```

See Also

CREATE PACKAGE, DROP PACKAGE

CREATE PROCEDURE

Name

CREATE PROCEDURE -- define a new stored procedure

Synopsis

```
CREATE [OR REPLACE] PROCEDURE <name> [ (<parameters>) ]
[
    IMMUTABLE
  | STABLE
  | VOLATILE
  | DETERMINISTIC
  | [ NOT ] LEAKPROOF
  | CALLED ON NULL INPUT
  | RETURNS NULL ON NULL INPUT
  | STRICT
  | [ EXTERNAL ] SECURITY INVOKER
  | [ EXTERNAL ] SECURITY DEFINER
  | AUTHID DEFINER
  | AUTHID CURRENT_USER
  | PARALLEL { UNSAFE | RESTRICTED | SAFE }
  | COST <execution_cost>
  | ROWS <result_rows>
  | SET <configuration_parameter>
    { TO <value> | = <value> | FROM CURRENT }
... ]
{ IS | AS }
[ PRAGMA AUTONOMOUS_TRANSACTION; ]
[ <declarations> ]
BEGIN
    <statements>
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 with the same input argument types in the same schema. However, procedures of different input argument types may share a name (this is called overloading). (Overloading of procedures is an Advanced Server feature - overloading of stored, standalone procedures is not compatible with Oracle databases.)

To update the definition of an existing procedure, use CREATE OR REPLACE PROCEDURE. It is not possible to change the name or argument types of a procedure this way (if you tried, you would actually be creating a new,

distinct procedure). When using OUT parameters, you cannot change the types of any OUT parameters except by dropping the procedure.

Parameters

name

name is the identifier of the procedure.

parameters

parameters is a list of formal parameters.

declarations

declarations are variable, cursor, type, or subprogram declarations. If subprogram declarations are included, they must be declared after all other variable, cursor, and type declarations.

statements

statements are SPL program statements (the BEGIN – END block may contain an EXCEPTION section).

IMMUTABLE

STABLE

VOLATILE

These attributes inform the query optimizer about the behavior of the procedure; you can specify only one choice. VOLATILE is the default behavior.

IMMUTABLE indicates that the procedure cannot modify the database and always reaches the same result when given the same argument values; it does not do database lookups or otherwise use information not directly present in its argument list. If you include this clause, any call of the procedure with all-constant arguments can be immediately replaced with the procedure value.

STABLE indicates that the procedure cannot modify the database, and that within a single table scan, it will consistently return the same result for the same argument values, but that its result could change across SQL statements. This is the appropriate selection for procedures that depend on database lookups, parameter variables (such as the current time zone), etc.

VOLATILE indicates that the procedure value can change even within a single table scan, so no optimizations can be made. Please note that any function that has side-effects must be classified volatile, even if its result is quite predictable, to prevent calls from being optimized away.

DETERMINISTIC

DETERMINISTIC is a synonym for IMMUTABLE. A DETERMINISTIC procedure cannot modify the database and always reaches the same result when given the same argument values; it does not do database lookups or otherwise use information not directly present in its argument list. If you include this clause, any call of the procedure with all-constant arguments can be immediately replaced with the procedure value.

[NOT] LEAKPROOF

A LEAKPROOF procedure has no side effects, and reveals no information about the values used to call the procedure.

CALLED ON NULL INPUT

RETURNS NULL ON NULL INPUT

STRICT

`CALLED ON NULL INPUT` (the default) indicates that the procedure will be called normally when some of its arguments are `NULL`. It is the author's responsibility to check for `NULL` values if necessary and respond appropriately.

`RETURNS NULL ON NULL INPUT` or `STRICT` indicates that the procedure always returns `NULL` whenever any of its arguments are `NULL`. If these clauses are specified, the procedure is not executed when there are `NULL` arguments; instead a `NULL` result is assumed automatically.

[`EXTERNAL`] `SECURITY DEFINER`

`SECURITY DEFINER` specifies that the procedure will execute with the privileges of the user that created it; this is the default. The key word `EXTERNAL` is allowed for SQL conformance, but is optional.

[`EXTERNAL`] `SECURITY INVOKER`

The `SECURITY INVOKER` clause indicates that the procedure will execute with the privileges of the user that calls it. The key word `EXTERNAL` is allowed for SQL conformance, but is optional.

`AUTHID DEFINER`

`AUTHID CURRENT_USER`

The `AUTHID DEFINER` clause is a synonym for [`EXTERNAL`] `SECURITY DEFINER`. If the `AUTHID` clause is omitted or if `AUTHID DEFINER` is specified, the rights of the procedure owner are used to determine access privileges to database objects.

The `AUTHID CURRENT_USER` clause is a synonym for [`EXTERNAL`] `SECURITY INVOKER`. If `AUTHID CURRENT_USER` is specified, the rights of the current user executing the procedure are used to determine access privileges.

`PARALLEL { UNSAFE | RESTRICTED | SAFE }`

The `PARALLEL` clause enables the use of parallel sequential scans (parallel mode). A parallel sequential scan uses multiple workers to scan a relation in parallel during a query in contrast to a serial sequential scan.

When set to `UNSAFE`, the procedure cannot be executed in parallel mode. The presence of such a procedure forces a serial execution plan. This is the default setting if the `PARALLEL` clause is omitted.

When set to `RESTRICTED`, the procedure can be executed in parallel mode, but the execution is restricted to the parallel group leader. If the qualification for any particular relation has anything that is parallel restricted, that relation won't be chosen for parallelism.

When set to `SAFE`, the procedure can be executed in parallel mode with no restriction.

`COST execution_cost`

`execution_cost` is a positive number giving the estimated execution cost for the procedure, in units of `cpu_operator_cost`. If the procedure returns a set, this is the cost per returned row. Larger values cause the planner to try to avoid evaluating the function more often than necessary.

`ROWS result_rows`

`result_rows` is a positive number giving the estimated number of rows that the planner should expect the procedure to return. This is only allowed when the procedure is declared to return a set. The default assumption is 1000 rows.

`SET configuration_parameter { TO value | = value | FROM CURRENT }`

The `SET` clause causes the specified configuration parameter to be set to the specified value when the procedure is entered, and then restored to its prior value when the procedure exits. `SET FROM CURRENT` saves the session's current value of the parameter as the value to be applied when the procedure is entered.

If a `SET` clause is attached to a procedure, then the effects of a `SET LOCAL` command executed inside the procedure for the same variable are restricted to the procedure; the configuration parameter's prior value is restored at procedure exit. An ordinary `SET` command (without `LOCAL`) overrides the `SET` clause, much as it would do for a previous `SET LOCAL` command, with the effects of such a command persisting after procedure exit, unless the current transaction is rolled back.

`PRAGMA AUTONOMOUS_TRANSACTION`

`PRAGMA AUTONOMOUS_TRANSACTION` is the directive that sets the procedure as an autonomous transaction.

Note:

- The `STRICT`, `LEAKPROOF`, `PARALLEL`, `COST`, `ROWS` and `SET` keywords provide extended functionality for Advanced Server and are not supported by Oracle.
- The `IMMUTABLE`, `STABLE`, `STRICT`, `LEAKPROOF`, `COST`, `ROWS` and `PARALLEL { UNSAFE | RESTRICTED | SAFE }` attributes are only supported for EDB SPL procedures.
- By default, stored procedures are created as `SECURITY DEFINERS`; stored procedures defined in `plpgsql` are created as `SECURITY INVOKERS`.

Examples

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

```
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

(continues on next page)

(continued from previous page)

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.

```
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;

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
```

(continues on next page)

(continued from previous page)

Name	: CLARK
Job	: MANAGER

The following example demonstrates using the `AUTHID DEFINER` and `SET` clauses in a procedure declaration. The `update_salary` procedure conveys the privileges of the role that defined the procedure to the role that is calling the procedure (while the procedure executes):

```
CREATE OR REPLACE PROCEDURE update_salary(id INT, new_salary NUMBER)
  SET SEARCH_PATH = 'public' SET WORK_MEM = '1MB'
  AUTHID DEFINER IS
BEGIN
  UPDATE emp SET salary = new_salary WHERE emp_id = id;
END;
```

Include the `SET` clause to set the procedure's search path to `public` and the work memory to 1MB. Other procedures, functions and objects will not be affected by these settings.

In this example, the `AUTHID DEFINER` clause temporarily grants privileges to a role that might otherwise not be allowed to execute the statements within the procedure. To instruct the server to use the privileges associated with the role invoking the procedure, replace the `AUTHID DEFINER` clause with the `AUTHID CURRENT_USER` clause.

See Also

DROP PROCEDURE, ALTER PROCEDURE

CREATE PROFILE

Name

CREATE PROFILE – create a new profile

Synopsis

```
CREATE PROFILE <profile_name>
    [LIMIT {<parameter value>} ... ];
```

Description

CREATE PROFILE creates a new profile. Include the LIMIT clause and one or more space-delimited parameter/value pairs to specify the rules enforced by Advanced Server.

Advanced Server creates a default profile named DEFAULT. When you use the CREATE ROLE command to create a new role, the new role is automatically associated with the DEFAULT profile. If you upgrade from a previous version of Advanced Server to Advanced Server 10, the upgrade process will automatically create the roles in the upgraded version to the DEFAULT profile.

You must be a superuser to use CREATE PROFILE.

Include the LIMIT clause and one or more space-delimited parameter/value pairs to specify the rules enforced by Advanced Server.

Parameters

profile_name

The name of the profile.

parameter

The password attribute that will be monitored by the rule.

value

The value the parameter must reach before an action is taken by the server.

Advanced Server supports the value shown below for each parameter:

FAILED_LOGIN_ATTEMPTS specifies the number of failed login attempts that a user may make before the server locks the user out of their account for the length of time specified by PASSWORD_LOCK_TIME. Supported values are:

- An INTEGER value greater than 0.
- DEFAULT - the value of FAILED_LOGIN_ATTEMPTS specified in the DEFAULT profile.
- UNLIMITED – the connecting user may make an unlimited number of failed login attempts.

`PASSWORD_LOCK_TIME` specifies the length of time that must pass before the server unlocks an account that has been locked because of `FAILED_LOGIN_ATTEMPTS`. Supported values are:

- A `NUMERIC` value greater than or equal to 0. To specify a fractional portion of a day, specify a decimal value. For example, use the value 4.5 to specify 4 days, 12 hours.
- `DEFAULT` - the value of `PASSWORD_LOCK_TIME` specified in the `DEFAULT` profile.
- `UNLIMITED` – the account is locked until it is manually unlocked by a database superuser.

`PASSWORD_LIFE_TIME` specifies the number of days that the current password may be used before the user is prompted to provide a new password. Include the `PASSWORD_GRACE_TIME` clause when using the `PASSWORD_LIFE_TIME` clause to specify the number of days that will pass after the password expires before connections by the role are rejected. If `PASSWORD_GRACE_TIME` is not specified, the password will expire on the day specified by the default value of `PASSWORD_GRACE_TIME`, and the user will not be allowed to execute any command until a new password is provided. Supported values are:

- A `NUMERIC` value greater than or equal to 0. To specify a fractional portion of a day, specify a decimal value. For example, use the value 4.5 to specify 4 days, 12 hours.
- `DEFAULT` - the value of `PASSWORD_LIFE_TIME` specified in the `DEFAULT` profile.
- `UNLIMITED` – The password does not have an expiration date.

`PASSWORD_GRACE_TIME` specifies the length of the grace period after a password expires until the user is forced to change their password. When the grace period expires, a user will be allowed to connect, but will not be allowed to execute any command until they update their expired password. Supported values are:

- A `NUMERIC` value greater than or equal to 0. To specify a fractional portion of a day, specify a decimal value. For example, use the value 4.5 to specify 4 days, 12 hours.
- `DEFAULT` - the value of `PASSWORD_GRACE_TIME` specified in the `DEFAULT` profile.
- `UNLIMITED` – The grace period is infinite.

`PASSWORD_REUSE_TIME` specifies the number of days a user must wait before re-using a password. The `PASSWORD_REUSE_TIME` and `PASSWORD_REUSE_MAX` parameters are intended to be used together. If you specify a finite value for one of these parameters while the other is `UNLIMITED`, old passwords can never be reused. If both parameters are set to `UNLIMITED` there are no restrictions on password reuse. Supported values are:

- A `NUMERIC` value greater than or equal to 0. To specify a fractional portion of a day, specify a decimal value. For example, use the value 4.5 to specify 4 days, 12 hours.
- `DEFAULT` - the value of `PASSWORD_REUSE_TIME` specified in the `DEFAULT` profile.
- `UNLIMITED` – The password can be re-used without restrictions.

`PASSWORD_REUSE_MAX` specifies the number of password changes that must occur before a password can be reused. The `PASSWORD_REUSE_TIME` and `PASSWORD_REUSE_MAX` parameters are intended to be used together. If you specify a finite value for one of these parameters while the other is `UNLIMITED`, old passwords can never be reused. If both parameters are set to `UNLIMITED` there are no restrictions on password reuse. Supported values are:

- An `INTEGER` value greater than or equal to 0.
- `DEFAULT` - the value of `PASSWORD_REUSE_MAX` specified in the `DEFAULT` profile.
- `UNLIMITED` – The password can be re-used without restrictions.

`PASSWORD_VERIFY_FUNCTION` specifies password complexity. Supported values are:

- The name of a PL/SQL function.

- **DEFAULT** - the value of `PASSWORD_VERIFY_FUNCTION` specified in the **DEFAULT** profile.
- **NULL**

`PASSWORD_ALLOW_HASHED` specifies whether an encrypted password to be allowed for use or not. If you specify the value as **TRUE**, the system allows a user to change the password by specifying a hash computed encrypted password on the client side. However, if you specify the value as **FALSE**, then a password must be specified in a plain-text form in order to be validated effectively, else an error will be thrown if a server receives an encrypted password. Supported values are:

- A **BOOLEAN** value **TRUE/ON/YES/1** or **FALSE/OFF/NO/0**.
- **DEFAULT** – the value of `PASSWORD_ALLOW_HASHED` specified in the **DEFAULT** profile.

Note: The `PASSWORD_ALLOW_HASHED` is not an Oracle-compatible parameter.

Notes

Use `DROP PROFILE` command to remove the profile.

Examples

The following command creates a profile named `acctg`. The profile specifies that if a user has not authenticated with the correct password in five attempts, the account will be locked for one day:

```
CREATE PROFILE acctg LIMIT
  FAILED_LOGIN_ATTEMPTS 5
  PASSWORD_LOCK_TIME 1;
```

The following command creates a profile named `sales`. The profile specifies that a user must change their password every 90 days:

```
CREATE PROFILE sales LIMIT
  PASSWORD_LIFE_TIME 90
  PASSWORD_GRACE_TIME 3;
```

If the user has not changed their password before the 90 days specified in the profile has passed, they will be issued a warning at login. After a grace period of 3 days, their account will not be allowed to invoke any commands until they change their password.

The following command creates a profile named `accts`. The profile specifies that a user cannot re-use a password within 180 days of the last use of the password, and must change their password at least 5 times before re-using the password:

```
CREATE PROFILE accts LIMIT
  PASSWORD_REUSE_TIME 180
  PASSWORD_REUSE_MAX 5;
```

The following command creates a profile named `resources`; the profile calls a user-defined function named `password_rules` that will verify that the password provided meets their standards for complexity:

```
CREATE PROFILE resources LIMIT
  PASSWORD_VERIFY_FUNCTION password_rules;
```

See Also

[ALTER PROFILE](#), [DROP PROFILE](#)

CREATE QUEUE

Advanced Server includes extra syntax (not offered by Oracle) with the `CREATE QUEUE SQL` command. This syntax can be used in association with `DBMS_AQADM`.

Name

`CREATE QUEUE` – create a queue.

Synopsis

Use `CREATE QUEUE` to define a new queue:

```
CREATE QUEUE <name> QUEUE TABLE <queue_table_name> [ ( { <option_name  
option_value>} [, ... ] ) ]
```

where `option_name` and the corresponding `option_value` can be:

```
TYPE [normal_queue | exception_queue]  
RETRIES [INTEGER]  
RETRYDELAY [DOUBLE PRECISION]  
RETENTION [DOUBLE PRECISION]
```

Description

The `CREATE QUEUE` command allows a database superuser or any user with the system-defined `aq_administrator_role` privilege to create a new queue in the current database.

If the name of the queue is schema-qualified, the queue is created in the specified schema. If a schema is not included in the `CREATE QUEUE` command, the queue is created in the current schema. A queue may only be created in the schema in which the queue table resides. The name of the queue must be unique from the name of any other queue in the same schema.

Use `DROP QUEUE` to remove a queue.

Parameters

`name`

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

`queue_table_name`

The name of the queue table with which this queue is associated.

`option_name option_value`

The name of any options that will be associated with the new queue, and the corresponding value for the option. If the call to `CREATE QUEUE` includes duplicate option names, the server will return an error. The following values are supported:

TYPE	Specify <code>normal_queue</code> to indicate that the queue is a normal queue, or <code>exception_queue</code> to indicate that the queue is an exception queue. An exception queue will only accept dequeue operations.
RETRIES	An INTEGER value that specifies the maximum number of attempts to remove a message from a queue.
RETRYDELAY	A DOUBLE PRECISION value that specifies the number of seconds after a ROLLBACK that the server will wait before retrying a message.
RETENTION	A DOUBLE PRECISION value that specifies the number of seconds that a message will be saved in the queue table after dequeuing.

Examples

The following command creates a queue named `work_order` that is associated with a queue table named `work_order_table`:

```
CREATE QUEUE work_order QUEUE TABLE work_order_table (RETRIES 5, RETRYDELAY 2);
```

The server will allow 5 attempts to remove a message from the queue, and enforce a retry delay of 2 seconds between attempts.

See Also

ALTER QUEUE, DROP QUEUE

CREATE QUEUE TABLE

Advanced Server includes extra syntax (not offered by Oracle) with the `CREATE QUEUE TABLE` SQL command. This syntax can be used in association with `DBMS_AQADM`.

Name

`CREATE QUEUE TABLE`-- create a new queue table.

Synopsis

Use `CREATE QUEUE TABLE` to define a new queue table:

```
CREATE QUEUE TABLE <name> OF <type_name> [ ( { <option_name option_value> }  
[, ... ] ) ]
```

where `option_name` and the corresponding `option_value` can be:

option_name	option_value
<code>SORT_LIST</code>	<code>priority</code> , <code>enq_time</code>
<code>MULTIPLE_CONSUMERS</code>	<code>FALSE</code> , <code>TRUE</code>
<code>MESSAGE_GROUPING</code>	<code>NONE</code> , <code>TRANSACTIONAL</code>
<code>STORAGE_CLAUSE</code>	<code>TABLESPACE</code> <code>tablespace_name</code> , <code>PCTFREE</code> integer, <code>PCTUSED</code> integer, <code>INITTRANS</code> integer, <code>MAXTRANS</code> integer, <code>STORAGE</code> <code>storage_option</code> Where <code>storage_option</code> is one or more of the following: <code>MINEXTENTS</code> integer, <code>MAXEXTENTS</code> integer, <code>PCTINCREASE</code> integer, <code>INITIAL</code> <code>size_clause</code> , <code>NEXT</code> , <code>FREELISTS</code> integer, <code>OPTIMAL</code> <code>size_clause</code> , <code>BUFFER_POOL</code> { <code>KEEP</code> <code>RECYCLE</code> <code>DEFAULT</code> }. Please note that only the <code>TABLESPACE</code> option is enforced; all others are accepted for compatibility and ignored. Use the <code>TABLESPACE</code> clause to specify the name of a tablespace in which the table will be created.

Description

`CREATE QUEUE TABLE` allows a superuser or a user with the `aq_administrator_role` privilege to create a new queue table.

If the call to `CREATE QUEUE TABLE` includes a schema name, the queue table is created in the specified schema. If no schema name is provided, the new queue table is created in the current schema.

The name of the queue table must be unique from the name of any other queue table in the same schema.

Parameters

`name`

The name (optionally schema-qualified) of the new queue table.

`type_name`

The name of an existing type that describes the payload of each entry in the queue table. For information about defining a type, see `CREATE TYPE`.

`option_name option_value`

The name of any options that will be associated with the new queue table, and the corresponding value for the option. If the call to `CREATE QUEUE TABLE` includes duplicate option names, the server will return an error. The following values are accepted:

<code>SORT_LIST</code>	Use the <code>SORT_LIST</code> option to control the dequeuing order of the queue; specify the names of the column(s) that will be used to sort the queue (in ascending order). The currently accepted values are the following combinations of <code>enq_time</code> and <code>priority</code> : <code>enq_time. priority</code> <code>priority. enq_time</code> <code>priority</code> <code>enq_time</code> Any other value will return an <code>ERROR</code> .
<code>MULTIPLE_CONSUMERS</code>	A <code>BOOLEAN</code> value that indicates if a message can have more than one consumer (<code>TRUE</code>), or are limited to one consumer per message (<code>FALSE</code>).
<code>MESSAGE_GROUPING</code>	Specify <code>none</code> to indicate that each message should be dequeued individually, or <code>transactional</code> to indicate that messages that are added to the queue as a result of one transaction should be dequeued as a group.
<code>STORAGE_CLAUSE</code>	Use <code>STORAGE_CLAUSE</code> to specify table attributes. <code>STORAGE_CLAUSE</code> may be <code>TABLESPACE tablespace_name</code> , <code>PCTFREE integer</code> , <code>PCTUSED integer</code> , <code>INITTRANS integer</code> , <code>MAXTRANS integer</code> , <code>STORAGE storage_option</code> Where <code>storage_option</code> is one or more of the following: <code>MINEXTENTS integer</code> , <code>MAXEXTENTS integer</code> , <code>PCTINCREASE integer</code> , <code>INITIAL size_clause</code> , <code>NEXT</code> , <code>FREELISTS integer</code> , <code>OPTIMAL size_clause</code> , <code>BUFFER_POOL {KEEP RECYCLE DEFAULT}</code> . Please note that only the <code>TABLESPACE</code> option is enforced; all others are accepted for compatibility and ignored. Use the <code>TABLESPACE</code> clause to specify the name of a tablespace in which the table will be created.

Examples

You must create a user-defined type before creating a queue table; the type describes the columns and data types within the table. The following command creates a type named `work_order`:

```
CREATE TYPE work_order AS (name VARCHAR2, project TEXT, completed BOOLEAN);
```

The following command uses the `work_order` type to create a queue table named `work_order_table`:

```
CREATE QUEUE TABLE work_order_table OF work_order (sort_list (enq_time, priority));
```

See Also

[*ALTER QUEUE TABLE*](#), [*DROP QUEUE TABLE*](#)

CREATE ROLE

Name

CREATE ROLE -- define a new database role

Synopsis

```
CREATE ROLE <name> [IDENTIFIED BY <password> [REPLACE old_password]]
```

Description

CREATE ROLE adds a new role to the 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: The 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

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. (Please note: if you are using an Oracle database, 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 compatible with Oracle databases.)

`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. (Please note: In 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 Advanced Server, there are other forms of the `CREATE SCHEMA` command that are not compatible with Oracle databases.

Examples

```
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;
```

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 Sequence Manipulation Functions of Database Compatibility for Oracle Developers Reference Guide.

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 compatible with Oracle databases.

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:

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

Select the next number from this sequence:

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

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

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

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

Select the next number from this sequence:

```
SELECT supplier_seq.NEXTVAL FROM DUAL;

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

See Also

ALTER SEQUENCE, DROP SEQUENCE

CREATE SYNONYM

Name

CREATE SYNONYM -- define a new synonym

Synopsis

```
CREATE [OR REPLACE] [PUBLIC] SYNONYM [<schema>.<syn_name>]
    FOR <object_schema>.<object_name>[@<dblink_name>];
```

Description

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

- tables
- views
- materialized views
- sequences
- stored procedures
- stored functions
- types
- objects that are accessible through a database link
- other 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.

dblink_name

`dblink_name` specifies the name of the database link through which an object is accessed.

Include the `REPLACE` clause to replace an existing synonym definition with a new synonym definition.

Include the `PUBLIC` clause to create the synonym in the `public` schema. The `CREATE PUBLIC SYNONYM` command, compatible with Oracle databases, 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](#)

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> } [, ...]  
)  
[ WITH ( ROWIDS [= <value> ] ) ]  
[ ON COMMIT { PRESERVE ROWS | DELETE ROWS } ]  
[ TABLESPACE <tablespace> ]  
{ NOPARALLEL | PARALLEL [ <integer> ] }
```

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 [ <create_index_statement> ] TABLESPACE <tablespace> ] |  
  PRIMARY KEY ( <column_name> [, ...] )  
    [ USING INDEX [ <create_index_statement> ] 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.

The `PARALLEL` clause sets the degree of parallelism for a table. If you do not specify the `PARALLEL` clause, the server determines a value based on the relation size.

The `NOPARALLEL` clause reset the parallelism for default execution, and `reloptions` will show the `parallel_workers` parameter as 0.

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 compatible with Oracle databases.) 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 Advanced Server, refer to Data Types of *Database Compatibility for Oracle Developers Reference Guide*.

`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.)

PRIMARY KEY - column constraint

PRIMARY KEY (column_name [, ...]) - table constraint

The primary key constraint specifies that a column or columns of a table may contain only unique (non-duplicate), non-null values. Technically, **PRIMARY KEY** is merely a combination of **UNIQUE** and **NOT NULL**, but identifying a set of columns as primary key also provides metadata about the design of the schema, as a primary key implies that other tables may rely on this set of columns as a unique identifier for rows.

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.

CHECK (expression)

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

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

REFERENCES reftable [(refcolumn `` ) ] [ ``ON DELETE action] - column constraint

FOREIGN KEY (column [, ...]) REFERENCES reftable [(refcolumn [, ...])] [ON DELETE action] - table constraint

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.

`WITH ( ROWIDS [= value ] )`

The `ROWIDS` option for a table include `value` equals to `TRUE/ON/1` or `FALSE/OFF/0`. When set to `TRUE/ON/1`, a `ROWID` column is created in the new table. `ROWID` is an auto-incrementing value based on a sequence that starts with 1 and assigned to each row of a table. If a value is not specified then the default value is always `TRUE`.

By default, a unique index is created on a `ROWID` column. The `ALTER` and `DROP` operations are restricted on a `ROWID` column.

`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 compatible with Oracle databases. 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 [create_index_statement] 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.

If you specify the create_index_statement option, the database server creates an index enabling unique or primary key constraints. The columns specified in the constraint and the columns of an index must be the same, but their order of appearance may differ.

PARALLEL

Include the PARALLEL clause to specify the degree of parallelism for the table; set the parallel_workers parameter equal to the degree of parallelism to perform a parallel scan of a table. Alternatively, if you specify PARALLEL but do not include a degree of parallelism, an index will use default parallelism.

NOPARALLEL

Specify NOPARALLEL for default execution.

integer

The integer indicates the degree of parallelism, which is a number of parallel_workers used in the parallel operation to perform a parallel scan on a table.

Notes

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:

```
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.

```
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  
                  REFERENCES dept(deptno)  
);
```

Create table dept in tablespace diskvoll1:

```
CREATE TABLE dept (
    deptno          NUMBER(2) NOT NULL CONSTRAINT dept_pk PRIMARY KEY,
    dname           VARCHAR2(14),
    loc             VARCHAR2(13)
) TABLESPACE diskvoll1;
```

The following PARALLEL example creates a table sales and sets a degree of parallelism to 6:

```
CREATE TABLE sales (deptno number) PARALLEL 6 WITH (FILLFACTOR=66);
```

The following NOPARALLEL example creates a table sales_order and sets a degree of parallelism to 0:

```
CREATE TABLE sales_order (deptno number) NOPARALLEL WITH (FILLFACTOR=66);
```

The following example creates a table named dept; the definition creates a unique key on the dname column. The constraint dept_dname_uq identifies the dname column as a unique key. The preceding statement includes the USING_INDEX clause, which explicitly creates an index on a table dept with the index statement specified to enable the unique constraint.

```
CREATE TABLE dept (
    deptno          NUMBER(2) NOT NULL,
    dname           VARCHAR2(14),
    loc             VARCHAR2(13),
    CONSTRAINT dept_dname_uq UNIQUE(dname)
        USING INDEX (CREATE UNIQUE INDEX idx_dept_dname_uq ON dept(dname))
);
```

The following example creates a table named emp; the definition creates a primary key on the ename column. The emp_ename_pk constraint identifies the ename column as a primary key of the emp table. The preceding statement includes the USING_INDEX clause, which explicitly creates an index on a table emp with the index statement specified to enable the primary constraint.

```
CREATE TABLE emp (
    empno          NUMBER(4) NOT NULL,
    ename          VARCHAR2(10),
    job            VARCHAR2(9),
    sal            NUMBER(7,2),
    deptno         NUMBER(2),
    CONSTRAINT emp_ename_pk PRIMARY KEY (ename)
        USING INDEX (CREATE INDEX idx_emp_ename_pk ON emp (ename))
);
```

See Also

[*ALTER TABLE*](#), [*DROP TABLE*](#)

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.

query

A query statement (a SELECT command). Refer to SELECT for a description of the allowed syntax.

CREATE TRIGGER

Name

CREATE TRIGGER -- define a simple trigger

Synopsis

```
CREATE [ OR REPLACE ] TRIGGER <name>
{ BEFORE | AFTER | INSTEAD OF }
{ INSERT | UPDATE | DELETE | TRUNCATE }
  [ OR { INSERT | UPDATE | DELETE | TRUNCATE } ] [, ...]
ON <table>
[ REFERENCING { OLD AS <old> | NEW AS <new> } ...]
[ FOR EACH ROW ]
[ WHEN <condition> ]
[ DECLARE
  [ PRAGMA AUTONOMOUS_TRANSACTION; ]
  <declaration>; [, ...] ]
BEGIN
  <statement>; [, ...]
[ EXCEPTION
  { WHEN <exception> [ OR <exception> ] [...] THEN
    <statement>; [, ...] } [, ...]
]
END
```

Name

CREATE TRIGGER -- define a compound trigger

Synopsis

```
CREATE [ OR REPLACE ] TRIGGER <name>
FOR { INSERT | UPDATE | DELETE | TRUNCATE }
  [ OR { INSERT | UPDATE | DELETE | TRUNCATE } ] [, ...]
ON <table>
[ REFERENCING { OLD AS <old> | NEW AS <new> } ...]
[ WHEN <condition> ]
COMPOUND TRIGGER
[ <private_declaration>; ] ...
[ <procedure_or_function_definition> ] ...
<compound_trigger_definition>
END
```

Where `private_declaration` is an identifier of a private variable that can be accessed by any procedure or function. There can be zero, one, or more private variables. `private_declaration` can be any of the following:

- Variable Declaration
- Record Declaration
- Collection Declaration
- REF CURSOR and Cursor Variable Declaration
- TYPE Definitions for Records, Collections, and REF CURSORS
- Exception
- Object Variable Declaration

Where procedure_or_function_definition :=

```
procedure_definition | function_definition
```

Where procedure_definition :=

```
PROCEDURE proc_name[ argument_list ]  
  [ options_list ]  
  { IS | AS }  
  procedure_body  
END [ proc_name ];
```

Where procedure_body :=

```
[ declaration; ] [, ...]  
BEGIN  
  statement; [...]  
[ EXCEPTION  
  { WHEN exception [OR exception] [...] THEN statement; }  
  [...]  
]
```

Where function_definition :=

```
FUNCTION func_name [ argument_list ]  
  RETURN rettype [ DETERMINISTIC ]  
  [ options_list ]  
  { IS | AS }  
  function_body  
END [ func_name ] ;
```

Where function_body :=

```
[ declaration; ] [, ...]  
BEGIN  
  statement; [...]  
[ EXCEPTION  
  { WHEN exception [ OR exception ] [...] THEN statement; }  
  [...]  
]
```

Where compound_trigger_definition :=

```
{ compound_trigger_event } { IS | AS }  
  compound_trigger_body  
END [ compound_trigger_event ] [ ... ]
```

Where compound_trigger_event :=

```
[ BEFORE STATEMENT | BEFORE EACH ROW | AFTER EACH ROW |
  AFTER STATEMENT | INSTEAD OF EACH ROW ]
```

Where compound_trigger_body :=

```
[ declaration; ] [, ...]
BEGIN
    statement; [...]
[ EXCEPTION
    { WHEN exception [OR exception] [...] THEN statement; }
    [...]
]
```

Description

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

If you are using the CREATE TRIGGER keywords to create a new trigger, the name of the new trigger must not match any existing trigger defined on the same table. New triggers will be created in the same schema as the table on which the triggering event is defined.

If you are updating the definition of an existing trigger, use the CREATE OR REPLACE TRIGGER keywords.

When you use syntax that is compatible with Oracle to create a trigger, the trigger runs as a SECURITY DEFINER function.

Parameters

name

The name of the trigger to create.

BEFORE | AFTER

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

INSTEAD OF

INSTEAD OF trigger modifies an updatable view; the trigger will execute to update the underlying table(s) appropriately. The INSTEAD OF trigger is executed for each row of the view that is updated or modified.

INSERT | UPDATE | DELETE | TRUNCATE

Defines the triggering event.

table

The name of the table or view 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 simple 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 compound trigger definition includes a statement-level trigger having a WHEN clause, then the trigger is executed without evaluating the expression in the WHEN clause. Similarly, if a compound trigger

definition includes a row-level trigger having a `WHEN` clause, then the trigger is executed if the expression evaluates to `TRUE`.

If the trigger includes the `INSTEAD OF` keywords, it may not include a `WHEN` clause. A `WHEN` clause cannot contain subqueries.

`REFERENCING { OLD AS old | NEW AS new } ...`

`REFERENCING` clause to reference old rows and new rows, but restricted in that `old` may only be replaced by an identifier named `old` or any equivalent that is saved in all lowercase (for example, `REFERENCING OLD AS old`, `REFERENCING OLD AS OLD`, or `REFERENCING OLD AS "old"`). Also, `new` may only be replaced by an identifier named `new` or any equivalent that is saved in all lowercase (for example, `REFERENCING NEW AS new`, `REFERENCING NEW AS NEW`, or `REFERENCING NEW AS "new"`).

Either one, or both phrases `OLD AS old` and `NEW AS new` may be specified in the `REFERENCING` clause (for example, `REFERENCING NEW AS New OLD AS Old`).

This clause is not compatible with Oracle databases in that identifiers other than `old` or `new` may not be used.

`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.

`PRAGMA AUTONOMOUS_TRANSACTION`

`PRAGMA AUTONOMOUS_TRANSACTION` is the directive that sets the trigger as an autonomous transaction.

declaration

A variable, type, `REF CURSOR`, or subprogram declaration. If subprogram declarations are included, they must be declared after all other variable, type, and `REF CURSOR` declarations.

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.

```
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;
```

(continues on next page)

(continued from previous page)

```

    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.

```

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;
```

The following is an example of a compound trigger that records a change to the employee salary by defining a compound trigger hr_trigger on table emp.

First, create a table named emp.

```

CREATE TABLE emp(EMPNO INT, ENAME TEXT, SAL INT, DEPTNO INT);
CREATE TABLE
```

Then, create a compound trigger named hr_trigger. The trigger utilizes each of the four timing-points to modify the salary with an INSERT, UPDATE, or DELETE statement. In the global declaration section, the initial salary is declared as 10,000.

```

CREATE OR REPLACE TRIGGER hr_trigger
    FOR INSERT OR UPDATE OR DELETE ON emp
    COMPOUND TRIGGER
-- Global declaration.
var_sal NUMBER := 10000;

BEFORE STATEMENT IS
BEGIN
    var_sal := var_sal + 1000;
    DBMS_OUTPUT.PUT_LINE('Before Statement: ' || var_sal);
END BEFORE STATEMENT;

BEFORE EACH ROW IS
BEGIN
    var_sal := var_sal + 1000;
    DBMS_OUTPUT.PUT_LINE('Before Each Row: ' || var_sal);
```

(continues on next page)

(continued from previous page)

```

END BEFORE EACH ROW;

AFTER EACH ROW IS
BEGIN
    var_sal := var_sal + 1000;
    DBMS_OUTPUT.PUT_LINE('After Each Row: ' || var_sal);
END AFTER EACH ROW;

AFTER STATEMENT IS
BEGIN
    var_sal := var_sal + 1000;
    DBMS_OUTPUT.PUT_LINE('After Statement: ' || var_sal);
END AFTER STATEMENT;

END hr_trigger;

Output: Trigger created.

```

INSERT the record into table emp.

```
INSERT INTO emp (EMPNO, ENAME, SAL, DEPTNO) VALUES (1111, 'SMITH', 10000, 20);
```

The INSERT statement produces the following output:

```

Before Statement: 11000
Before each row: 12000
After each row: 13000
After statement: 14000
INSERT 0 1

```

The UPDATE statement will update the employee salary record, setting the salary to 15000 for a specific employee number.

```
UPDATE emp SET SAL = 15000 where EMPNO = 1111;
```

The UPDATE statement produces the following output:

```

Before Statement: 11000
Before each row: 12000
After each row: 13000
After statement: 14000
UPDATE 1

SELECT * FROM emp;
 EMPNO | ENAME |  SAL  | DEPTNO
-----+-----+-----+-----
   1111 | SMITH | 15000 |      20
(1 row)

```

The DELETE statement deletes the employee salary record.

```
DELETE from emp where EMPNO = 1111;
```

The DELETE statement produces the following output:

```

Before Statement: 11000
Before each row: 12000
After each row: 13000
After statement: 14000
DELETE 1

SELECT * FROM emp;
  EMPNO | ENAME | SAL | DEPTNO
-----+-----+-----+-----
(0 rows)

```

The TRUNCATE statement removes all the records from the emp table.

```

CREATE OR REPLACE TRIGGER hr_trigger
FOR TRUNCATE ON emp
  COMPOUND TRIGGER
-- Global declaration.
var_sal NUMBER := 10000;
BEFORE STATEMENT IS
BEGIN
  var_sal := var_sal + 1000;
  DBMS_OUTPUT.PUT_LINE('Before Statement: ' || var_sal);
END BEFORE STATEMENT;

AFTER STATEMENT IS
BEGIN
  var_sal := var_sal + 1000;
  DBMS_OUTPUT.PUT_LINE('After Statement: ' || var_sal);
END AFTER STATEMENT;

END hr_trigger;

Output: Trigger created.

```

The TRUNCATE statement produces the following output:

```

TRUNCATE emp;
Before Statement: 11000
After statement: 12000
TRUNCATE TABLE

```

Note: The TRUNCATE statement may be used only at a BEFORE STATEMENT or AFTER STATEMENT timing-point.

The following example creates a compound trigger named hr_trigger on the emp table with a WHEN condition that checks and prints employee salary whenever an INSERT, UPDATE, or DELETE statement affects the emp table. The database evaluates the WHEN condition for a row-level trigger, and the trigger is executed once per row if the WHEN condition evaluates to TRUE. The statement-level trigger is executed irrespective of the WHEN condition.

```

CREATE OR REPLACE TRIGGER hr_trigger
FOR INSERT OR UPDATE OR DELETE ON emp
  REFERENCING NEW AS new OLD AS old
  WHEN (old.sal > 5000 OR new.sal < 8000)
  COMPOUND TRIGGER

BEFORE STATEMENT IS
BEGIN

```

(continues on next page)

(continued from previous page)

```

    DBMS_OUTPUT.PUT_LINE('Before Statement');
END BEFORE STATEMENT;

BEFORE EACH ROW IS
BEGIN
    DBMS_OUTPUT.PUT_LINE('Before Each Row: ' || :OLD.sal || ' ' || :NEW.sal);
END BEFORE EACH ROW;

AFTER EACH ROW IS
BEGIN
    DBMS_OUTPUT.PUT_LINE('After Each Row: ' || :OLD.sal || ' ' || :NEW.sal);
END AFTER EACH ROW;

AFTER STATEMENT IS
BEGIN
    DBMS_OUTPUT.PUT_LINE('After Statement');
END AFTER STATEMENT;

END hr_trigger;

```

Insert the record into table emp.

```
INSERT INTO emp(EMPNO, ENAME, SAL, DEPTNO) VALUES(1111, 'SMITH', 1600, 20);
```

The INSERT statement produces the following output:

```

Before Statement
Before Each Row: 1600
After Each Row: 1600
After Statement
INSERT 0 1

```

The UPDATE statement will update the employee salary record, setting the salary to 7500.

```
UPDATE emp SET SAL = 7500 where EMPNO = 1111;
```

The UPDATE statement produces the following output:

```

Before Statement
Before Each Row: 1600 7500
After Each Row: 1600 7500
After Statement
UPDATE 1

SELECT * from emp;
 empno |  ename  |  sal  | deptno
-----+-----+-----+-----
  1111 | SMITH   | 7500  |     20
(1 row)

```

The DELETE statement deletes the employee salary record.

```
DELETE from emp where EMPNO = 1111;
```

The DELETE statement produces the following output:

```
Before Statement
Before Each Row: 7500
After Each Row: 7500
After Statement
DELETE 1

SELECT * from emp;
 empno |  ename  |  sal  | deptno
-----+-----+-----+-----
(0 rows)
```

See Also

ALTER TRIGGER, DROP TRIGGER

CREATE TYPE

Name

CREATE TYPE -- define a new user-defined type, which can be an object type, a collection type (a nested table type or a varray type), or a composite type.

Synopsis

Object Type

```
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>
}
```

Nested Table Type

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

Varray Type

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

Composite Type

```
CREATE [ OR REPLACE ] TYPE <name> { IS | AS }
( [ attribute <datatype> ][, ...]
)
```

Description

CREATE TYPE defines a new, user-defined data type. The types that can be created are an object type, a nested table type, a varray type, or a composite type. Nested table and varray types belong to the category of types known as collections.

Composite types are not compatible with Oracle databases. However, composite types can be accessed by SPL programs as with other types described in this section.

Note:

- For packages only, a composite type can be included in a user-defined record type declared with the TYPE IS RECORD statement within the package specification or package body. Such nested structure is not permitted in other SPL programs such as functions, procedures, triggers, etc.
- In the CREATE TYPE command, 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.
- 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.
- 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.

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 or composite type.

`datatype`

The data type that defines an attribute of the object type or composite 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

Creating an Object Type

Create object type `addr_obj_typ`.

```
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`.

```
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`.

```
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  
);
```

Creating a Collection Type

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

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

Creating and Using a Composite Type

The following example shows the usage of a composite type accessed from an anonymous block.

The composite type is created by the following:

```
CREATE OR REPLACE TYPE emphist_typ AS (
    empno          NUMBER(4),
    ename          VARCHAR2(10),
    hiredate       DATE,
    job            VARCHAR2(9),
    sal            NUMBER(7,2)
);
```

The following is the anonymous block that accesses the composite type:

```
DECLARE
    v_emphist      EMPHIST_TYP;
BEGIN
    v_emphist.empno    := 9001;
    v_emphist.ename    := 'SMITH';
    v_emphist.hiredate := '01-AUG-17';
    v_emphist.job      := 'SALESMAN';
    v_emphist.sal      := 8000.00;
    DBMS_OUTPUT.PUT_LINE('  EMPNO: ' || v_emphist.empno);
    DBMS_OUTPUT.PUT_LINE('  ENAME: ' || v_emphist.ename);
    DBMS_OUTPUT.PUT_LINE(' HIREDATE: ' || v_emphist.hiredate);
    DBMS_OUTPUT.PUT_LINE('   JOB: ' || v_emphist.job);
    DBMS_OUTPUT.PUT_LINE('   SAL: ' || v_emphist.sal);
END;

      EMPNO: 9001
      ENAME: SMITH
 HIREDATE: 01-AUG-17 00:00:00
       JOB: SALESMAN
       SAL: 8000.00
```

The following example shows the usage of a composite type accessed from a user-defined record type, declared within a package body.

The composite type is created by the following:

```
CREATE OR REPLACE TYPE salhist_typ AS (
    startdate      DATE,
    job            VARCHAR2(9),
    sal            NUMBER(7,2)
);
```

The package specification is defined by the following:

```
CREATE OR REPLACE PACKAGE emp_salhist
IS
    PROCEDURE fetch_emp (
        p_empno      IN NUMBER
    );
END;
```

The package body is defined by the following:

```

CREATE OR REPLACE PACKAGE BODY emp_salhist
IS
    TYPE emprec_typ IS RECORD (
        empno      NUMBER(4),
        ename       VARCHAR(10),
        salhist     SALHIST_TYP
    );
    TYPE emp_arr_typ IS TABLE OF emprec_typ INDEX BY BINARY_INTEGER;
    emp_arr          emp_arr_typ;

    PROCEDURE fetch_emp (
        p_empno      IN NUMBER
    )
    IS
        CURSOR emp_cur IS SELECT e.empno, e.ename, h.startdate, h.job, h.sal
            FROM emp e, jobhist h
            WHERE e.empno = p_empno
            AND e.empno = h.empno;

        i            INTEGER := 0;
    BEGIN
        DBMS_OUTPUT.PUT_LINE('EMPNO  ENAME      STARTDATE  JOB          ' ||
            'SAL          ');
        DBMS_OUTPUT.PUT_LINE('-----  -----  -----  -----  ' ||
            '-----');

        FOR r_emp IN emp_cur LOOP
            i := i + 1;
            emp_arr(i) := (r_emp.empno, r_emp.ename,
                (r_emp.startdate, r_emp.job, r_emp.sal));
        END LOOP;

        FOR i IN 1 .. emp_arr.COUNT LOOP
            DBMS_OUTPUT.PUT_LINE(emp_arr(i).empno || ' ' ||
                RPAD(emp_arr(i).ename,8) || ' ' ||
                TO_CHAR(emp_arr(i).salhist.startdate,'DD-MON-YY') || ' ' ||
                RPAD(emp_arr(i).salhist.job,10) || ' ' ||
                TO_CHAR(emp_arr(i).salhist.sal,'99,999.99'));
        END LOOP;
    END;
END;

```

Note that in the declaration of the TYPE emprec_typ IS RECORD data structure in the package body, the salhist field is defined with the SALHIST_TYP composite type as created by the CREATE TYPE salhist_typ statement.

The associative array definition TYPE emp_arr_typ IS TABLE OF emprec_typ references the record type data structure emprec_typ that includes the field salhist that is defined with the SALHIST_TYP composite type.

Invocation of the package procedure that loads the array from a join of the emp and jobhist tables, then displays the array content is shown by the following:

```

EXEC emp_salhist.fetch_emp(7788);

EMPNO  ENAME      STARTDATE  JOB          SAL
-----  -----  -----  -----  -----
7788    SCOTT      19-APR-87  CLERK        1,000.00

```

(continues on next page)

(continued from previous page)

7788	SCOTT	13-APR-88	CLERK	1,040.00
7788	SCOTT	05-MAY-90	ANALYST	3,000.00

EDB-SPL Procedure successfully completed

See Also

CREATE TYPE BODY, DROP TYPE

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`.

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 pragma, declarations, and SPL statements that comprise the body of the function or procedure. The pragma may be `PRAGMA AUTONOMOUS_TRANSACTION` to set the function or procedure as an autonomous transaction.

`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.

```

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.

```

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;

```

See Also

CREATE TYPE, DROP TYPE

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 an 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

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

CREATE USER|ROLE... PROFILE MANAGEMENT CLAUSES

Name

CREATE USER|ROLE

Synopsis

```
CREATE USER|ROLE <name> [[WITH] option [...]]
```

where *option* can be the following compatible clauses:

```
    PROFILE <profile_name>
|  ACCOUNT {LOCK|UNLOCK}
|  PASSWORD EXPIRE [AT '<timestamp>']
```

or *option* can be the following non-compatible clauses:

```
|  LOCK TIME '<timestamp>'
```

For information about the administrative clauses of the CREATE USER or CREATE ROLE command that are supported by Advanced Server, see the PostgreSQL core documentation available at:

<https://www.postgresql.org/docs/current/static/sql-commands.html>

Description

CREATE ROLE|USER... PROFILE adds a new role with an associated profile to an Advanced Server database cluster.

Roles created with the CREATE USER command are (by default) login roles. Roles created with the CREATE ROLE command are (by default) not login roles. To create a login account with the CREATE ROLE command, you must include the LOGIN keyword.

Only a database superuser can use the CREATE USER|ROLE clauses that enforce profile management; these clauses enforce the following behaviors:

Include the PROFILE clause and a *profile_name* to associate a pre-defined profile with a role, or to change which pre-defined profile is associated with a user.

Include the ACCOUNT clause and the LOCK or UNLOCK keyword to specify that the user account should be placed in a locked or unlocked state.

Include the LOCK TIME '*timestamp*' clause and a date/time value to lock the role at the specified time, and unlock the role at the time indicated by the PASSWORD_LOCK_TIME parameter of the profile assigned to this role. If LOCK TIME is used with the ACCOUNT LOCK clause, the role can only be unlocked by a database superuser with the ACCOUNT UNLOCK clause.

Include the `PASSWORD EXPIRE` clause with the optional `AT 'timestamp'` keywords to specify a date/time when the password associated with the role will expire. If you omit the `AT 'timestamp'` keywords, the password will expire immediately.

Each login role may only have one profile. To discover the profile that is currently associated with a login role, query the `profile` column of the `DBA_USERS` view.

Parameters

`name`

The name of the role.

`profile_name`

The name of the profile associated with the role.

`timestamp`

The date and time at which the clause will be enforced. When specifying a value for `timestamp`, enclose the value in single-quotes.

Examples

The following example uses `CREATE USER` to create a login role named `john` who is associated with the `acctg_profile` profile:

```
CREATE USER john PROFILE acctg_profile IDENTIFIED BY "1safepwd";
```

`john` can log in to the server, using the password `1safepwd`.

The following example uses `CREATE ROLE` to create a login role named `john` who is associated with the `acctg_profile` profile:

```
CREATE ROLE john PROFILE acctg_profile LOGIN PASSWORD "1safepwd";
```

`john` can log in to the server, using the password `1safepwd`.

See Also

ALTER USER|ROLE... PROFILE MANAGEMENT CLAUSES

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.

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

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.

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 the `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.

`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

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

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 PUBLIC DATABASE LINK

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, ALTER DIRECTORY

DROP FUNCTION

Name

DROP FUNCTION -- remove a function

Synopsis

```
DROP FUNCTION [ IF EXISTS ] <name>
  [ ([ [ <argmode> ] [ <argname> ] <argtype> ] [, ...]) ]
  [ CASCADE | RESTRICT ]
```

Description

DROP FUNCTION removes the definition of an existing function. To execute this command 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 this is an overloaded function. (This requirement is not compatible with Oracle databases. In Oracle, only the function name is specified. Advanced Server allows overloading of function names, so the function signature given by the input argument data types is required in the Advanced Server DROP FUNCTION command of an overloaded function.)

Usage of IF EXISTS, CASCADE, or RESTRICT is not compatible with Oracle databases and is used only by Advanced Server.

Parameters

IF EXISTS

Do not throw an error if the function does not exist. A notice is issued in this case.

name

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

argmode

The mode of an argument: 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 compatible with Oracle databases and applies only to 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 compatible with Oracle databases and applies only to Advanced Server.)

argtype

The data type of an argument of the function. (Specification of argtype is not compatible with Oracle databases and applies only to Advanced Server.)

CASCADE

Automatically drop objects that depend on the function (such as operators or triggers), and in turn all objects that depend on those objects.

RESTRICT

Refuse to drop the function if any objects depend on it. This is the default.

Examples

The following command removes the `emp_comp` function.

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

See Also

CREATE FUNCTION

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

CREATE INDEX, ALTER INDEX

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

DROP PROCEDURE

Name

DROP PROCEDURE -- remove a procedure

Synopsis

```
DROP PROCEDURE [ IF EXISTS ] <name>
  [ ([ [ <argmode> ] [ <argname> ] <argtype> ] [, ...]) ]
  [ CASCADE | RESTRICT ]
```

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. All input (IN, IN OUT) argument data types to the procedure must be specified if this is an overloaded procedure. (This requirement is not compatible with Oracle databases. In Oracle, only the procedure name is specified. Advanced Server allows overloading of procedure names, so the procedure signature given by the input argument data types is required in the Advanced Server DROP PROCEDURE command of an overloaded procedure.)

Usage of IF EXISTS, CASCADE, or RESTRICT is not compatible with Oracle databases and is used only by Advanced Server.

Parameters

IF EXISTS

Do not throw an error if the procedure does not exist. A notice is issued in this case.

name

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

argmode

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

argname

The name of an argument. Note that DROP PROCEDURE does not actually pay any attention to argument names, since only the argument data types are needed to determine the procedure's identity. (Specification of argname is not compatible with Oracle databases and applies only to Advanced Server.)

argtype

The data type of an argument of the procedure. (Specification of `argtype` is not compatible with Oracle databases and applies only to Advanced Server.)

CASCADE

Automatically drop objects that depend on the procedure, and in turn all objects that depend on those objects.

RESTRICT

Refuse to drop the procedure if any objects depend on it. This is the default.

Examples

The following command removes the `select_emp` procedure.

```
DROP PROCEDURE select_emp;
```

See Also

CREATE PROCEDURE, ALTER PROCEDURE

DROP PROFILE

Name

DROP PROFILE – drop a user-defined profile

Synopsis

```
DROP PROFILE [IF EXISTS] <profile_name> [CASCADE | RESTRICT];
```

Description

Include the IF EXISTS clause to instruct the server to not throw an error if the specified profile does not exist. The server will issue a notice if the profile does not exist.

Include the optional CASCADE clause to reassign any users that are currently associated with the profile to the default profile, and then drop the profile. Include the optional RESTRICT clause to instruct the server to not drop any profile that is associated with a role. This is the default behavior.

Parameters

profile_name

The name of the profile being dropped.

Examples

The following example drops a profile named acctg_profile:

```
DROP PROFILE acctg_profile CASCADE;
```

The command first re-associates any roles associated with the acctg_profile profile with the default profile, and then drops the acctg_profile profile.

The following example drops a profile named acctg_profile:

```
DROP PROFILE acctg_profile RESTRICT;
```

The RESTRICT clause in the command instructs the server to not drop acctg_profile if there are any roles associated with the profile.

See Also

CREATE PROFILE, ALTER PROFILE

DROP QUEUE

Advanced Server includes extra syntax (not offered by Oracle) with the `DROP QUEUE SQL` command. This syntax can be used in association with `DBMS_AQADM`.

Name

`DROP QUEUE --` drop an existing queue.

Synopsis

Use `DROP QUEUE` to drop an existing queue:

```
DROP QUEUE [IF EXISTS] <name>
```

Description

`DROP QUEUE` allows a superuser or a user with the `aq_administrator_role` privilege to drop an existing queue.

Parameters

`name`

The name (possibly schema-qualified) of the queue that is being dropped.

`IF EXISTS`

Include the `IF EXISTS` clause to instruct the server to not return an error if the queue does not exist. The server will issue a notice.

Examples

The following example drops a queue named `work_order`:

```
DROP QUEUE work_order;
```

See Also

CREATE QUEUE, ALTER QUEUE

DROP QUEUE TABLE

Advanced Server includes extra syntax (not offered by Oracle) with the `DROP QUEUE TABLE SQL` command. This syntax can be used in association with `DBMS_AQADM`.

Name

`DROP QUEUE TABLE`-- drop a queue table.

Synopsis

Use `DROP QUEUE TABLE` to delete a queue table:

```
DROP QUEUE TABLE [ IF EXISTS ] <name> [, ...]
[CASCADE | RESTRICT]
```

Description

`DROP QUEUE TABLE` allows a superuser or a user with the `aq_administrator_role` privilege to delete a queue table.

Parameters

`name`

The name (possibly schema-qualified) of the queue table that will be deleted.

`IF EXISTS`

Include the `IF EXISTS` clause to instruct the server to not return an error if the queue table does not exist. The server will issue a notice.

`CASCADE`

Include the `CASCADE` keyword to automatically delete any objects that depend on the queue table.

`RESTRICT`

Include the `RESTRICT` keyword to instruct the server to refuse to delete the queue table if any objects depend on it. This is the default.

Examples

The following example deletes a queue table named `work_order_table` and any objects that depend on it:

```
DROP QUEUE TABLE work_order_table CASCADE;
```

See Also

CREATE QUEUE TABLE, ALTER QUEUE TABLE

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.

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 DROP PUBLIC SYNONYM command, compatible with Oracle databases, drops a synonym that resides in the public schema:

```
DROP PUBLIC SYNONYM syn_name;
```

The following example drops the synonym, personnel:

```
DROP SYNONYM personnel;
```

See Also

[*CREATE SYNONYM*](#)

DROP ROLE

Name

`DROP ROLE -- remove a database role`

Synopsis

<code>DROP ROLE <name> [CASCADE]</code>

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:

<code>DROP ROLE admins;</code>

See Also

[CREATE ROLE](#), [SET ROLE](#), [GRANT](#), [REVOKE](#)

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

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, ...);
CREATE TABLE items
  (order_id REFERENCES orders, quantity int, ...);
```

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

CREATE TABLE, ALTER TABLE

DROP TABLESPACE

Name

DROP TABLESPACE -- remove a tablespace

Synopsis

```
DROP TABLESPACE <tablespacename>
```

Description

DROP TABLESPACE removes a tablespace from the system.

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

Parameters

tablespacename

The name of a tablespace.

Examples

To remove tablespace employee_space from the system:

```
DROP TABLESPACE employee_space;
```

See Also

ALTER TABLESPACE

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 a trigger named emp_salary_trig:

```
DROP TRIGGER emp_salary_trig;
```

See Also

CREATE TRIGGER, ALTER TRIGGER

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 nor to composite 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 the object type named `addr_obj_typ`:

`DROP TYPE addr_obj_typ;`

Drop the nested table type named `budget_tbl_typ`:

`DROP TYPE budget_tbl_typ;`

See Also

[CREATE TYPE](#), [CREATE TYPE BODY](#)

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 named `john` that owns no objects nor has been granted any privileges on other objects:

```
DROP USER john;
```

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`:

```
DROP USER john CASCADE;
```

See Also

CREATE USER, ALTER USER

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 a database superuser or 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).

The form of the `DROP VIEW` command compatible with Oracle does not support a `CASCADE` clause; to drop a view and its dependencies, use the PostgreSQL-compatible form of the `DROP VIEW` command. For more information, see the PostgreSQL core documentation at:

<https://www.postgresql.org/docs/current/static/sql-dropview.html>

Parameters

name

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

Examples

This command will remove the view named `dept_30`:

```
DROP VIEW dept_30;
```

See Also

[CREATE VIEW](#)

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);
```


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 ] [ argname ] argtype ] [, ...] )
TO { username | groupname | PUBLIC } [, ...]
[ WITH GRANT OPTION ]

GRANT { EXECUTE | ALL [ PRIVILEGES ] }
ON PROCEDURE progname
[ ( [ [ argmode ] [ argname ] 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 } [, ...]
```

(continues on next page)

(continued from previous page)

```
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 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.

64.1 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 Advanced Server syntax for granting the `EXECUTE` privilege is not fully compatible with Oracle databases. 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. For functions, 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 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.

64.2 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, 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

64.3 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. The role is exempt from all security policies in the database.

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 joe;
```

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 commands compatible with Oracle databases.

See Also

REVOKE, ALTER ROLE

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`:

```
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:

```
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:

```
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:

```
CREATE TABLE deptnames (
    deptname          VARCHAR2(14)
);
INSERT INTO deptnames SELECT dname FROM dept;
```


LOCK

Name

`LOCK -- lock a table`

Synopsis

<code>LOCK TABLE <name> [, ...] IN <lockmode> MODE [NOWAIT]</code>
--

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, 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 the server uses the most restrictive mode, `ACCESS EXCLUSIVE`. (`ACCESS EXCLUSIVE` is not compatible with Oracle databases. In Advanced Server, this configuration 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.

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 } [, ...]
FROM { username | groupname } [, ...]

REVOKE CREATE [ PUBLIC ] DATABASE LINK
FROM { username | groupname }

REVOKE DROP PUBLIC DATABASE LINK
```

(continues on next page)

(continued from previous page)

```

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 option compatible with Oracle databases. By default Oracle always cascades dependent privileges, but 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.

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 `REVOKE` command, compatible with Oracle databases.

Examples

Revoke insert privilege for the public on table `emp`:

```
REVOKE INSERT ON emp FROM PUBLIC;
```

Revoke all privileges from user `mary` on view `salesemp`:

```
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](#)

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.

Note: Executing a ROLLBACK in a plpgsql procedure will throw an error if there is an Oracle-style SPL procedure on the runtime stack.

Examples

To abort all changes:

```
ROLLBACK;
```

See Also

COMMIT, ROLLBACK TO SAVEPOINT, SAVEPOINT

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, if needed.

ROLLBACK TO SAVEPOINT 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:

```
\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

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

SELECT

Name

SELECT -- retrieve rows from a table or view

Synopsis

```
SELECT [ optimizer_hint ] [ ALL | DISTINCT | UNIQUE ]
  * | 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 SIBLINGS BY expression [ ASC | DESC ] [, ...] ] ]
[ GROUP BY { expression | ROLLUP ( expr_list ) |
            CUBE ( expr_list ) | GROUPING SETS ( expr_list ) } [, ...]
  [ LEVEL ] ]
[ HAVING condition [, ...] ]
[ { UNION [ ALL ] | INTERSECT | MINUS } select ]
[ ORDER BY expression [ ASC | DESC ] [, ...] ]
[ FOR UPDATE [WAIT n|NOWAIT|SKIP LOCKED]]
```

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` | `UNIQUE` eliminates duplicate rows from the result. `ALL` (the default) will return all candidate rows, including duplicates. (See `DISTINCT` | `UNIQUE` 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 Functions and Operators of *Database Compatibility for Oracle Developers Reference Guide* for information about optimizer hints.

71.1 FROM Clause

The FROM clause specifies one or more source tables for a SELECT statement. The syntax is:

```
FROM source [, ...]
```

Where `source` can be one of 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 the following:

`[ 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.

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.

Example

The following example selects all of the entries from the `dept` table:

```
SELECT * FROM dept;
deptno |  dname   |  loc
-----+-----+-----
    10 | ACCOUNTING | NEW YORK
    20 | RESEARCH  | DALLAS
    30 | SALES     | CHICAGO
    40 | OPERATIONS | BOSTON
(4 rows)
```

71.2 WHERE Clause

The optional `WHERE` clause has the 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.

Example

The following example joins the contents of the `emp` and `dept` tables, `WHERE` the value of the `deptno` column in the `emp` table is equal to the value of the `deptno` column in the `dept` table:

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

(continues on next page)

(continued from previous page)

deptno	dname	empno	ename	mgr	hiredate
10	ACCOUNTING	7934	MILLER	7782	23-JAN-82 00:00:00
10	ACCOUNTING	7782	CLARK	7839	09-JUN-81 00:00:00
10	ACCOUNTING	7839	KING		17-NOV-81 00:00:00
20	RESEARCH	7788	SCOTT	7566	19-APR-87 00:00:00
20	RESEARCH	7566	JONES	7839	02-APR-81 00:00:00
20	RESEARCH	7369	SMITH	7902	17-DEC-80 00:00:00
20	RESEARCH	7876	ADAMS	7788	23-MAY-87 00:00:00
20	RESEARCH	7902	FORD	7566	03-DEC-81 00:00:00
30	SALES	7521	WARD	7698	22-FEB-81 00:00:00
30	SALES	7844	TURNER	7698	08-SEP-81 00:00:00
30	SALES	7499	ALLEN	7698	20-FEB-81 00:00:00
30	SALES	7698	BLAKE	7839	01-MAY-81 00:00:00
30	SALES	7654	MARTIN	7698	28-SEP-81 00:00:00
30	SALES	7900	JAMES	7698	03-DEC-81 00:00:00

(14 rows)

71.3 GROUP BY Clause

The optional GROUP BY clause has the form:

```
GROUP BY { expression | ROLLUP ( expr_list ) |
          CUBE ( expr_list ) | GROUPING SETS ( expr_list ) } [, ...]
```

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.

ROLLUP, CUBE, and GROUPING SETS are extensions to the GROUP BY clause for supporting multidimensional analysis.

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.

Example

The following example computes the sum of the sal column in the emp table, grouping the results by department number:

```
SELECT deptno, SUM(sal) AS total
FROM emp
GROUP BY deptno;

deptno | total
-----+-----
10 | 8750.00
20 | 10875.00
30 | 9400.00
(3 rows)
```

71.4 HAVING Clause

The optional HAVING clause has the form:

```
HAVING condition
```

where `condition` is the same as specified for the `WHERE` clause.

HAVING eliminates group rows that do not satisfy the specified 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.

Example

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;

deptno | total
-----+-----
      10 | 8750.00
      30 | 9400.00
(2 rows)
```

71.5 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.

Example

The SELECT list in the following example specifies that the result set should include the `empno` column, the `ename` column, the `mgr` column and the `hiredate` column:

```
SELECT empno, ename, mgr, hiredate FROM emp;

empno | ename  | mgr  | hiredate
-----+-----+-----+-----
  7934 | MILLER | 7782 | 23-JAN-82 00:00:00
  7782 | CLARK  | 7839 | 09-JUN-81 00:00:00
  7839 | KING   |      | 17-NOV-81 00:00:00
  7788 | SCOTT  | 7566 | 19-APR-87 00:00:00
  7566 | JONES  | 7839 | 02-APR-81 00:00:00
  7369 | SMITH  | 7902 | 17-DEC-80 00:00:00
  7876 | ADAMS  | 7788 | 23-MAY-87 00:00:00
  7902 | FORD   | 7566 | 03-DEC-81 00:00:00
```

(continues on next page)

(continued from previous page)

7521		WARD		7698		22-FEB-81		00:00:00
7844		TURNER		7698		08-SEP-81		00:00:00
7499		ALLEN		7698		20-FEB-81		00:00:00
7698		BLAKE		7839		01-MAY-81		00:00:00
7654		MARTIN		7698		28-SEP-81		00:00:00
7900		JAMES		7698		03-DEC-81		00:00:00

(14 rows)

71.6 UNION Clause

The UNION clause has the 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`.

71.7 INTERSECT Clause

The `INTERSECT` clause has the 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)`.

71.8 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`.

71.9 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 start_expression`

The rows returned by the `FROM` clause on which `start_expression` evaluates to `TRUE` become the root nodes of the hierarchy.

`ORDER SIBLINGS BY expression [ ASC | DESC ] [, ...]`

Sibling rows of the hierarchy are ordered by `expression` in the result set.

Note: Advanced Server does not support the use of `AND` (or other operators) in the `CONNECT BY` clause.

71.10 ORDER BY Clause

The optional `ORDER BY` clause has the form:

```
ORDER BY expression [ ASC | DESC ] [, ...]
```

`expression` can be the name or ordinal number of an output column (`SELECT` list item), or it can be an arbitrary expression formed from input-column values.

The `ORDER BY` clause causes the result rows to be sorted according to the specified expressions. If two rows are equal according to the 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 `AS` clause.

It is also possible to use arbitrary expressions in the `ORDER BY` clause, including columns that do not appear in the `SELECT` result list. Thus the following statement is valid:

```
SELECT ename FROM emp ORDER BY empno;
```

A limitation of this feature is that an `ORDER BY` clause applying to the result of a `UNION`, `INTERSECT`, or `MINUS` clause may only specify an output column name or number, not an expression.

If an `ORDER BY` expression is a simple name that matches both a result column name and an input column name, `ORDER BY` will interpret it as the result column name. This is the opposite of the choice that `GROUP BY` will make in the same situation. This inconsistency is made to be compatible with the SQL standard.

Optionally one may add the key word `ASC` (ascending) or `DESC` (descending) after any expression in the `ORDER BY` clause. If not specified, `ASC` is assumed by default.

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.

Note: If `SELECT DISTINCT` is specified or if a `SELECT` statement includes the `SELECT DISTINCT ... ORDER BY` clause then all the expressions in `ORDER BY` must be present in the select list of the `SELECT DISTINCT` query.

Examples

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;
```

deptno	dname	loc
10	ACCOUNTING	NEW YORK
40	OPERATIONS	BOSTON
20	RESEARCH	DALLAS
30	SALES	CHICAGO

(4 rows)

```
SELECT * FROM dept ORDER BY 2;
```

deptno	dname	loc
10	ACCOUNTING	NEW YORK
40	OPERATIONS	BOSTON
20	RESEARCH	DALLAS
30	SALES	CHICAGO

(4 rows)

The following example uses the `SELECT DISTINCT ... ORDER BY` clause to fetch the `job` and `deptno` from table `emp`:

```
CREATE TABLE EMP (EMPNO NUMBER(4) NOT NULL,
ENAME VARCHAR2(10),
JOB VARCHAR2(9),
DEPTNO NUMBER(2));
```

```

INSERT INTO EMP VALUES (7369, 'SMITH', 'CLERK', 20);
INSERT 0 1
INSERT INTO EMP VALUES (7499, 'ALLEN', 'SALESMAN', 30);
INSERT 0 1
INSERT INTO EMP VALUES (7521, 'WARD', 'SALESMAN', 30);
INSERT 0 1
INSERT INTO EMP VALUES (7566, 'JONES', 'MANAGER', 20);
INSERT 0 1

```

```

SELECT DISTINCT e.job, e.deptno FROM emp e ORDER BY e.job, e.deptno;
 job          | deptno
-----+-----
 CLERK       |      20
 MANAGER     |      20
 SALESMAN    |      30
(3 rows)

```

71.11 DISTINCT | UNIQUE Clause

If a `SELECT` statement specifies `DISTINCT` or `UNIQUE`, all duplicate rows are removed from the result set (one row is kept from each group of duplicates). The `DISTINCT` or `UNIQUE` clause are synonymous when used with a `SELECT` statement. The `ALL` keyword specifies the opposite: all rows are kept; that is the default.

Error messages resulting from the improper use of a `SELECT` statement that includes the `DISTINCT` or `UNIQUE` keywords will include both the `DISTINCT` | `UNIQUE` keywords as shown below:

```
psql: ERROR: FOR UPDATE is not allowed with DISTINCT/UNIQUE clause
```

71.12 FOR UPDATE Clause

The `FOR UPDATE` clause takes the form:

```
FOR UPDATE [WAIT n|NOWAIT|SKIP LOCKED]
```

`FOR UPDATE` causes the rows retrieved by the `SELECT` statement to be locked as though for update. This prevents a row from being modified or deleted by other transactions until the current transaction ends; any transaction that attempts to `UPDATE`, `DELETE`, or `SELECT FOR UPDATE` a selected row will be blocked until the current transaction ends. 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 first 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 cannot be clearly identified with individual table rows (for example, with aggregation).

Use `FOR UPDATE` options to specify locking preferences:

- Include the `WAIT n` keywords to specify the number of seconds (or fractional seconds) that the `SELECT` statement will wait for a row locked by another session. Use a decimal form to specify fractional seconds; for example, `WAIT 1.5` instructs the server to wait one and a half seconds. Specify up to 4 digits to the right of the decimal.
- Include the `NOWAIT` keyword to report an error immediately if a row cannot be locked by the current session.

- Include `SKIP LOCKED` to instruct the server to lock rows if possible, and skip rows that are already locked by another session.

SET CONSTRAINTS

Name

`SET CONSTRAINTS` -- set constraint checking modes for the current transaction

Synopsis

<code>SET CONSTRAINTS { ALL name [, ...] } { DEFERRED IMMEDIATE }</code>
--

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.

SET ROLE

Name

SET ROLE -- set the current user identifier of the current session

Synopsis

```
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 is 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

You can use this command, 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. 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. 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`:

```
SET ROLE admins;
```

User `mary` reverts back to her own identity:

```
SET ROLE NONE;
```


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 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.

TRUNCATE

Name

TRUNCATE -- empty a table

Synopsis

```
TRUNCATE TABLE <name> [DROP STORAGE]
```

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.

The DROP STORAGE clause is accepted for compatibility, but is ignored.

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

The following command truncates a table named `accounts`:

```
TRUNCATE TABLE accounts;
```

See Also

DROP VIEW, DELETE

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.

table

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

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 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';
```


CONCLUSION

EDB Postgres™ Advanced Server Database Compatibility for Oracle® Developers SQL Guide

Copyright © 2007 - 2020 EnterpriseDB Corporation.

All rights reserved.

EnterpriseDB® Corporation

34 Crosby Drive, Suite 201, Bedford, MA 01730, USA

T +1 781 357 3390 F +1 978 467 1307 E

info@enterprisedb.com

www.enterprisedb.com

- EnterpriseDB and Postgres Enterprise Manager are registered trademarks of EnterpriseDB Corporation. EDB and EDB Postgres are trademarks of EnterpriseDB Corporation. Oracle is a registered trademark of Oracle, Inc. Other trademarks may be trademarks of their respective owners.
- EDB designs, establishes coding best practices, reviews, and verifies input validation for the logon UI for EDB products where present. EDB follows the same approach for additional input components, however the nature of the product may require that it accepts freeform SQL, WMI or other strings to be entered and submitted by trusted users for which limited validation is possible. In such cases it is not possible to prevent users from entering incorrect or otherwise dangerous inputs.
- EDB reserves the right to add features to products that accept freeform SQL, WMI or other potentially dangerous inputs from authenticated, trusted users in the future, but will ensure all such features are designed and tested to ensure they provide the minimum possible risk, and where possible, require superuser or equivalent privileges.
- EDB does not warrant that we can or will anticipate all potential threats and therefore our process cannot fully guarantee that all potential vulnerabilities have been addressed or considered.

A

- ALTER DIRECTORY, 3
- ALTER INDEX, 5
- ALTER PROCEDURE, 7
- ALTER PROFILE, 9
- ALTER QUEUE, 13
- ALTER QUEUE TABLE, 17
- ALTER ROLE – Managing Database Link
and DBMS_RLS Privileges, 21
- ALTER ROLE... IDENTIFIED BY, 19
- ALTER SEQUENCE, 25
- ALTER SESSION, 27
- ALTER TABLE, 29
- ALTER TABLESPACE, 35
- ALTER TRIGGER, 33
- ALTER USER... IDENTIFIED BY, 37
- ALTER USER|ROLE... PROFILE MANAGEMENT
CLAUSES, 39

C

- CALL, 41
- COMMENT, 43
- COMMIT, 45
- Conclusion, 233
- CREATE DATABASE, 47
- CREATE DIRECTORY, 61
- CREATE FUNCTION, 63
- CREATE INDEX, 69
- CREATE MATERIALIZED VIEW, 73
- CREATE PACKAGE, 75
- CREATE PACKAGE BODY, 77
- CREATE PROCEDURE, 83
- CREATE PROFILE, 89
- CREATE PUBLIC DATABASE LINK, 49
- CREATE QUEUE, 93
- CREATE QUEUE TABLE, 95
- CREATE ROLE, 97
- CREATE SCHEMA, 99
- CREATE SEQUENCE, 101
- CREATE SYNONYM, 105
- CREATE TABLE, 107
- CREATE TABLE AS, 115

- CREATE TRIGGER, 117
- CREATE TYPE, 127
- CREATE TYPE BODY, 135
- CREATE USER, 139
- CREATE USER|ROLE... PROFILE
MANAGEMENT CLAUSES, 141
- CREATE VIEW, 143

D

- DELETE, 145
- DROP DATABASE LINK, 147
- DROP DIRECTORY, 149
- DROP FUNCTION, 151
- DROP INDEX, 153
- DROP PACKAGE, 155
- DROP PROCEDURE, 157
- DROP PROFILE, 159
- DROP QUEUE, 161
- DROP QUEUE TABLE, 163
- DROP ROLE, 167
- DROP SEQUENCE, 169
- DROP SYNONYM, 165
- DROP TABLE, 171
- DROP TABLESPACE, 173
- DROP TRIGGER, 175
- DROP TYPE, 177
- DROP USER, 179
- DROP VIEW, 181

E

- EXEC, 183

G

- GRANT, 185

I

- INSERT, 193

L

- LOCK, 197

R

- REVOKE, 199

ROLLBACK, [203](#)

ROLLBACK TO SAVEPOINT, [205](#)

S

SAVEPOINT, [207](#)

SELECT, [209](#)

SET CONSTRAINTS, [221](#)

SET ROLE, [223](#)

SET TRANSACTION, [225](#)

T

TRUNCATE, [227](#)

U

UPDATE, [229](#)