14 SQL Statements: CREATE LIBRARY to CREATE SCHEMA

This chapter contains the following SQL statements:

CREATE LIBRARY

Purpose

Use the `CREATE LIBRARY` statement to create a schema object associated with an operating-system shared library. The name of this schema object can then be used in the *`call_spec`* of `CREATE FUNCTION` or `CREATE PROCEDURE` statements, or when declaring a function or procedure in a package or type, so that SQL and PL/SQL can call to third-generation-language (3GL) functions and procedures.

See Also:

CREATE FUNCTION and *Oracle Database PL/SQL Language Reference* for more information on functions and procedures

Prerequisites

The `CREATE LIBRARY` statement is valid only on platforms that support shared libraries and dynamic linking.

To create a library in your own schema, you must have the `CREATE LIBRARY` system privilege. To create a library in another user's schema, you must have the `CREATE ANY LIBRARY` system privilege.

To use the library in the *`call_spec`* of a `CREATE FUNCTION` statement, or when declaring a function in a package or type, you must have the `EXECUTE` object privilege on the library and the `CREATE FUNCTION` system privilege. Refer to *Oracle Database PL/SQL Language Reference* for information on the *`call_spec`* of a `CREATE FUNCTION` statement.

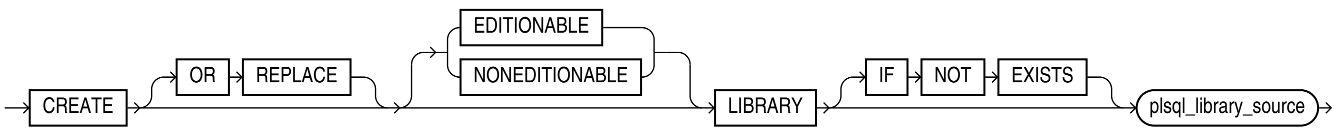
To use the library in the *`call_spec`* of a `CREATE PROCEDURE` statement, or when declaring a procedure in a package or type, you must have the `EXECUTE` object privilege on the library and the `CREATE PROCEDURE` system privilege. Refer to *Oracle Database PL/SQL Language Reference* for information on the *`call_spec`* of a `CREATE PROCEDURE` statement.

To execute a procedure or function defined with the *`call_spec`* (including a procedure or function defined within a package or type), you must have the `EXECUTE` object privilege on the procedure or function (but you do not need the `EXECUTE` object privilege on the library).

Syntax

Libraries are defined using PL/SQL. Therefore, the syntax diagram in this book shows only the SQL keywords. Refer to *Oracle Database PL/SQL Language Reference* for the PL/SQL syntax, semantics, and examples.

***create_library*::=**



(*`plsql_library_source`*: See *Oracle Database PL/SQL Language Reference*.)

Semantics

OR REPLACE

Specify `OR REPLACE` to re-create the library if it already exists. Use this clause to change the definition of an existing library without dropping, re-creating, and regranting object privileges granted on it.

Users who had previously been granted privileges on a redefined library can still access the library without being regranted the privileges.

IF NOT EXISTS

Specifying `IF NOT EXISTS` has the following effects:

- If the library does not exist, a new library is created at the end of the statement.
- If the library exists, this is the library you have at the end of the statement. A new one is not created because the older one is detected.

You can have *one* of `OR REPLACE` or `IF NOT EXISTS` in a statement at a time. Using both `OR REPLACE` with `IF NOT EXISTS` in the very same statement results in the following error: `ORA-11541: REPLACE and IF NOT EXISTS cannot coexist in the same DDL statement`.

Using `IF EXISTS` with `CREATE` results in `ORA-11543: Incorrect IF NOT EXISTS clause for CREATE statement`.

[EDITIONABLE | NONEDITIONABLE]

Use these clauses to specify whether the library is an editioned or noneditioned object if editioning is enabled for the schema object type `LIBRARY` in *`schema`*. The default is `EDITIONABLE`. For information about editioned and noneditioned objects, see *Oracle Database Development Guide*.

plsql_library_source

See *Oracle Database PL/SQL Language Reference* for the syntax and semantics of the *`plsql_library_source`*.

CREATE LOCKDOWN PROFILE

Purpose

Use the `CREATE LOCKDOWN PROFILE` statement to create a PDB lockdown profile. You can use PDB lockdown profiles in a multitenant container database (CDB) to restrict user operations in PDBs.

After you create a PDB lockdown profile, you can add restrictions to the profile with the `ALTER LOCKDOWN PROFILE` statement. You can restrict user operations associated with certain database features, options, and SQL statements.

When a lockdown profile is assigned to a PDB, users in that PDB cannot perform the operations that are the disabled for the profile. To assign a lockdown profile, set its name for the value of the `PDB_LOCKDOWN` initialization parameter. You can assign a lockdown profile to individual PDBs, or to all PDBs in a CDB or application container, as follows:

- If you set `PDB_LOCKDOWN` while connected to a CDB root, then the lockdown profile applies to all PDBs in the CDB. It does not apply to the CDB root.
- If you set `PDB_LOCKDOWN` while connected to an application root, then the lockdown profile applies to the application root and all PDBs in the application container.
- If you set `PDB_LOCKDOWN` while connected to a particular PDB, then the lockdown profile applies to that PDB and overrides the lockdown profile for the CDB or application container, if one exists.

See Also:

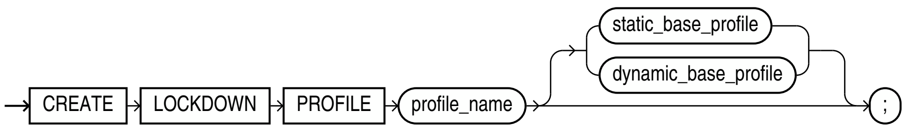
- ALTER LOCKDOWN PROFILE and DROP LOCKDOWN PROFILE
- *Oracle Database Security Guide* for more information on PDB lockdown profiles

Prerequisites

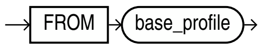
- The `CREATE LOCKDOWN PROFILE` statement must be issued from the CDB or the Application Root.
- You must have the `CREATE LOCKDOWN PROFILE` system privilege in the container in which the statement is issued.
- The PDB lockdown profile name must be unique in the container in which the statement is issued.

Syntax

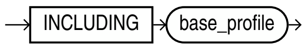
***create_lockdown_profile*::=**



***static_base_profile* ::=**



***dynamic_base_profile* ::=**



Semantics

profile_name

You can create a new PDB lockdown profile with a name that you specify. The name must satisfy the requirements listed in "Database Object Naming Rules ". The lockdown profile can be derived from a static, or dynamic base profile.

static_base_profile

Use this option to create a new lockdown profile with a base profile. The rules of the base profile in effect at profile creation time will be copied to the new lockdown profile. Changes to the base profile after the lockdown profile is created will not apply to the lockdown profile.

dynamic_base_profile

Use this option to create a new lockdown profile that will change with changes to the base profile. The new lockdown profile will inherit `DISABLE` rules of the base profile as well and subsequent changes to the base profile. The rules of the base profile have precedence in any conflict with rules that may be explicitly added to the lockdown profile. For example, the `OPTION_VALUE` clause of the base profile takes precedence over the `OPTION_VALUE` clause of the dynamic base profile.

Example

The following statement creates PDB lockdown profile `hr_prof` with a dynamic base profile `PRIVATE_DBAAS`:

```
CREATE LOCKDOWN PROFILE hr_prof INCLUDING PRIVATE_DBAAS;
```

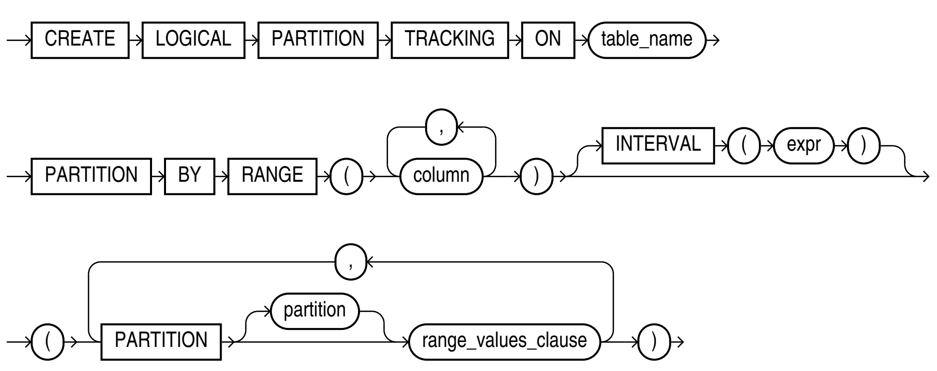
CREATE LOGICAL PARTITION TRACKING

Purpose

Use the `CREATE LOGICAL PARTITION TRACKING` statement to define a logical partitioning scheme on a table for being leveraged by materialized views and logical partition change tracking. You can define the logical partitions of your tables independently of any existing or non-existing partitioning schema of a table.

Syntax

create_logical_partition_tracking::=



Semantics

Logical partition tracking is supported on a single key column within the table. The datatype of the key column can be of the following data types: `NUMBER`, `DATE`, `CHAR`, `VARCHAR`, `VARCHAR2`, `TIMESTAMP`, `TIMSTAMP WITH TIME ZONE`.

Only `RANGE` and `INTERVAL` logical partitions are supported on the base table.

See Also:

- *Refreshing Materialized Views* of the *Data Warehousing Guide*.

CREATE MATERIALIZED VIEW

Purpose

Use the `CREATE MATERIALIZED VIEW` statement to create a **materialized view**. A materialized view is a database object that contains the results of a query. The `FROM` clause of the query can name tables, views, and other materialized views. Collectively these objects are called **master tables** (a replication term) or **detail tables** (a data warehousing term). This reference uses "master tables" for consistency. The databases containing the master tables are called the **master databases**.

> **Note:**
>
> The keyword `SNAPSHOT` is supported in place of `MATERIALIZED VIEW` for backward compatibility.

For replication purposes, materialized views allow you to maintain read-only copies of remote data on your local node. You can select data from a materialized view as you would from a table or view. In replication environments, the materialized views commonly created are **primary key**, **rowid**, **object**, and **subquery** materialized views.

> **See Also:**
>
> *Oracle Database Administrator’s Guide* for information on the types of materialized views used to support replication

For data warehousing purposes, the materialized views commonly created are **materialized aggregate views**, **single-table materialized aggregate views**, and **materialized join views**. All three types of materialized views can be used by query rewrite, an optimization technique that transforms a user request written in terms of master tables into a semantically equivalent request that includes one or more materialized views.

> **See Also:**
>
> - ALTER MATERIALIZED VIEW
> - *Oracle Database Data Warehousing Guide* for information on the types of materialized views used to support data warehousing

Prerequisites

The privileges required to create a materialized view should be granted directly rather than through a role.

To create a materialized view **in your own schema:**

- You must have been granted the `CREATE MATERIALIZED VIEW` system privilege **and** either the `CREATE TABLE` or `CREATE ANY TABLE` system privilege.
- You must also have access to any master tables of the materialized view that you do not own, either through a `READ` or `SELECT` object privilege on each of the tables or through the `READ ANY TABLE` or `SELECT ANY TABLE` system privilege.

To create a materialized view **in another user's schema:**

- You must have the `CREATE ANY MATERIALIZED VIEW` system privilege.
- The owner of the materialized view must have the `CREATE TABLE` system privilege. The owner must also have access to any master tables of the materialized view that the schema owner does not own (for example, if the master tables are on a remote database) and to any materialized view logs defined on those master tables, either through a `READ` or `SELECT` object privilege on each of the tables or through the `READ ANY TABLE` or `SELECT ANY TABLE` system privilege.

To create a refresh-on-commit materialized view (`REFRESH ON COMMIT` clause), in addition to the preceding privileges, you must have the `ON COMMIT REFRESH` object privilege on any master tables that you do not own or you must have the `ON COMMIT REFRESH` system privilege.

To create the materialized view **with query rewrite enabled**, in addition to the preceding privileges:

- If the schema owner does not own the master tables, then the schema owner must have the `GLOBAL QUERY REWRITE` privilege or the `QUERY REWRITE` object privilege on each table outside the schema.
- If you are defining the materialized view on a prebuilt container (`ON PREBUILT TABLE` clause), then you must have the `READ` or `SELECT` privilege `WITH GRANT OPTION` on the container table.

The user whose schema contains the materialized view must have sufficient quota in the target tablespace to store the master table and index of the materialized view or must have the `UNLIMITED TABLESPACE` system privilege.

When you create a materialized view, Oracle Database creates one internal table and at least one index, and may create one view, all in the schema of the materialized view. Oracle Database uses these objects to maintain the materialized view data. You must have the privileges necessary to create these objects.

You can create the following types of local materialized views (including both `ON COMMIT` and `ON DEMAND`) on master tables with commit SCN-based materialized view logs:

- Materialized aggregate views, including materialized aggregate views on a single table
- Materialized join views
- Primary-key-based and rowid-based single table materialized views
- `UNION ALL` materialized views, where each `UNION ALL` branch is one of the above materialized view types

You cannot create remote materialized views on master tables with commit SCN-based materialized view logs.

Creating a materialized view on master tables with different types of materialized view logs (that is, a master table with timestamp-based materialized view logs and a master table with commit SCN-based materialized view logs) is not supported and causes `ORA-32414`.

To specify an edition in the *`evaluation_edition_clause`* or the *`unusable_editions_clause`*, you must have the `USE` privilege on the edition.

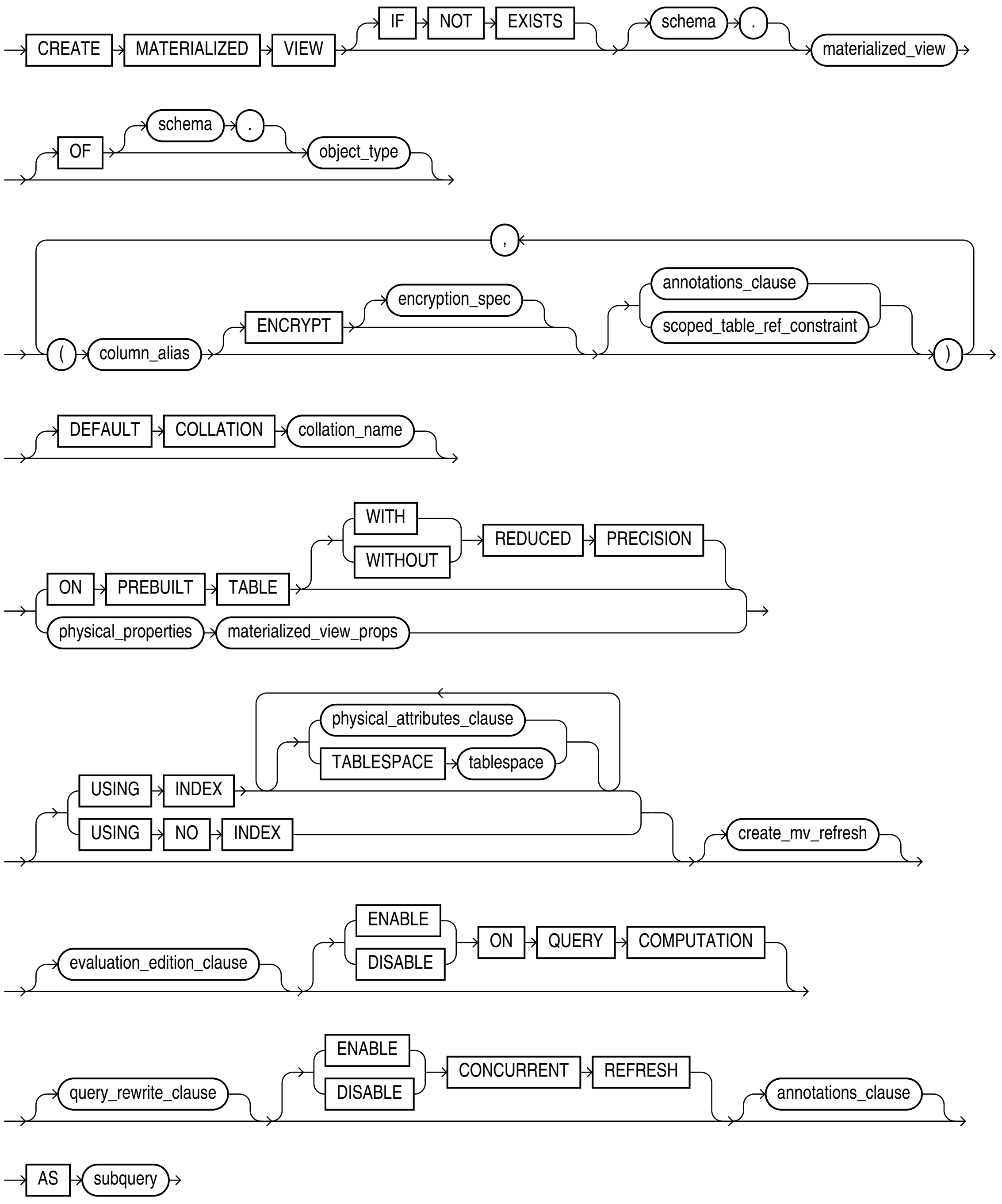
To perform select from a materialized view, you must have the `SELECT ANY TABLE` system privilege.

See Also:

- CREATE TABLE, CREATE VIEW , and CREATE INDEX for information on these privileges
- *Oracle Database Administrator's Guide* for information about the prerequisites that apply to creating replication materialized views
- *Oracle Database Data Warehousing Guide* for information about the prerequisites that apply to creating data warehousing materialized views

Syntax

***create_materialized_view*::=**

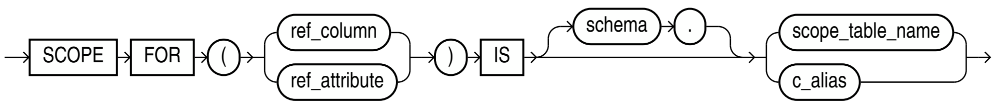


(*scoped_table_ref_constraint*::=, *physical_properties*::=, *materialized_view_props*::=, *physical_attributes_clause*::=, *create_mv_refresh*::=, *evaluation_edition_clause*::=, *query_rewrite_clause*::=, *subquery*::=, *annotations_clause*)

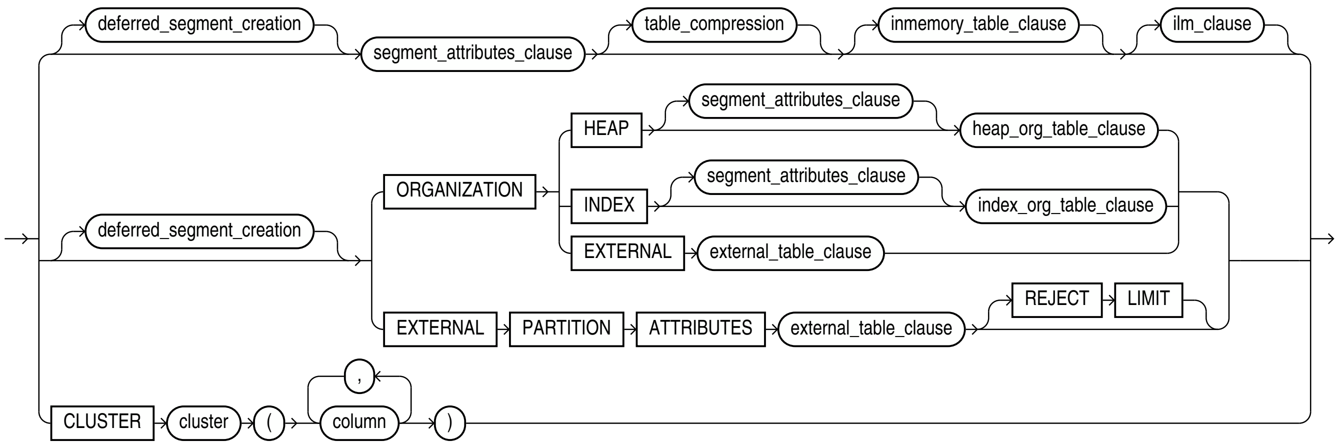
***annotations_clause*::=**

For the full syntax and semantics of the `annotations_clause` see *annotations_clause*.

***scoped_table_ref_constraint*::=**

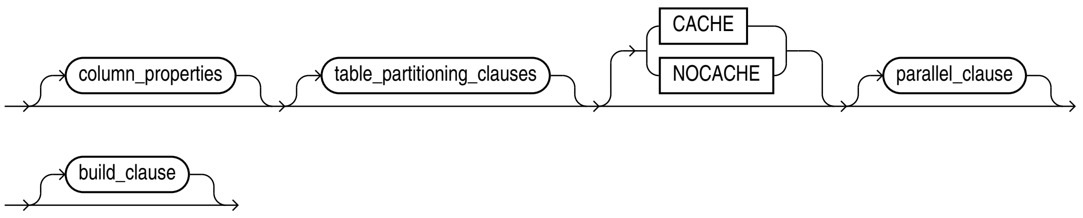


***physical_properties*::=**



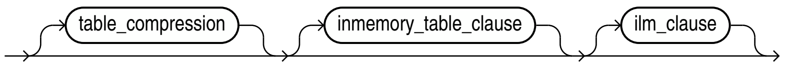
(*deferred_segment_creation*::=, *segment_attributes_clause*::=, *table_compression*::=, *inmemory_table_clause*::=, *heap_org_table_clause*::=, *index_org_table_clause*::=)

***materialized_view_props*::=**

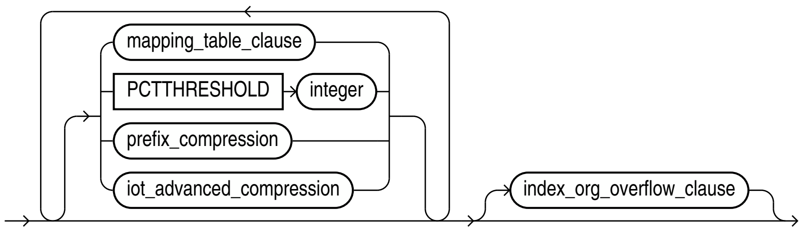


(*column_properties*::=, *table_partitioning_clauses*::=—part of `CREATE TABLE` syntax, *parallel_clause*::=, *build_clause*::=)

***heap_org_table_clause*::=**

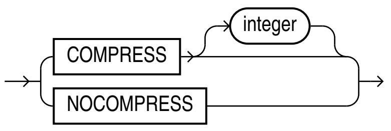


***index_org_table_clause*::=**

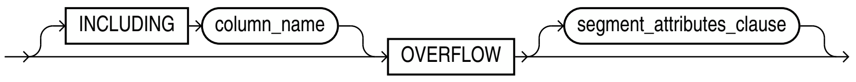


(`mapping_table_clause`: not supported with materialized views, *prefix_compression*::=, *index_org_overflow_clause*::=)

***prefix_compression*::=**

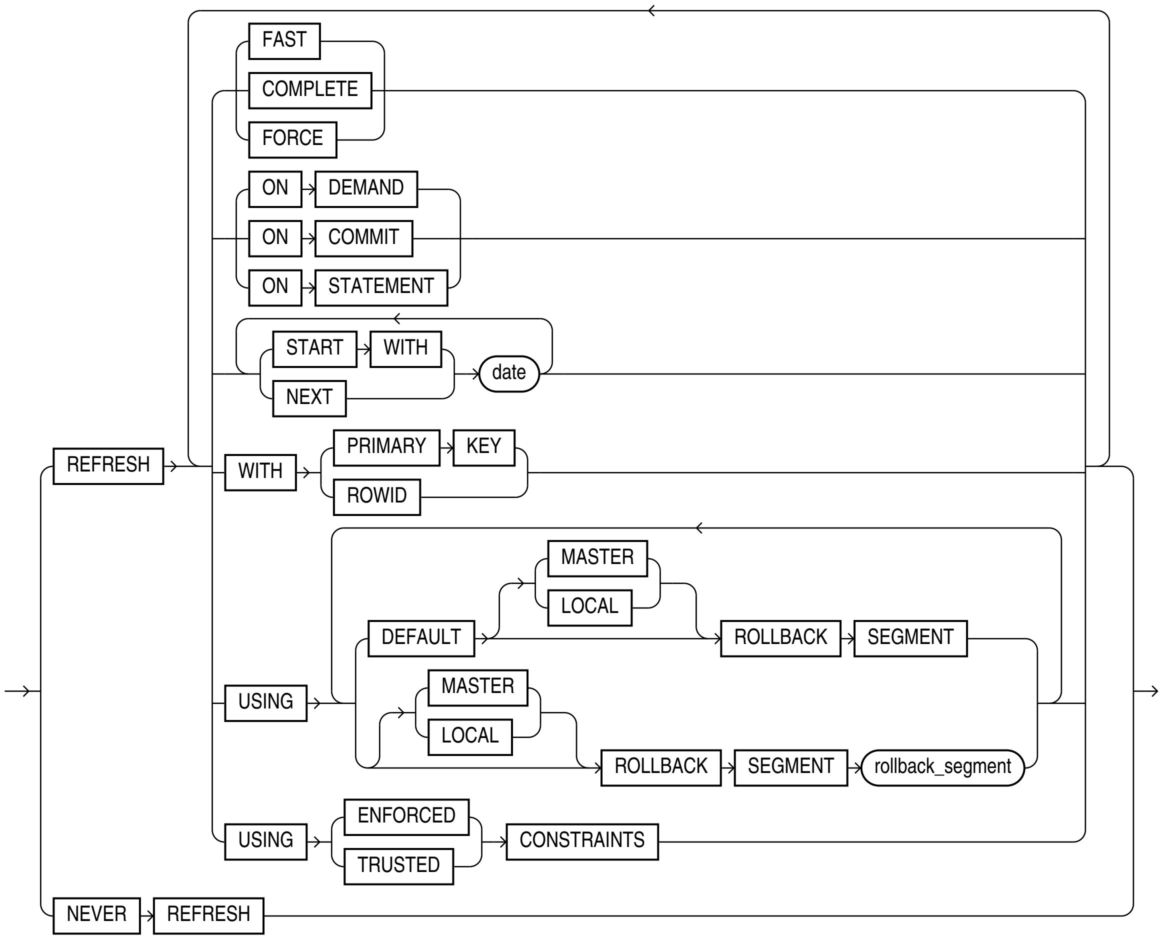


***index_org_overflow_clause*::=**

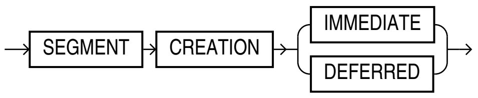


(*segment_attributes_clause*::=)

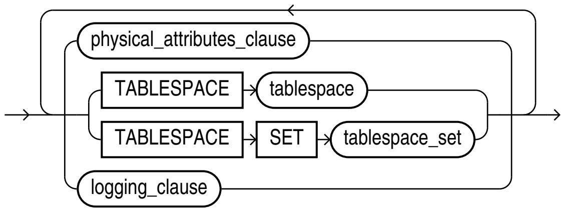
***create_mv_refresh*::=**



***deferred_segment_creation*::=**

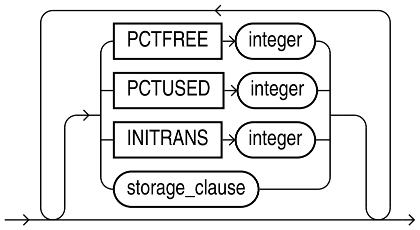


***segment_attributes_clause*::=**



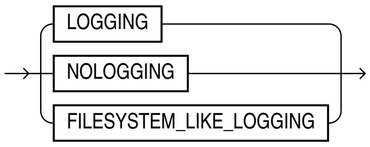
(*physical_attributes_clause::=*, `TABLESPACE SET`: not supported with `CREATE MATERIALIZED VIEW`, *logging_clause::=*)

***physical_attributes_clause*::=**

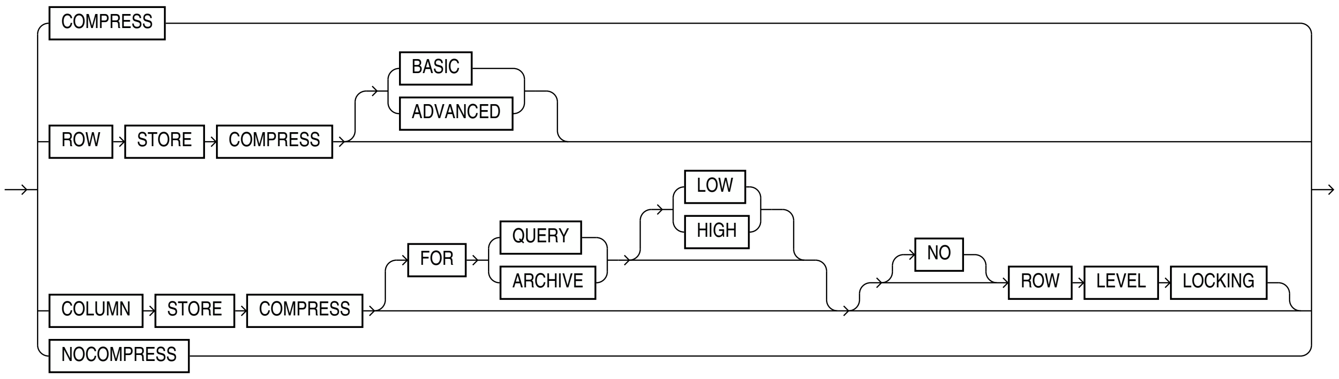


(*logging_clause*::=)

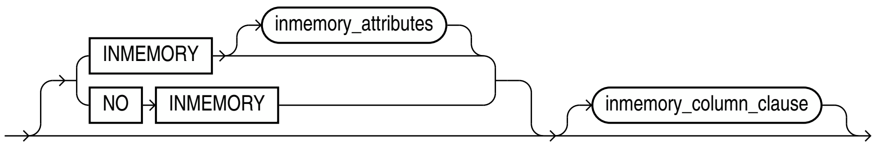
***logging_clause*::=**



***table_compression*::=**

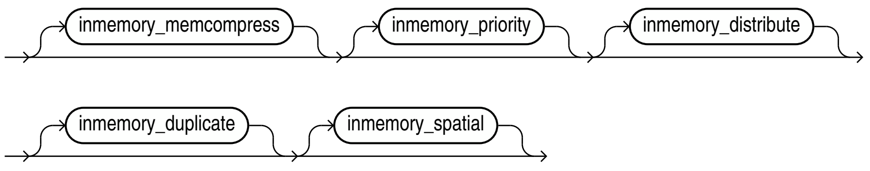


***inmemory_table_clause*::=**



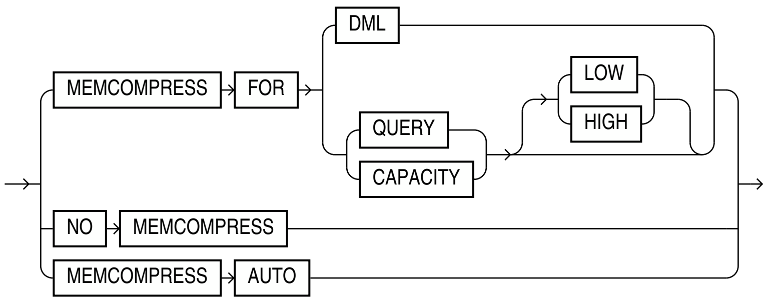
(*inmemory_attributes*::=, *inmemory_column_clause*::=)

inmemory_attributes::=

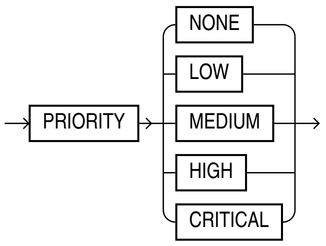


(*inmemory_memcompress*::=, *inmemory_priority*::=, *inmemory_distribute*::=, *inmemory_duplicate*::=)

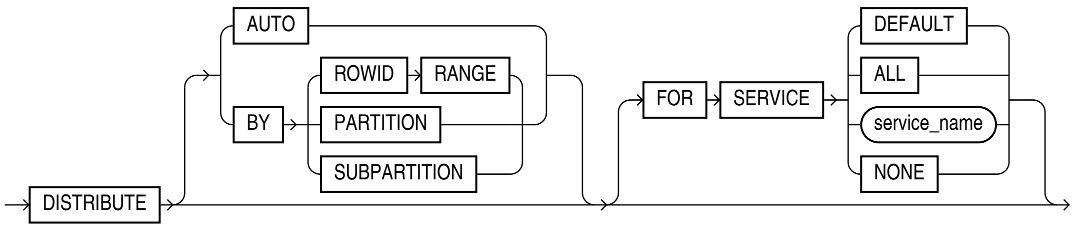
inmemory_memcompress::=



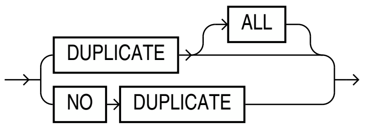
inmemory_priority::=



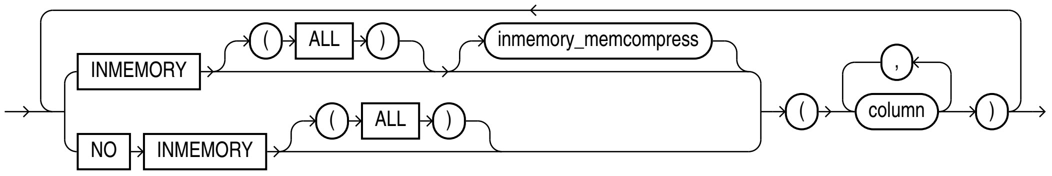
inmemory_distribute::=



***inmemory_duplicate*::=**

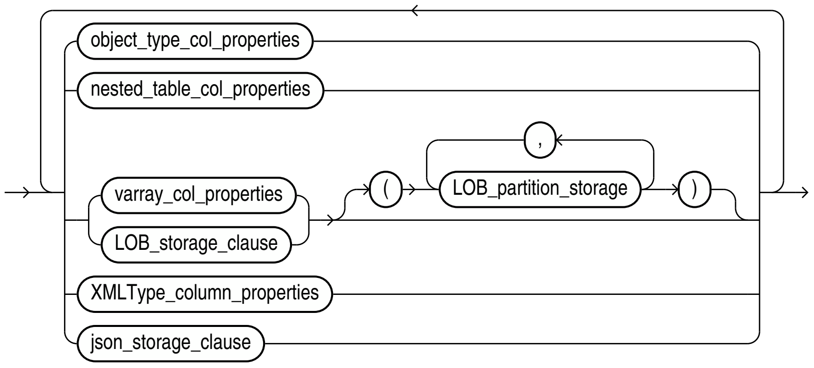


***inmemory_column_clause*::=**



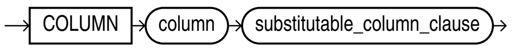
(*inmemory_memcompress*::=)

***column_properties*::=**



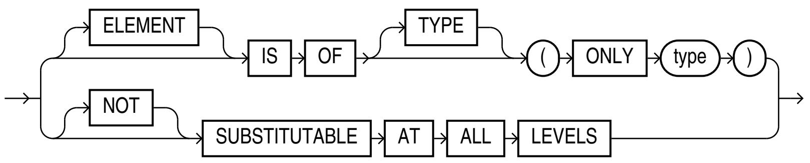
(*object_type_col_properties*::=, *nested_table_col_properties*::=, *varray_col_properties*::=, *LOB_partition_storage*::=, *LOB_storage_clause*::=, `XMLType_column_properties`: not supported for materialized views)

***object_type_col_properties*::=**

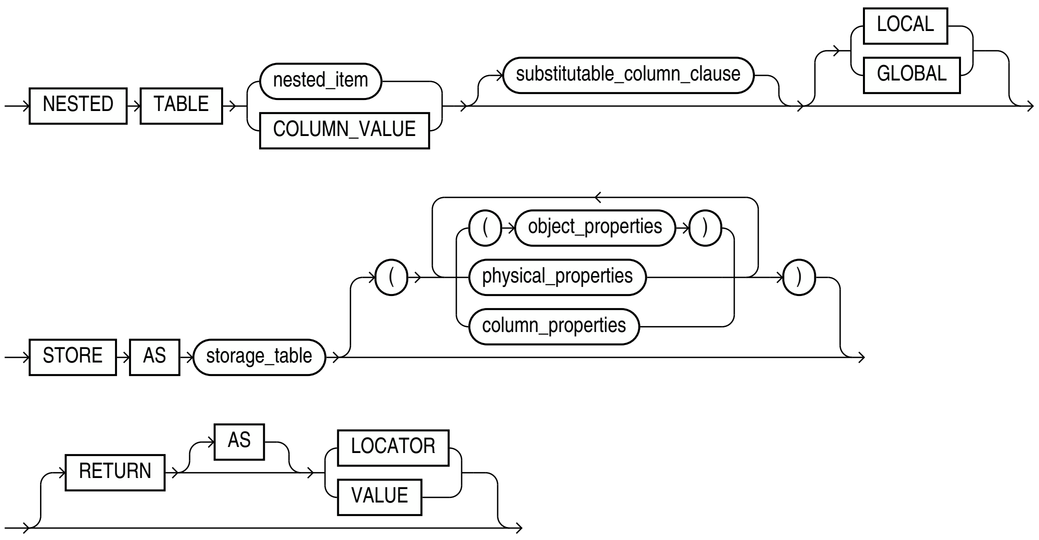


(*substitutable_column_clause*::=)

***substitutable_column_clause*::=**

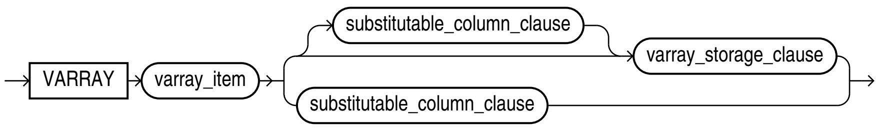


***nested_table_col_properties*::=**



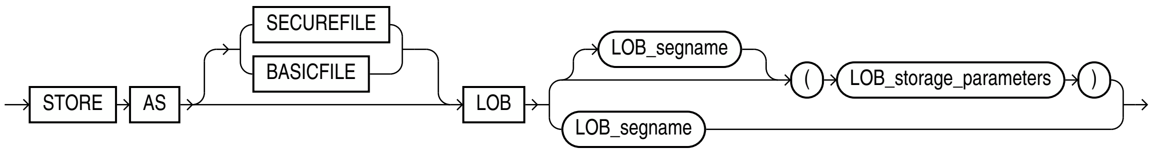
(*substitutable_column_clause*::=, *object_properties*::=, *physical_properties*::=—part of `CREATE TABLE` syntax, *column_properties*::=)

***varray_col_properties*::=**



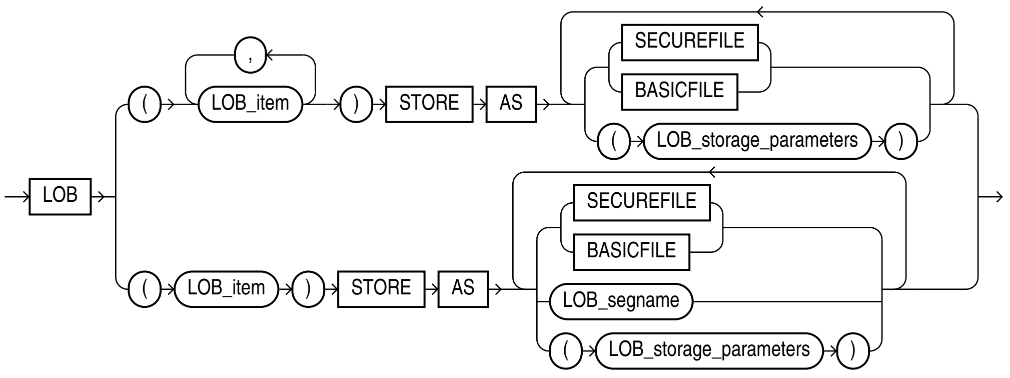
(*substitutable_column_clause*::=, *varray_storage_clause*::=)

***varray_storage_clause*::=**



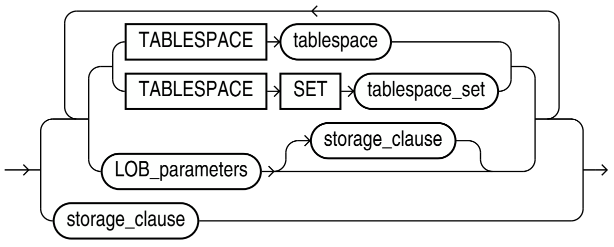
(*LOB_parameters::=*)

***LOB_storage_clause*::=**



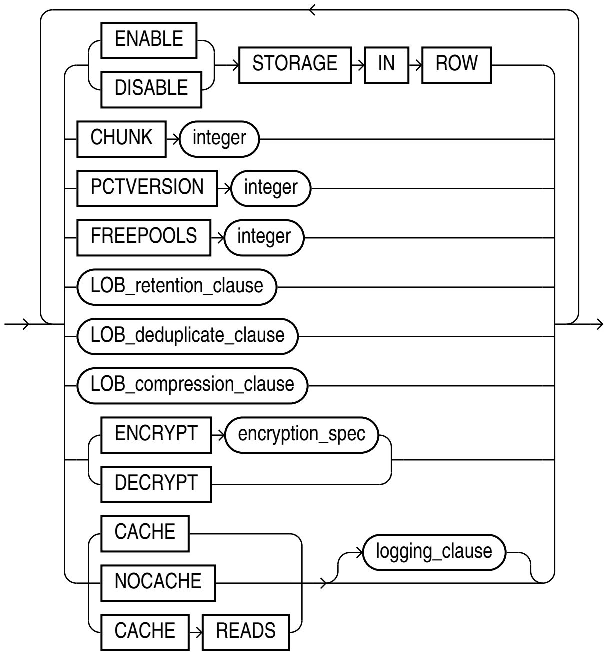
(*LOB_storage_parameters::=*)

***LOB_storage_parameters*::=**



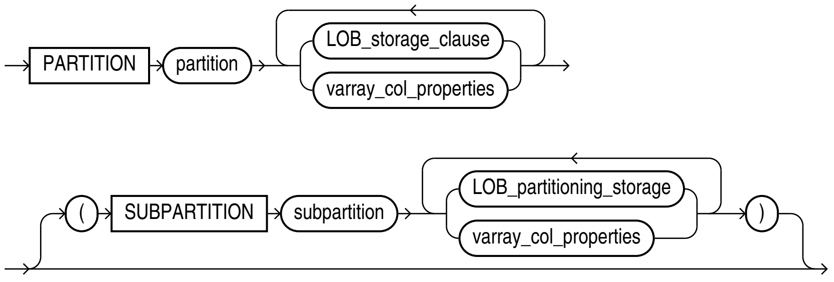
(`TABLESPACE SET`: not supported with `CREATE MATERIALIZED VIEW`, *LOB_parameters::=*, *storage_clause::=*)

***LOB_parameters*::=**



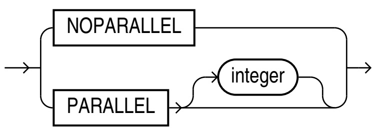
(*storage_clause*::=, *logging_clause*::=)

***LOB_partition_storage*::=**

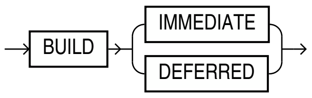


(*LOB_storage_clause*::=, *varray_col_properties*::=)

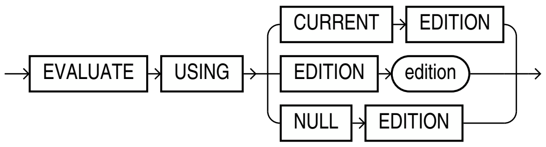
***parallel_clause*::=**



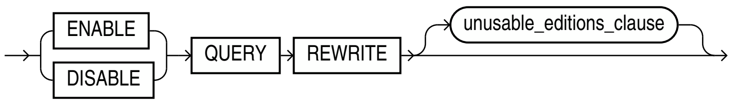
build_clause::=



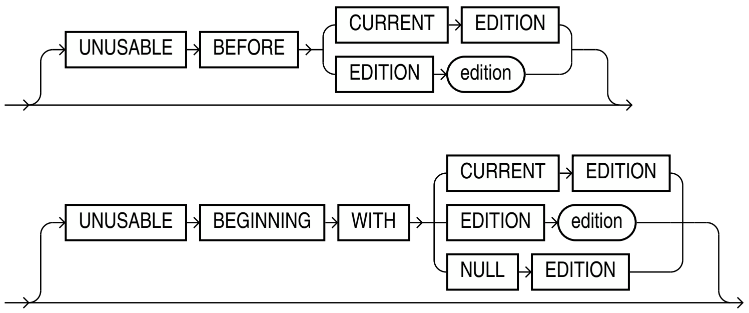
evaluation_edition_clause::=



query_rewrite_clause::=



unusable_editions_clause::=



Semantics

IF NOT EXISTS

Specifying `IF NOT EXISTS` has the following effects:

- If the view does not exist, a new view is created at the end of the statement.
- If the view exists, this is the view you have at the end of the statement. A new one is not created because the older one is detected.

Using `IF EXISTS` with `CREATE` results in `ORA-11543: Incorrect IF NOT EXISTS clause for CREATE statement`.

annotations_clause

For the full semantics of the annotations clause see *annotations_clause*.

schema

Specify the schema to contain the materialized view. If you omit *schema*, then Oracle Database creates the materialized view in your schema.

materialized_view

Specify the name of the materialized view to be created. The name must satisfy the requirements listed in "Database Object Naming Rules ". Oracle Database generates names for the table and indexes used to maintain the materialized view by adding a prefix or suffix to the materialized view name.

column_alias

You can specify a column alias for each column of the materialized view. The column alias list explicitly resolves any column name conflict, eliminating the need to specify aliases in the SELECT clause of the materialized view. If you specify any column alias in this clause, then you must specify an alias for each data source referenced in the SELECT clause.

ENCRYPT clause

Use this clause to encrypt this column of the materialized view. Refer to the CREATE TABLE clause *encryption_spec* for more information on column encryption.

OF *object_type*

The OF *object_type* clause lets you explicitly create an **object materialized view** of type *object_type*.

See Also:

See CREATE TABLE ... *object_table* for more information on the OF *type_name* clause

scoped_table_ref_constraint

Use the SCOPE FOR clause to restrict the scope of references to a single object table. You can refer either to the table name with *scope_table_name* or to a column alias. The values in the REF column or attribute point to objects in *scope_table_name* or *c_alias*, in which object instances of the same type as the REF column are stored. If you specify aliases, then they must have a one-to-one correspondence with the columns in the SELECT list of the defining query of the materialized view.

See Also:

"SCOPE REF Constraints" for more information

DEFAULT COLLATION

Use this clause to specify the default collation for the materialized view. The default collation is used as the derived collation for all the character literals included in the defining query of the materialized view. The default collation is not used by the materialized view columns; the collations for the materialized view columns are derived from the materialized view's defining subquery. The `CREATE MATERIALIZED VIEW` statement fails with an error, or the materialized view is created in an invalid state, if any of its character columns is based on an expression in the defining subquery that has no derived collation.

For *`collation_name`*, specify a valid named collation or pseudo-collation.

If you omit this clause, then the default collation for the materialized view is set to the effective schema default collation of the schema containing the materialized view. Refer to the DEFAULT_COLLATION clause of `ALTER SESSION` for more information on the effective schema default collation.

You can specify the `DEFAULT COLLATION` clause only if the `COMPATIBLE` initialization parameter is set to `12.2` or greater, and the `MAX_STRING_SIZE` initialization parameter is set to `EXTENDED`.

To change the default collation for a materialized view, you must recreate the materialized view.

Restrictions on the Default Collation for Materialized Views

The following restrictions apply when specifying the default collation for a materialized view:

- If the defining query of the materialized view contains the `WITH` *`plsql_declarations`* clause, then the default collation of the materialized view must be `USING_NLS_COMP`.
- If the materialized view is created on a prebuilt table, then the declared collations of the table columns must be the same as the corresponding collations of the materialized view columns, as derived from the defining query.

ON PREBUILT TABLE Clause

The `ON PREBUILT TABLE` clause lets you register an existing table as a preinitialized materialized view. This clause is particularly useful for registering large materialized views in a data warehousing environment. The table must have the same name and be in the same schema as the resulting materialized view.

If the materialized view is dropped, then the preexisting table reverts to its identity as a table.

Note:

This clause assumes that the table object reflects the materialization of a subquery. Oracle strongly recommends that you ensure that this assumption is true in order to ensure that the materialized view correctly reflects the data in its master tables.

The `ON PREBUILT TABLE` clause could be useful in the following scenarios:

- You have a table representing the result of a query. Creating the table was an expensive operation that possibly took a long time. You want to create a materialized view on the query. You can use the `ON PREBUILT TABLE` clause to avoid the expense of executing the query and populating the container for the materialized view.

- You temporarily discontinue having a materialized view, but keep its container table, using the `DROP MATERIALIZED VIEW` ... `PRESERVE TABLE` statement. You then decide to recreate the materialized view and you know that the master tables of the view have not changed. You can create the materialized view using the `ON PREBUILT TABLE` clause. This avoids the expense and time of creating and populating the container table for the materialized view.

If you specify `ON PREBUILT TABLE`, then Oracle database does not create the `I_SNAP$` index. This index improves fast refresh performance. If you want the benefits of this index, then you can create it manually. Refer to *Oracle Database Data Warehousing Guide* for more information.

WITH REDUCED PRECISION

Specify `WITH REDUCED PRECISION` to authorize the loss of precision that will result if the precision of the table or materialized view columns do not exactly match the precision returned by *`subquery`*.

WITHOUT REDUCED PRECISION

Specify `WITHOUT REDUCED PRECISION` to require that the precision of the table or materialized view columns match exactly the precision returned by *`subquery`*, or the create operation will fail. This is the default.

Restrictions on Using Prebuilt Tables

Prebuilt tables are subject to the following restrictions:

- Each column alias in *`subquery`* must correspond to a column in the prebuilt table, and corresponding columns must have matching data types.
- If you specify this clause, then you cannot specify a `NOT NULL` constraint for any column that is not referenced in *`subquery`* unless you also specify a default value for that column.
- You cannot specify the `ON PREBUILT TABLE` clause when creating a rowid materialized view.

See Also:

"Creating Prebuilt Materialized Views: Example"

physical_properties_clause

The components of the *`physical_properties_clause`* have the same semantics for materialized views that they have for tables, with exceptions and additions described in the sections that follow.

Restriction on the *physical_properties_clause*

You cannot specify `ORGANIZATION EXTERNAL` for a materialized view.

deferred_segment_creation

Use this clause to determine when the segment for this materialized view should be created. See the `CREATE TABLE` clause *deferred_segment_creation* for more information.

segment_attributes_clause

Use the *`segment_attributes_clause`* to establish values for the `PCTFREE`, `PCTUSED`, and `INITRANS` parameters, the storage characteristics for the materialized view, to assign a tablespace, and to specify whether logging is to occur. In the USING INDEX clause, you cannot specify PCTFREE or PCTUSED.

TABLESPACE Clause

Specify the tablespace in which the materialized view is to be created. If you omit this clause, then Oracle Database creates the materialized view in the default tablespace of the schema containing the materialized view.

See Also:

physical_attributes_clause and storage_clause for a complete description of these clauses, including default values

logging_clause

Specify LOGGING or NOLOGGING to establish the logging characteristics for the materialized view. The logging characteristic affects the creation of the materialized view and any nonatomic refresh that is initiated by way of the DBMS_REFRESH package. The default is the logging characteristic of the tablespace in which the materialized view resides.

See Also:

logging_clause for a full description of this clause and *Oracle Database PL/SQL Packages and Types Reference* for more information on atomic and nonatomic refresh

table_compression

Use the *table_compression* clause to instruct the database whether to compress data segments to reduce disk and memory use. This clause has the same semantics in CREATE MATERIALIZED VIEW and CREATE TABLE. Refer to the *table_compression* clause in the documentation on CREATE TABLE for the full semantics of this clause.

inmemory_table_clause

Use the *inmemory_table_clause* to enable or disable the materialized view for the In-Memory Column Store (IM column store). This clause has the same semantics as the *inmemory_table_clause* in the CREATE TABLE documentation.

inmemory_column_clause

Use the *inmemory_column_clause* to disable specific materialized view columns for the IM column store, and to specify the data compression method for specific columns. This clause has the same semantics here as it has for the *inmemory_column_clause* in the CREATE TABLE documentation, with the following addition: If you specify the *inmemory_column_clause*, then you must also specify a *column_alias* for each column in the materialized view.

index_org_table_clause

The ORGANIZATION INDEX clause lets you create an index-organized materialized view. In such a materialized view, data rows are stored in an index defined on the primary key of the materialized view. You can specify index organization for the following types of materialized views:

- Read-only and updatable object materialized views. You must ensure that the master table has a primary key.
- Read-only and updatable primary key materialized views.
- Read-only rowid materialized views.

The keywords and parameters of the *`index_org_table_clause`* have the same semantics as described in `CREATE TABLE`, with the restrictions that follow.

See Also:

The *index_org_table_clause* of `CREATE TABLE`

Restrictions on Index-Organized Materialized Views

Index-organized materialized views are subject to the following restrictions:

- You cannot specify the following `CREATE MATERIALIZED VIEW` clauses: `CACHE` or `NOCACHE`, `CLUSTER`, or `ON PREBUILT TABLE`.
- In the *`index_org_table_clause`*:
  - You cannot specify the *`mapping_table_clause`*.
  - You can specify `COMPRESS` only for a materialized view based on a composite primary key. You can specify `NOCOMPRESS` for a materialized view based on either a simple or composite primary key.

CLUSTER Clause

The `CLUSTER` clause lets you create the materialized view as part of the specified cluster. A cluster materialized view uses the space allocation of the cluster. Therefore, you do not specify physical attributes or the `TABLESPACE` clause with the `CLUSTER` clause.

Restriction on Cluster Materialized Views

If you specify `CLUSTER`, then you cannot specify the *`table_partitioning_clauses`* in *`materialized_view_props`*.

materialized_view_props

Use these property clauses to describe a materialized view that is not based on an existing table. To create a materialized view that is based on an existing table, use the `ON PREBUILT TABLE` clause.

column_properties

The *`column_properties`* clause lets you specify the storage characteristics of a LOB, nested table, varray, or `XMLType` column. The *`object_type_col_properties`* are not relevant for a materialized view.

See Also:

CREATE TABLE for detailed information about specifying the parameters of this clause

table_partitioning_clauses

The `table_partitioning_clauses` let you specify that the materialized view is partitioned on specified ranges of values or on a hash function. Partitioning of materialized views is the same as partitioning of tables.

See Also:

table_partitioning_clauses in the `CREATE TABLE` documentation

CACHE | NOCACHE

For data that will be accessed frequently, `CACHE` specifies that the blocks retrieved for this table are placed at the most recently used end of the least recently used (LRU) list in the buffer cache when a full table scan is performed. This attribute is useful for small lookup tables. `NOCACHE` specifies that the blocks are placed at the least recently used end of the LRU list.

Note:

`NOCACHE` has no effect on materialized views for which you specify `KEEP` in the `storage_clause`.

See Also:

CREATE TABLE for information about specifying `CACHE` or `NOCACHE`

parallel_clause

The `parallel_clause` lets you indicate whether parallel operations will be supported for the materialized view and sets the default degree of parallelism for queries and DML on the materialized view after creation.

For complete information on this clause, refer to *parallel_clause* in the documentation on `CREATE TABLE`.

build_clause

The `build_clause` lets you specify when to populate the materialized view.

IMMEDIATE

Specify `IMMEDIATE` to indicate that the materialized view is to be populated immediately. This is the default.

DEFERRED

Specify `DEFERRED` to indicate that the materialized view is to be populated by the next `REFRESH` operation. The first (deferred) refresh must always be a complete refresh. Until then, the materialized view has a staleness value of `UNUSABLE`, so it cannot be used for query rewrite.

USING INDEX Clause

The `USING INDEX` clause lets you establish the value of the `INITRANS` and `STORAGE` parameters for the default index Oracle Database uses to maintain the materialized view data. If `USING INDEX` is not specified, then default values are used for the index. Oracle Database uses the default index to speed up incremental (`FAST`) refresh of the materialized view.

Restriction on USING INDEX clause

You cannot specify the `PCTUSED` parameter in this clause.

USING NO INDEX Clause

Specify `USING NO INDEX` to suppress the creation of the default index. You can create an alternative index explicitly by using the `CREATE INDEX` statement. You should create such an index if you specify `USING NO INDEX` and you are creating the materialized view with the fast refresh method (`REFRESH FAST`).

create_mv_refresh

Use the *`create_mv_refresh`* clause to specify the default methods, modes, and times for the database to refresh the materialized view. If the master tables of a materialized view are modified, then the data in the materialized view must be updated to make the materialized view accurately reflect the data currently in its master tables. This clause lets you schedule the times and specify the method and mode for the database to refresh the materialized view.

Restriction on Synchronous Refresh

If you are using the synchronous refresh method, then you must specify `ON DEMAND` and `USING TRUSTED CONSTRAINTS`.

Note:

This clause only sets the default refresh options. For instructions on actually implementing the refresh, refer to *Oracle Database Administrator’s Guide* and *Oracle Database Data Warehousing Guide*.

See Also:

- "Periodic Refresh of Materialized Views: Example" and "Automatic Refresh Times for Materialized Views: Example"
- *Oracle Database PL/SQL Packages and Types Reference* for more information on refresh methods
- *Oracle Database Data Warehousing Guide* to learn how to use refresh statistics to monitor the performance of materialized view refresh operations

FAST Clause

Specify `FAST` to indicate the fast refresh method, which performs the refresh according to the changes that have occurred to the master tables. The changes for conventional DML changes are stored in the materialized view log associated with the master table. The changes for direct-path `INSERT` operations are stored in the direct loader log.

If you specify `REFRESH FAST`, then the `CREATE` statement will fail unless materialized view logs already exist for the materialized view master tables. Oracle Database creates the direct loader log automatically when a direct-path `INSERT` takes place. No user intervention is needed.

For both conventional DML changes and for direct-path `INSERT` operations, other conditions may restrict the eligibility of a materialized view for fast refresh.

Restrictions on FAST Refresh

`FAST` refresh is subject to the following restrictions:

- When you specify `FAST` refresh at create time, Oracle Database verifies that the materialized view you are creating is eligible for fast refresh. When you change the refresh method to `FAST` in an `ALTER MATERIALIZED VIEW` statement, Oracle Database does not perform this verification. If the materialized view is not eligible for fast refresh, then Oracle Database returns an error when you attempt to refresh this view.
- Materialized views are not eligible for fast refresh if the defining query contains an analytic function or the `XMLTable` function.
- Materialized views are not eligible for fast refresh if the defining query references a table on which an `XMLIndex` index is defined.
- You cannot fast refresh a materialized view if any of its columns is encrypted.

See Also:

- *Oracle Database Administrator's Guide* for restrictions on fast refresh in replication environments
- *Oracle Database Data Warehousing Guide* for restrictions on fast refresh in data warehousing environments
- The `EXPLAIN_MVIEW` procedure of the `DBMS_MVIEW` package for help diagnosing problems with fast refresh and the `TUNE_MVIEW` procedure of the `DBMS_MVIEW` package for correction of query rewrite problems
- "Analytic Functions "
- "Creating a Fast Refreshable Materialized View: Example"

COMPLETE Clause

Specify `COMPLETE` to indicate the complete refresh method, which is implemented by executing the defining query of the materialized view. If you request a complete refresh, then Oracle Database performs a complete refresh even if a fast refresh is possible.

FORCE Clause

Specify `FORCE` to indicate that when a refresh occurs, Oracle Database will perform a fast refresh if one is possible or a complete refresh if fast refresh is not possible. If you do not specify a refresh method (`FAST`, `COMPLETE`, or `FORCE`), then `FORCE` is the default.

ON COMMIT Clause

Specify ON COMMIT to indicate that a refresh is to occur whenever the database commits a transaction that operates on a master table of the materialized view. This clause may increase the time taken to complete the commit, because the database performs the refresh operation as part of the commit process.

You can specify only one of the ON COMMIT, ON DEMAND, and ON STATEMENT clauses. If you specify ON COMMIT, then you cannot also specify START WITH or NEXT.

Restrictions on Refreshing ON COMMIT

The following restrictions apply to the ON COMMIT clause:

- This clause is not supported for materialized views containing object types or Oracle-supplied types.
- This clause is not supported for materialized views with remote tables.
- If you specify this clause, then you cannot subsequently execute a distributed transaction on any master table of this materialized view. For example, you cannot insert into the master by selecting from a remote table. The ON DEMAND clause does not impose this restriction on subsequent distributed transactions on master tables.

ON DEMAND Clause

Specify ON DEMAND to indicate that database will not refresh the materialized view unless the user manually launches a refresh through one of the three DBMS_MVIEW refresh procedures.

You can specify only one of the ON COMMIT, ON DEMAND, and ON STATEMENT clauses. If you omit all three of these clauses, then ON DEMAND is the default. You can override this default setting by specifying the START WITH or NEXT clauses, either in the same CREATE MATERIALIZED VIEW statement or a subsequent ALTER MATERIALIZED VIEW statement.

START WITH and NEXT take precedence over ON DEMAND. Therefore, in most circumstances it is not meaningful to specify ON DEMAND when you have specified START WITH or NEXT.

See Also:

- *Oracle Database PL/SQL Packages and Types Reference* for information on these procedures
- *Oracle Database Data Warehousing Guide* on the types of materialized views you can create by specifying REFRESH ON DEMAND

ON STATEMENT Clause

Specify ON STATEMENT to indicate that an automatic refresh is to occur every time a DML operation is performed on any of the materialized view's base tables.

You can specify only one of the ON COMMIT, ON DEMAND, and ON STATEMENT clauses. You can specify ON STATEMENT only when *creating* a materialized view. You cannot subsequently *alter* the materialized view to use ON STATEMENT refresh.

Restrictions on Refreshing ON STATEMENT

The following restrictions apply to the ON STATEMENT clause:

- This clause can be used only with materialized views that are fast refreshable. The `ON STATEMENT` clause must be specified with the `REFRESH FAST` clause.
- The base tables referenced in the materialized view's defining query must be connected in a join graph that uses the star schema or snowflake schema model. The query must contain exactly one centralized fact table and one or more dimension tables, with all pairs of joined tables being related using primary key-foreign key constraints.
  - There is no restriction on the depth of the snowflake model.
  - The constraints can be in `RELY` mode. However, you must include the `USING TRUSTED CONSTRAINT` clause while creating the materialized view to use the `RELY` constraint.
- The materialized view's defining query must include the `ROWID` column of the fact table.
- The materialized view's defining query cannot include any of the following: invisible columns, ANSI join syntax, complex query, inline view as base table, composite primary key, `LONG` columns, and LOB columns.
- You cannot alter the definition of an existing materialized view that uses the `ON STATEMENT` refresh mode.
- You cannot alter an existing materialized view and enable `ON STATEMENT` refresh for it.
- The following operations cause a materialized view with `ON STATEMENT` refresh to become unusable:
  - `UPDATE` operations on one or more dimension tables on which the materialized view is based
  - Partition maintenance operations and `TRUNCATE` operations on any base table

    However, a materialized view with the `ON STATEMENT` refresh mode can be partitioned.
- All the restrictions that apply to the `ON COMMIT` clause apply to `ON STATEMENT`.

START WITH Clause

Specify a datetime expression for the first automatic refresh time.

NEXT Clause

Specify a datetime expression for calculating the interval between automatic refreshes.

Both the `START WITH` and `NEXT` values must evaluate to a time in the future. If you omit the `START WITH` value, then the database determines the first automatic refresh time by evaluating the `NEXT` expression with respect to the creation time of the materialized view. If you specify a `START WITH` value but omit the `NEXT` value, then the database refreshes the materialized view only once. If you omit both the `START WITH` and `NEXT` values, or if you omit the *`create_mv_refresh`* entirely, then the database does not automatically refresh the materialized view.

WITH PRIMARY KEY Clause

Specify `WITH PRIMARY KEY` to create a primary key materialized view. This is the default and should be used in all cases except those described for `WITH ROWID`. Primary key materialized views allow materialized view master tables to be reorganized without affecting the eligibility of the materialized view for fast refresh. The master table must contain an enabled primary key constraint, and the defining query of the materialized view must specify all of the primary key columns directly. In the defining query, the primary key columns cannot be specified as the argument to a function such as `UPPER`.

Restriction on Primary Key Materialized Views

You cannot specify this clause for an object materialized view. Oracle Database implicitly refreshes objects materialized WITH OBJECT ID.

See Also:

Oracle Database Administrator's Guide for detailed information about primary key materialized views and "Creating Primary Key Materialized Views: Example"

WITH ROWID Clause

Specify WITH ROWID to create a rowid materialized view. Rowid materialized views are useful if the materialized view does not include all primary key columns of the master tables. Rowid materialized views must be based on a single table and cannot contain any of the following:

- Distinct or aggregate functions
- GROUP BY or CONNECT BY clauses
- Subqueries
- Joins
- Set operations

The WITH ROWID clause has no effect if there are multiple master tables in the defining query.

Rowid materialized views are not eligible for fast refresh after a master table reorganization until a complete refresh has been performed.

Restriction on Rowid Materialized Views

You cannot specify this clause for an object materialized view. Oracle Database implicitly refreshes objects materialized WITH OBJECT ID.

See Also:

"Creating Materialized Aggregate Views: Example" and "Creating Rowid Materialized Views: Example"

USING ROLLBACK SEGMENT Clause

This clause is not valid if your database is in automatic undo mode, because in that mode Oracle Database uses undo tablespaces instead of rollback segments. Oracle strongly recommends that you use automatic undo mode. This clause is supported for backward compatibility with replication environments containing older versions of Oracle Database that still use rollback segments.

For *rollback_segment*, specify the remote rollback segment to be used during materialized view refresh.

DEFAULT

DEFAULT specifies that Oracle Database will choose automatically which rollback segment to use. If you specify DEFAULT, then you cannot specify *rollback_segment*. DEFAULT is most useful when modifying, rather than creating, a materialized view.

See Also:

ALTER MATERIALIZED VIEW

MASTER

`MASTER` specifies the remote rollback segment to be used at the remote master site for the individual materialized view.

LOCAL

`LOCAL` specifies the remote rollback segment to be used for the local refresh group that contains the materialized view. This is the default.

See Also:

Oracle Database PL/SQL Packages and Types Reference for information on specifying the local materialized view rollback segment using the `DBMS_REFRESH` package

If you omit *`rollback_segment`*, then the database automatically chooses the rollback segment to be used. One master rollback segment is stored for each materialized view and is validated during materialized view creation and refresh. If the materialized view is complex, then the database ignores any master rollback segment you specify.

USING ... CONSTRAINTS Clause

The `USING` ... `CONSTRAINTS` clause lets Oracle Database choose more rewrite options during the refresh operation, resulting in more efficient refresh execution. The clause lets Oracle Database use unenforced constraints, such as dimension relationships or constraints in the `RELY` state, rather than relying only on enforced constraints during the refresh operation.

The `USING TRUSTED CONSTRAINTS` clause enables you to create a materialized view on top of a table that has a non-NULL Virtual Private Database (VPD) policy on it. In this case, you must ensure that the materialized view behaves correctly. Materialized view results are computed based on the rows and columns filtered by VPD policy. Therefore, you must coordinate the materialized view definition with the VPD policy to ensure the correct results. Without the `USING TRUSTED CONSTRAINTS` clause, any VPD policy on a master table will prevent a materialized view from being created.

Note:

The `USING TRUSTED CONSTRAINTS` clause lets Oracle Database use dimension and constraint information that has been declared trustworthy by the database administrator but that has not been validated by the database. If the dimension and constraint information is valid, then performance may improve. However, if this information is invalid, then the refresh procedure may corrupt the materialized view even though it returns a success status.

If you omit this clause, then the default is `USING ENFORCED CONSTRAINTS`.

NEVER REFRESH Clause

Specify `NEVER REFRESH` to prevent the materialized view from being refreshed with any Oracle Database refresh mechanism or packaged procedure. Oracle Database will ignore any `REFRESH` statement on the materialized view issued from such a procedure. If you specify this clause, then you can perform DML operations on the materialized view. To reverse this clause, you must issue an `ALTER MATERIALIZED VIEW` ... `REFRESH` statement.

evaluation_edition_clause

You must specify this clause if *`subquery`* references an editioned object. Use this clause to specify the edition that is searched during name resolution of the editioned object—the evaluation edition.

- Specify `CURRENT EDITION` to search the edition in which this DDL statement is executed.
- Specify `EDITION` *`edition`* to search *`edition`*.
- Specifying `NULL EDITION` is equivalent to omitting the *`evaluation_edition_clause`*.

If you omit the *`evaluation_edition_clause`*, then editioned objects are invisible during name resolution and an error will result. Dropping the evaluation edition invalidates the materialized view.

> **See Also:**
>
> *Oracle Database Development Guide* for more information on specifying the evaluation edition for a materialized view

{ ENABLE | DISABLE } ON QUERY COMPUTATION

This clause lets you create a real-time materialized view or a regular view. A real-time materialized view provides fresh data to user queries even when the materialized view is not in sync with its base tables due to data changes. Instead of modifying the materialized view, the optimizer writes a query that combines the existing rows in the materialized view with changes recorded in log files (either materialized view logs or the direct loader logs). This is called on-query computation.

- Specify `ENABLE ON QUERY COMPUTATION` to create a real-time materialized view by enabling on-query computation. This allows you to directly query up-to-date data from the materialized view by specifying the `FRESH_MV` hint in the `SELECT` statement. If the materialized view is also enabled for query rewrite, then on-query computation occurs automatically during query rewrite.
- Specify `DISABLE ON QUERY COMPUTATION` to create a regular materialized view by disabling on-query computation. This is the default.

Restrictions on Real-Time Materialized Views

Real-time materialized views are subject to the following restrictions:

- Real-time materialized views cannot be used when one or more materialized view logs created on the base tables are either unusable or nonexistent.
- A real-time materialized view must be refreshable using out-of-place refresh, log-based refresh, or partition change tracking (PCT) refresh.
- A refresh-on-commit materialized view cannot be a real-time materialized view.

- If a real-time materialized view is a nested materialized view that is defined on top of one or more base materialized views, then query rewrite occurs only if all the base materialized views are fresh. If one or more base materialized views are stale, then query rewrite is not performed using this real-time materialized view.
- The cursors of queries that directly access real-time materialized views are not shared.

See Also:

- FRESH_MV Hint
- *Oracle Database Data Warehousing Guide* for more information on real-time materialized views

query_rewrite_clause

The *`query_rewrite_clause`* lets you specify whether the materialized view is eligible to be used for query rewrite.

ENABLE Clause

Specify `ENABLE` to enable the materialized view for query rewrite. If you also specify the *`unusable_editions_clause`*, then the materialized view is not enabled for query rewrite in the unusable editions.

Restrictions on Enabling Query Rewrite

Enabling of query rewrite is subject to the following restrictions:

- You can enable query rewrite only if all user-defined functions in the materialized view are `DETERMINISTIC`.
- You can enable query rewrite only if expressions in the statement are repeatable. For example, you cannot include `CURRENT_TIME` or `USER`, sequence values (such as the `CURRVAL` or `NEXTVAL` pseudocolumns), or the `SAMPLE` clause (which may sample different rows as the contents of the materialized view change).

Note:

- Query rewrite is disabled by default, so you must specify this clause to make materialized views eligible for query rewrite.
- After you create the materialized view, you must collect statistics on it using the `DBMS_STATS` package. Oracle Database needs the statistics generated by this package to optimize query rewrite.

See Also:

- *Oracle Database Data Warehousing Guide* for more information on query rewrite
- *Oracle Database Data Warehousing Guide* to learn how to use refresh statistics to monitor the performance of materialized view refresh operations
- *Oracle Database PL/SQL Packages and Types Reference* for information about the `DBMS_STATS` package
- The `EXPLAIN_MVIEW` procedure of the `DBMS_MVIEW` package for help diagnosing problems with query rewrite and the `TUNE_MVIEW` procedure of the `DBMS_MVIEW` package for correction of query rewrite problems
- CREATE FUNCTION

DISABLE Clause

Specify `DISABLE` to indicate that the materialized view is not eligible for use by query rewrite. A disabled materialized view can be refreshed.

unusable_editions_clause

This clause lets you specify that the materialized view is not eligible for query rewrite in one or more editions. You can specify this clause regardless of whether you specify the `ENABLE` or `DISABLE` clause. If you specify the `DISABLE` clause, then this clause will take effect if the materialized view is subsequently enabled for query rewrite using the `ALTER MATERIALIZED VIEW` ... `ENABLE QUERY REWRITE` statement.

UNUSABLE BEFORE Clause

This clause lets you specify that the materialized view is not eligible for query rewrite in the ancestors of an edition.

- If you specify `CURRENT EDITION`, then the materialized view is not eligible for query rewrite in the ancestors of the current edition.
- If you specify `EDITION` *`edition`*, then the materialized view is not eligible for query rewrite in the ancestors of the specified *`edition`*.

UNUSABLE BEGINNING WITH Clause

This clause lets you specify that the materialized view is not eligible for query rewrite in an edition and its descendants.

- If you specify `CURRENT EDITION`, then the materialized view is not eligible for query rewrite in the current edition and its descendants.
- If you specify `EDITION` *`edition`*, then the materialized view is not eligible for query rewrite in the specified edition and its descendants.
- Specifying `NULL EDITION` is equivalent to omitting the `UNUSABLE BEGINNING WITH` clause.

The materialized view has a dependency on each edition in which it is not eligible for query rewrite. If such an edition is subsequently dropped, then the dependency is removed. However, the materialized view is not invalidated.

ENABLE | DISABLE CONCURRENT REFRESH

Enable concurrent refresh to refresh the same on-commit atomic materialized view across multiple sessions concurrently. The materialized view is refreshed concurrently when multiple concurrent DML transactions on the same base table of the materialized view are committed.

There are no limitations on how many materialized views can be refreshed.

Concurrent refresh is disabled by default. You must explictly enable it on the materialized view. Note that you can enable it only on on-commit materialized views.

See Also:

Refreshing Materialized Views of the *Oracle Database Data Warehousing Guide*.

AS *subquery*

Specify the defining query of the materialized view. When you create the materialized view, Oracle Database executes this subquery and places the results in the materialized view. This subquery is any valid SQL subquery. However, not all subqueries are fast refreshable, nor are all subqueries eligible for query rewrite.

Notes on the Defining Query of a Materialized View

The following notes apply to materialized views:

- Oracle Database does not execute the defining query immediately if you specify `BUILD DEFERRED`.
- Oracle recommends that you qualify each table and view in the `FROM` clause of the defining query of the materialized view with the schema containing it.
- In order to create a materialized view whose defining query selects from a master table that has a Virtual Private Database (VPD) policy, you must specify the `REFRESH USING TRUSTED CONSTRAINTS` clause.

Restrictions on the Defining Query of a Materialized View

The materialized view query is subject to the following restrictions:

- The defining query of a materialized view can select from tables, views, or materialized views owned by the user `SYS`, but you cannot enable `QUERY REWRITE` on such a materialized view.
- The defining query of a materialized view cannot select from a `V$` view or a `GV$` view.
- You cannot define a materialized view with a subquery in the select list of the defining query. You can, however, include subqueries elsewhere in the defining query, such as in the `WHERE` clause.
- You cannot use the `AS OF` clause of the *`flashback_query_clause`* in the defining query of a materialized view.
- Materialized join views and materialized aggregate views with a `GROUP BY` clause cannot select from an index-organized table.
- Materialized views cannot contain columns of data type `LONG` or `LONG RAW`.
- Materialized views cannot contain virtual columns.

- You cannot create a materialized view log on a temporary table. Therefore, if the defining query references a temporary table, then this materialized view will not be eligible for `FAST` refresh, nor can you specify the `QUERY REWRITE` clause in this statement.
- If the `FROM` clause of the defining query references another materialized view, then you must always refresh the materialized view referenced in the defining query before refreshing the materialized view you are creating in this statement.
- Materialized views with join expressions in the defining query cannot have XML data type columns. The XML data types include `XMLType` and URI data type columns.

If you are creating a materialized view enabled for query rewrite, then:

- The defining query cannot contain, either directly or through a view, references to `ROWNUM`, `USER`, `SYSDATE`, remote tables, sequences, or PL/SQL functions that write or read database or package state.
- Neither the materialized view nor the master tables of the materialized view can be remote.

If you want the materialized view to be eligible for fast refresh using a materialized view log, or synchronous refresh using a staging log, then some additional restrictions apply.

See Also:

- *Oracle Database Data Warehousing Guide* for restrictions relating to using fast refresh and synchronous refresh
- *Oracle Database Administrator's Guide*for more information on restrictions relating to replication
- "Creating Materialized Join Views: Example", "Creating Subquery Materialized Views: Example", and "Creating a Nested Materialized View: Example"

Examples

The following examples require the materialized logs that are created in the "Examples" section of CREATE MATERIALIZED VIEW LOG .

Creating a Simple Materialized View: Example

The following statement creates a very simple materialized view based on the `employees` and table in the `hr` schema:

```
CREATE MATERIALIZED VIEW mv1 AS SELECT * FROM hr.employees;
```

By default, Oracle Database creates a primary key materialized view with refresh on demand only. If a materialized view log exists on `employees`, then `mv1` can be altered to be capable of fast refresh. If no such log exists, then only full refresh of `mv1` is possible. Oracle Database uses default storage properties for `mv1`. The only privileges required for this operation are the `CREATE MATERIALIZED VIEW` system privilege, and the `READ` or `SELECT` object privilege on `hr.employees`.

Creating Subquery Materialized Views: Example

The following statement creates a subquery materialized view based on the `customers` and `countries` tables in the `sh` schema at the `remote` database:

```
CREATE MATERIALIZED VIEW foreign_customers
   AS SELECT * FROM sh.customers@remote cu
   WHERE EXISTS
     (SELECT * FROM sh.countries@remote co
      WHERE co.country_id = cu.country_id);
```

Creating Materialized Aggregate Views: Example

The following statement creates and populates a materialized aggregate view on the sample `sh.sales` table and specifies the default refresh method, mode, and time. It uses the materialized view log created in "Creating a Materialized View Log for Fast Refresh: Examples", as well as the two additional logs shown here:

```
CREATE MATERIALIZED VIEW LOG ON times
   WITH ROWID, SEQUENCE (time_id, calendar_year)
   INCLUDING NEW VALUES;

CREATE MATERIALIZED VIEW LOG ON products
   WITH ROWID, SEQUENCE (prod_id)
   INCLUDING NEW VALUES;

CREATE MATERIALIZED VIEW sales_mv
   BUILD IMMEDIATE
   REFRESH FAST ON COMMIT
   AS SELECT t.calendar_year, p.prod_id,
      SUM(s.amount_sold) AS sum_sales
      FROM times t, products p, sales s
      WHERE t.time_id = s.time_id AND p.prod_id = s.prod_id
      GROUP BY t.calendar_year, p.prod_id;
```

Creating Materialized Join Views: Example

The following statement creates and populates the materialized aggregate view `sales_by_month_by_state` using tables in the sample `sh` schema. The materialized view will be populated with data as soon as the statement executes successfully. By default, subsequent refreshes will be accomplished by reexecuting the defining query of the materialized view:

```
CREATE MATERIALIZED VIEW sales_by_month_by_state
     TABLESPACE example
     PARALLEL 4
     BUILD IMMEDIATE
     REFRESH COMPLETE
     ENABLE QUERY REWRITE
     AS SELECT t.calendar_month_desc, c.cust_state_province,
        SUM(s.amount_sold) AS sum_sales
        FROM times t, sales s, customers c
        WHERE s.time_id = t.time_id AND s.cust_id = c.cust_id
        GROUP BY t.calendar_month_desc, c.cust_state_province;
```

Creating Prebuilt Materialized Views: Example

The following statement creates a materialized aggregate view for the preexisting summary table, `sales_sum_table`:

```
CREATE TABLE sales_sum_table
   (month VARCHAR2(8), state VARCHAR2(40), sales NUMBER(10,2));

CREATE MATERIALIZED VIEW sales_sum_table
   ON PREBUILT TABLE WITH REDUCED PRECISION
   ENABLE QUERY REWRITE
   AS SELECT t.calendar_month_desc AS month,
             c.cust_state_province AS state,
             SUM(s.amount_sold) AS sales
```

```
      FROM times t, customers c, sales s
      WHERE s.time_id = t.time_id AND s.cust_id = c.cust_id
      GROUP BY t.calendar_month_desc, c.cust_state_province;
```

In the preceding example, the materialized view has the same name and also has the same number of columns with the same data types as the prebuilt table. The `WITH REDUCED PRECISION` clause allows for differences between the precision of the materialized view columns and the precision of the values returned by the subquery.

Creating Primary Key Materialized Views: Example

The following statement creates the primary key materialized view `catalog` on the sample table `oe.product_information`:

```
CREATE MATERIALIZED VIEW catalog
   REFRESH FAST START WITH SYSDATE NEXT SYSDATE + 1/4096
   WITH PRIMARY KEY
   AS SELECT * FROM product_information;
```

Creating Rowid Materialized Views: Example

The following statement creates a rowid materialized view on the sample table `oe.orders`:

```
CREATE MATERIALIZED VIEW order_data REFRESH WITH ROWID
   AS SELECT * FROM orders;
```

Periodic Refresh of Materialized Views: Example

The following statement creates the primary key materialized view `emp_data` and populates it with data from the sample table `hr.employees`:

```
CREATE MATERIALIZED VIEW LOG ON employees
   WITH PRIMARY KEY
   INCLUDING NEW VALUES;

CREATE MATERIALIZED VIEW emp_data
   PCTFREE 5 PCTUSED 60
   TABLESPACE example
   STORAGE (INITIAL 50K)
   REFRESH FAST NEXT sysdate + 7
   AS SELECT * FROM employees;
```

The preceding statement does not include a `START WITH` parameter, so Oracle Database determines the first automatic refresh time by evaluating the `NEXT` value using the current `SYSDATE`. A materialized view log was created for the employee table, so Oracle Database performs a fast refresh of the materialized view every 7 days, beginning 7 days after the materialized view is created.

Because the materialized view conforms to the conditions for fast refresh, the database will perform a fast refresh. The preceding statement also establishes storage characteristics that the database uses to maintain the materialized view.

Automatic Refresh Times for Materialized Views: Example

The following statement creates the complex materialized view `all_customers` that queries the employee tables on the `remote` and `local` databases:

```
CREATE MATERIALIZED VIEW all_customers
   PCTFREE 5 PCTUSED 60
   TABLESPACE example
   STORAGE (INITIAL 50K)
   USING INDEX STORAGE (INITIAL 25K)
   REFRESH START WITH ROUND(SYSDATE + 1) + 11/24
   NEXT NEXT_DAY(TRUNC(SYSDATE), 'MONDAY') + 15/24
   AS SELECT * FROM sh.customers@remote
         UNION
      SELECT * FROM sh.customers@local;
```

Oracle Database automatically refreshes this materialized view tomorrow at 11:00 a.m. and subsequently every Monday at 3:00 p.m. The default refresh method is `FORCE`. The defining query contains a `UNION` operator, which is not supported for fast refresh, so the database will automatically perform a complete refresh.

The preceding statement also establishes storage characteristics for both the materialized view and the index that the database uses to maintain it:

- The first `STORAGE` clause establishes the sizes of the first and second extents of the materialized view as 50 kilobytes each.
- The second `STORAGE` clause, appearing with the `USING INDEX` clause, establishes the sizes of the first and second extents of the index as 25 kilobytes each.

Creating a Fast Refreshable Materialized View: Example

The following statement creates a fast-refreshable materialized view that selects columns from the `order_items` table in the sample `oe` schema, using the `UNION` set operator to restrict the rows returned from the `product_information` and `inventories` tables using `WHERE` conditions. The materialized view logs for `order_items` and `product_information` were created in the "Examples " section of `CREATE MATERIALIZED VIEW LOG`. This example also requires a materialized view log on `oe.inventories`.

```
CREATE MATERIALIZED VIEW LOG ON inventories
   WITH (quantity_on_hand);

CREATE MATERIALIZED VIEW warranty_orders REFRESH FAST AS
  SELECT order_id, line_item_id, product_id FROM order_items o
    WHERE EXISTS
    (SELECT * FROM inventories i WHERE o.product_id = i.product_id
      AND i.quantity_on_hand IS NOT NULL)
  UNION
    SELECT order_id, line_item_id, product_id FROM order_items
    WHERE quantity > 5;
```

The materialized view `warranty_orders` requires that materialized view logs be defined on `order_items` (with `product_id` as a join column) and on inventories (with `quantity_on_hand` as a filter column). See "Specifying Filter Columns for Materialized View Logs: Example" and "Specifying Join Columns for Materialized View Logs: Example".

Creating a Nested Materialized View: Example

The following example uses the materialized view from the preceding example as a master table to create a materialized view tailored for a particular sales representative in the sample `oe` schema:

```
CREATE MATERIALIZED VIEW my_warranty_orders
   AS SELECT w.order_id, w.line_item_id, o.order_date
   FROM warranty_orders w, orders o
   WHERE o.order_id = o.order_id
   AND o.sales_rep_id = 165;
```

Specify Annotations at the View Level

The following example adds annotations `Title` value `Tab1 MV1` and `Snapshot` without a value to the materialized view `MView1`:

```
CREATE MATERIALIZED VIEW MView1 ANNOTATIONS (Title 'Tab1 MV1', ADD Snapshot) AS SELECT *
from Table1;
```

Specify Annotations at the View and Column Level

The following example adds `Hidden` to column `T`, `Title` with value `Tab1 MV1`, and `Snapshot` without a value to the materialized view `MView1` :

```
CREATE MATERIALIZED VIEW MView1(T ANNOTATIONS (Hidden)) ANNOTATIONS (Title 'Tab1 MV1',
ADD Snapshot)
   AS SELECT * from Table1;
```

CREATE MATERIALIZED VIEW LOG

Purpose

Use the `CREATE MATERIALIZED VIEW LOG` statement to create a **materialized view log**, which is a table associated with the master table of a materialized view.

Note:

The keyword `SNAPSHOT` is supported in place of `MATERIALIZED VIEW` for backward compatibility.

Materialized view logs are used for two types of materialized view refreshes: fast refresh and synchronous refresh.

Fast refresh uses a conventional materialized view log. During a fast refresh (also called an incremental refresh), when DML changes are made to master table data, Oracle Database stores rows describing those changes in the materialized view log and then uses the materialized view log to refresh materialized views based on the master table.

Synchronous refresh uses a special type of materialized view log called a **staging log**. During a synchronous refresh, DML changes are first described in the staging log and then applied to the master tables and the materialized views simultaneously. This guarantees that the master table data and materialized view data are in sync throughout the refresh process. This refresh method is useful in data warehousing environments.

Without a materialized view log, Oracle Database must reexecute the materialized view query to refresh the materialized view. This process is called a **complete refresh**. Usually, a complete refresh takes more time to complete than a fast refresh or a synchronous refresh.

A materialized view log is located in the master database in the same schema as the master table. A master table can have only one materialized view log defined on it.

To fast refresh or synchronous refresh a materialized join view, you must create a materialized view log for each of the tables referenced by the materialized view.

Fast refresh supports two types of materialized view logs: timestamp-based materialized view logs and commit SCN-based materialized view logs. Timestamp-based materialized view logs use timestamps and require some setup operations when preparing to refresh the materialized view. Commit SCN-based materialized view logs use commit SCN data rather than timestamps, which removes the need for the setup operations and thus can improve the speed of the materialized view refresh. If you specify the COMMIT SCN clause, then a commit SCN-based materialized view log is created. Otherwise, a time-stamp based materialized view log is created. Note that only new materialized view logs can take advantage of COMMIT SCN. Existing materialized view logs cannot be altered to add COMMIT SCN unless they are dropped and recreated. Refer to *Oracle Database Data Warehousing Guide* for more information.

Synchronous refresh supports only timestamp-based staging logs.

See Also:

- CREATE MATERIALIZED VIEW , ALTER MATERIALIZED VIEW , *Oracle Database Concepts*, *Oracle Database Data Warehousing Guide*, and *Oracle Database Administrator's Guide* for information on materialized views in general
- ALTER MATERIALIZED VIEW LOG for information on modifying a materialized view log
- DROP MATERIALIZED VIEW LOG for information on dropping a materialized view log
- *Oracle Database Utilities* for information on using direct loader logs

Prerequisites

The privileges required to create a materialized view log directly relate to the privileges necessary to create the underlying objects associated with a materialized view log.

- If you own the master table, then you can create an associated materialized view log if you have the CREATE TABLE privilege.
- If you are creating a materialized view log for a table in another user's schema, then you must have the CREATE ANY TABLE and COMMENT ANY TABLE system privileges, as well as either the READ or SELECT object privilege on the master table or the READ ANY TABLE or SELECT ANY TABLE system privilege.

In either case, the owner of the materialized view log must have sufficient quota in the tablespace intended to hold the materialized view log or must have the UNLIMITED TABLESPACE system privilege.

See Also:

Oracle Database Data Warehousing Guide for more information about the prerequisites for creating a materialized view log

Restrictions

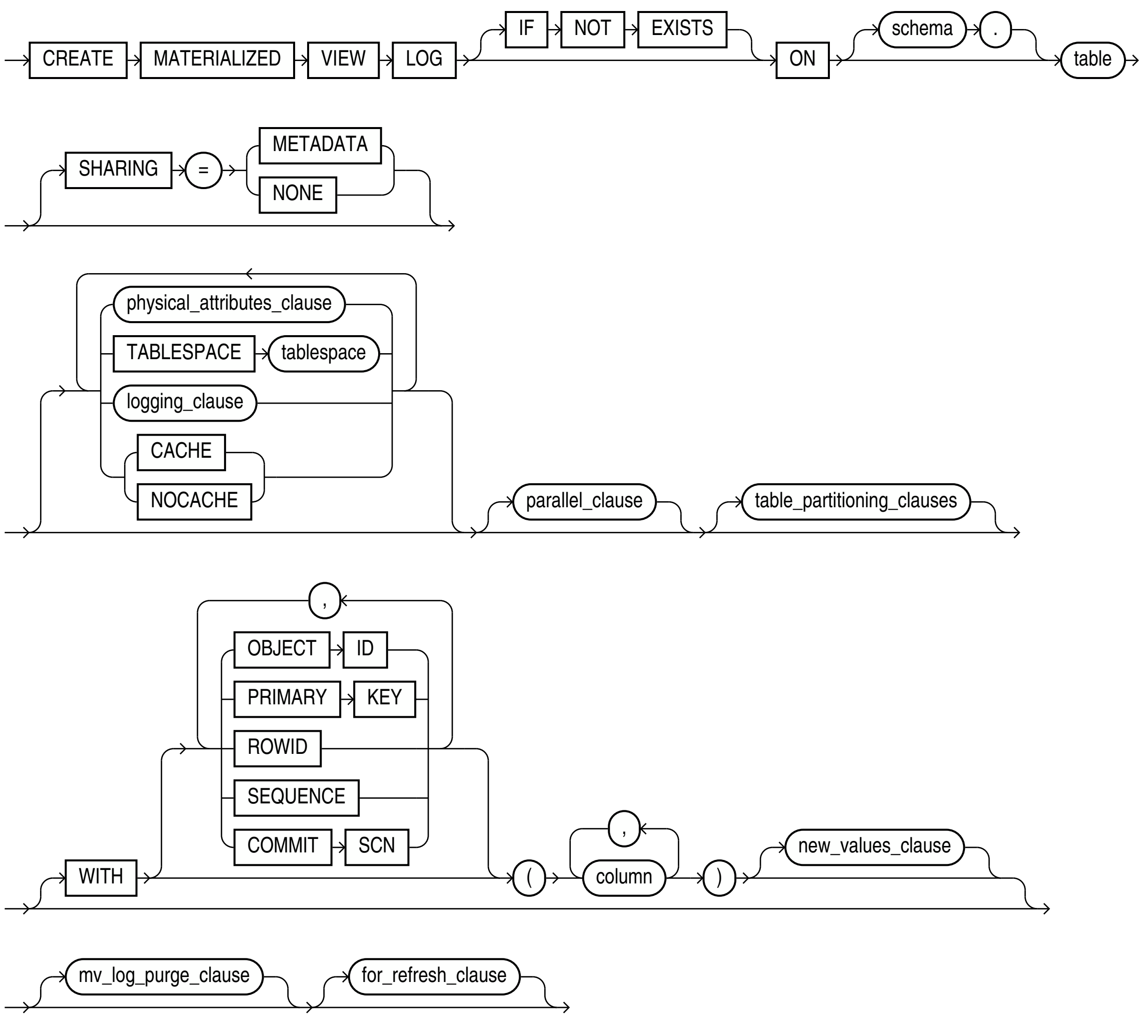
The statement CREATE MATERIALIZED VIEW LOG does not support the following columns in the Master Table:

- Hidden columns
- Identity columns
- BFILE columns

- Temporal validity columns

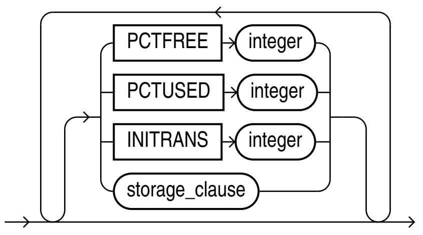
Syntax

***create_materialized_vw_log*::=**



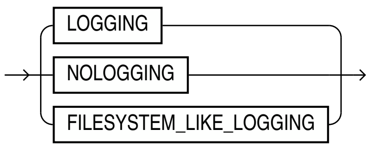
(*physical_attributes_clause*::=, *logging_clause*::=, *parallel_clause*::=, *table_partitioning_clauses*::= (in `CREATE TABLE`), *new_values_clause*::=, *mv_log_purge_clause*::=, *for_refresh_clause*::=.)

***physical_attributes_clause*::=**

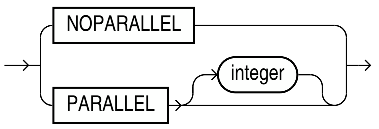


(*storage_clause::=*)

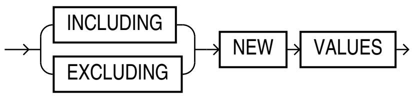
***logging_clause*::=**



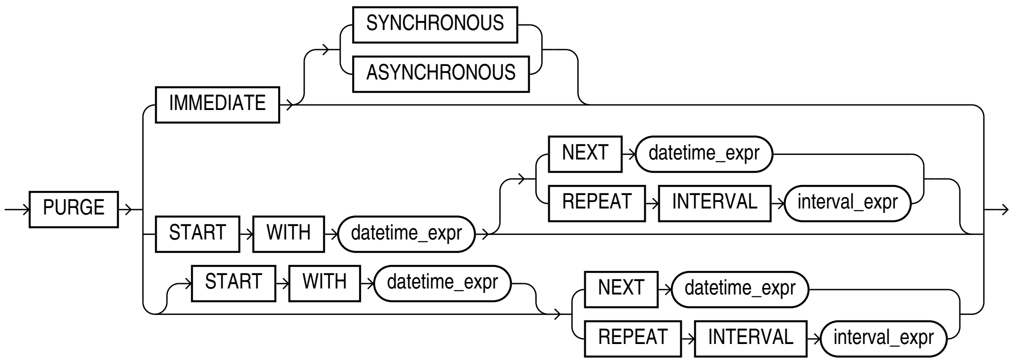
***parallel_clause*::=**



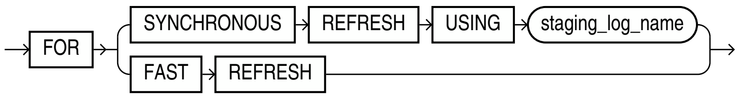
***new_values_clause*::=**



***mv_log_purge_clause*::=**



***for_refresh_clause*::=**



Semantics

IF NOT EXISTS

Specifying `IF NOT EXISTS` has the following effects:

- If the materialized view log does not exist, a new materialized view log is created at the end of the statement.
- If the materialized view log exists, this is the materialized view log you have at the end of the statement. A new one is not created because the older materialized view log is detected.

Using `IF EXISTS` with `CREATE` results in `ORA-11543: Incorrect IF NOT EXISTS clause for CREATE statement.`

schema

Specify the schema containing the materialized view log master table. If you omit *`schema`*, then Oracle Database assumes the master table is contained in your own schema. Oracle Database creates the materialized view log in the schema of its master table. You cannot create a materialized view log for a table in the schema of the user `SYS`.

table

Specify the name of the master table for which the materialized view log is to be created. Oracle Database encrypts any columns in the materialized view log that are encrypted in the master table, using the same encryption algorithm.

Restrictions on Master Tables of Materialized View Logs

The following restrictions apply to master tables of materialized view logs:

- You cannot create a materialized view log for a temporary table or for a view.
- You cannot create a materialized view log for a master table with a virtual column.

See Also:

"Creating a Materialized View Log for Fast Refresh: Examples"

SHARING

Use the sharing clause if you want to create the object in an application root in the context of an application maintenance. This type of object is called an application common object and it can be shared with the application PDBs that belong to the application root.

You can specify how the object is shared using one of the following sharing attributes:

- `METADATA` - A metadata link shares the metadata, but its data is unique to each container. This type of object is referred to as a metadata-linked application common object.
- `NONE` - The object is not shared and can only be accessed in the application root.

physical_attributes_clause

Use the *`physical_attributes_clause`* to define physical and storage characteristics for the materialized view log.

See Also:

physical_attributes_clause and storage_clause for a complete description these clauses, including default values

TABLESPACE Clause

Specify the tablespace in which the materialized view log is to be created. If you omit this clause, then the database creates the materialized view log in the default tablespace of the schema of the materialized view log.

logging_clause

Specify either `LOGGING` or `NOLOGGING` to establish the logging characteristics for the materialized view log. The default is the logging characteristic of the tablespace in which the materialized view log resides.

See Also:

logging_clause for a full description of this clause

CACHE | NOCACHE

For data that will be accessed frequently, `CACHE` specifies that the blocks retrieved for this log are placed at the most recently used end of the least recently used (LRU) list in the buffer cache when a full table scan is performed. This attribute is useful for small lookup tables.

`NOCACHE` specifies that the blocks are placed at the least recently used end of the LRU list. The default is `NOCACHE`.

Note:

`NOCACHE` has no effect on materialized view logs for which you specify `KEEP` in the *`storage_clause`*.

See Also:

CREATE TABLE for information about specifying `CACHE` or `NOCACHE`

parallel_clause

The *`parallel_clause`* lets you indicate whether parallel operations will be supported for the materialized view log.

For complete information on this clause, refer to *parallel_clause* in the documentation on `CREATE TABLE`.

table_partitioning_clauses

Use the *`table_partitioning_clauses`* to indicate that the materialized view log is partitioned on specified ranges of values or on a hash function. Partitioning of materialized view logs is the same as partitioning of tables.

See Also:

table_partitioning_clauses in the `CREATE TABLE` documentation

WITH Clause

Use the `WITH` clause to indicate whether the materialized view log should record the primary key, rowid, object ID, or a combination of these row identifiers when rows in the master are changed. You can also use this clause to add a sequence to the materialized view log to provide additional ordering information for its records.

This clause also specifies whether the materialized view log records additional columns that might be referenced as **filter columns**, which are non-primary-key columns referenced by subquery materialized views, or **join columns**, which are non-primary-key columns that define a join in the subquery `WHERE` clause.

If you omit this clause, or if you specify the clause without `PRIMARY KEY`, `ROWID`, or `OBJECT ID`, then the database stores primary key values by default. However, the database does not store primary key values implicitly if you specify only `OBJECT ID` or `ROWID` at create time. A primary key log, created either explicitly or by default, performs additional checking on the primary key constraint.

OBJECT ID

Specify `OBJECT ID` to indicate that the system-generated or user-defined object identifier of every modified row should be recorded in the materialized view log.

Restriction on OBJECT ID

You can specify `OBJECT ID` only when creating a log on an object table, and you cannot specify it for storage tables.

PRIMARY KEY

Specify `PRIMARY KEY` to indicate that the primary key of all rows changed should be recorded in the materialized view log.

ROWID

Specify `ROWID` to indicate that the rowid of all rows changed should be recorded in the materialized view log.

SEQUENCE

Specify `SEQUENCE` to indicate that a sequence value providing additional ordering information should be recorded in the materialized view log. Sequence numbers are necessary to support fast refresh after some update scenarios.

See Also:

Oracle Database Data Warehousing Guide for more information on the use of sequence numbers in materialized view logs and for examples that use this clause

COMMIT SCN

Without the `COMMIT SCN` clause, the materialized view log is based on timestamps and requires some setup operations when preparing to refresh the materialized view. Specify `COMMIT SCN` to instruct the database to use commit SCN data rather than timestamps. This setting removes the need for the setup operations and thus can improve the speed of the materialized view refresh.

You can create the following types of local materialized views (including both `ON COMMIT` and `ON DEMAND`) on master tables with commit SCN-based materialized view logs:

- Materialized aggregate views, including materialized aggregate views on a single table
- Materialized join views
- Primary-key-based and rowid-based single table materialized views
- `UNION ALL` materialized views, where each `UNION ALL` branch is one of the above materialized view types

You cannot create remote materialized views on master tables with commit SCN-based materialized view logs.

Restrictions on COMMIT SCN

The following restrictions apply to `COMMIT SCN`:

- Use of `COMMIT SCN` on a table with one or more LOB columns is not supported and causes `ORA-32421`.
- Creating a materialized view on master tables with different types of materialized view logs (that is, a master table with timestamp-based materialized view logs and a master table with commit SCN-based materialized view logs) is not supported and causes `ORA-32414`.
- If you specify `COMMIT SCN`, then you cannot specify `FOR SYNCHRONOUS REFRESH`.

column

Specify the columns whose values you want to be recorded in the materialized view log for all rows that are changed. Typically these columns are filter columns and join columns.

Restrictions on the WITH Clause

This clause is subject to the following restrictions:

- You can specify only one `PRIMARY KEY`, one `ROWID`, one `OBJECT ID`, one `SEQUENCE`, and one column list for each materialized view log.
- Primary key columns are implicitly recorded in the materialized view log. Therefore, you cannot specify any of the following combinations if *`column`* contains one of the primary key columns:

```
WITH ... PRIMARY KEY ... (column)
WITH ... (column) ... PRIMARY KEY
WITH (column)
```

See Also:

- CREATE MATERIALIZED VIEW for information on explicit and implicit inclusion of materialized view log values
- *Oracle Database Administrator's Guide* for more information about filter columns and join columns
- "Specifying Filter Columns for Materialized View Logs: Example" and "Specifying Join Columns for Materialized View Logs: Example"

NEW VALUES Clause

The `NEW VALUES` clause lets you determine whether Oracle Database saves both old and new values for update DML operations in the materialized view log.

See Also:

"Including New Values in Materialized View Logs: Example"

INCLUDING

Specify `INCLUDING` to save both new and old values in the log. If this log is for a table on which you have a single-table materialized aggregate view, and if you want the materialized view to be eligible for fast refresh, then you must specify `INCLUDING`.

EXCLUDING

Specify `EXCLUDING` to disable the recording of new values in the log. This is the default. You can use this clause to avoid the overhead of recording new values. Do not use this clause if you have a fast-refreshable single-table materialized aggregate view defined on the master table.

mv_log_purge_clause

Use this clause to specify the purge time for the materialized view log.

- `IMMEDIATE SYNCHRONOUS`: the materialized view log is purged immediately after refresh. This is the default.
- `IMMEDIATE ASYNCHRONOUS`: the materialized view log is purged in a separate Oracle Scheduler job after the refresh operation.
- `START WITH`, `NEXT`, and `REPEAT INTERVAL` set up a scheduled purge that is independent of the materialized view refresh and is initiated during `CREATE` or `ALTER MATERIALIZED VIEW LOG` statement. This is very similar to scheduled refresh syntax in a `CREATE` or `ALTER MATERIALIZED VIEW` statement:
  - The `START WITH` datetime expression specifies when the purge starts.
  - The `NEXT` datetime expression computes the next run time for the purge.

  If you specify `REPEAT INTERVAL`, then the next run time will be: `SYSDATE + `*`interval_expr`*.

  A `CREATE MATERIALIZED VIEW LOG` statement with a scheduled purge creates an Oracle Scheduler job to perform log purge. The job calls the `DBMS_SNAPSHOT.PURGE_LOG` procedure to purge the materialized view logs. This process allows you to amortize the purging costs over several materialized view refreshes.

Restriction on *mv_log_purge_clause*

This clause is not valid for materialized view logs on temporary tables.

See Also:

Oracle Database Data Warehousing Guide for more information on purging materialized view logs

for_refresh_clause

Use this clause to specify the refresh method for which the materialized view log will be used. You can specify only one refresh method for any given master table.

FOR SYNCHRONOUS REFRESH

Specify this clause to create a staging log that can be used for synchronous refresh. Use `staging_log_name` to specify the name of the staging log to be created. The staging log will be created in the schema in which the master table resides.

After you create the staging log, you cannot perform DML operations directly on the master table. You must use the procedures in the `DBMS_SYNC_REFRESH` package to prepare and execute change data operations.

Restrictions on Synchronous Refresh

The following restrictions apply to synchronous refresh:

- If you specify `FOR SYNCHRONOUS REFRESH`, then you cannot specify `COMMIT SCN`.
- To be eligible for synchronous refresh, the master table must satisfy the following criteria:
  - If the master table is a fact table, then it must be partitioned.
  - The master table must have a key. If the master table is a dimension table, then it must have a primary key defined on it. If the master table is a fact table, then the set of columns that are the foreign keys of the dimension tables joined to the fact table are deemed to be the key.
  - The master table cannot have a non-NULL Virtual Private Database (VPD) policy or a trigger defined on it.

  Oracle Database may allow you to create a staging log on a master table even if all of the preceding criteria are not met. However, the master table will not be eligible for synchronous refresh.
- Any existing materialized views on the master table must be refresh-on-demand materialized views. If an existing materialized view is a refresh-on-commit materialized view, then you must change it to a refresh-on-demand materialized view with the *alter_mv_refresh* clause of `ALTER MATERIALIZED VIEW` before you create the staging log.

See Also:

- *Oracle Database Data Warehousing Guide* for the complete steps for using synchronous refresh
- *Oracle Database PL/SQL Packages and Types Reference* for information on the `DBMS_SYNC_REFRESH` package

FOR FAST REFRESH

Specify this clause to create a materialized view log that can be used for fast refresh. The materialized view log will be created in the same schema in which the master table resides. This is the default.

Examples

Creating a Materialized View Log for Fast Refresh: Examples

The following statement creates a materialized view log on the `oe.customers` table that specifies physical and storage characteristics:

```
CREATE MATERIALIZED VIEW LOG ON customers
   PCTFREE 5
   TABLESPACE example
   STORAGE (INITIAL 10K);
```

The materialized view log on `customers` supports fast refresh for primary key materialized views only.

The following statement creates another version of the materialized view log with the `ROWID` clause, which enables fast refresh for more types of materialized views:

```
CREATE MATERIALIZED VIEW LOG ON customers WITH PRIMARY KEY, ROWID;
```

This materialized view log on `customers` makes fast refresh possible for rowid materialized views and for materialized join views. To provide for fast refresh of materialized aggregate views, you must also specify the `SEQUENCE` and `INCLUDING NEW VALUES` clauses, as shown in the example that follows.

Specify a Purge Repeat Interval for a Materialized View Log: Example

The following statement creates a materialized view log on the `oe.orders` table. The contents of the log will be purged once every five days, beginning five days after the creation date of the materialized view log:

```
CREATE MATERIALIZED VIEW LOG ON orders
  PCTFREE 5
  TABLESPACE example
  STORAGE (INITIAL 10K)
  PURGE REPEAT INTERVAL '5' DAY;
```

Specifying Filter Columns for Materialized View Logs: Example

The following statement creates a materialized view log on the `sh.sales` table and is used in "Creating Materialized Aggregate Views: Example". It specifies as filter columns all of the columns of the table referenced in that materialized view.

```
CREATE MATERIALIZED VIEW LOG ON sales
   WITH ROWID, SEQUENCE(amount_sold, time_id, prod_id)
   INCLUDING NEW VALUES;
```

Specifying Join Columns for Materialized View Logs: Example

The following statement creates a materialized view log on the `order_items` table of the sample `oe` schema. The log records primary keys and `product_id`, which is used as a join column in "Creating a Fast Refreshable Materialized View: Example".

```
CREATE MATERIALIZED VIEW LOG ON order_items WITH (product_id);
```

Including New Values in Materialized View Logs: Example

The following example creates a materialized view log on the `oe.product_information` table that specifies `INCLUDING NEW VALUES`:

```
CREATE MATERIALIZED VIEW LOG ON product_information
   WITH ROWID, SEQUENCE (list_price, min_price, category_id), PRIMARY KEY
   INCLUDING NEW VALUES;
```

You could create the following materialized aggregate view to use the `product_information` log:

```
CREATE MATERIALIZED VIEW products_mv
   REFRESH FAST ON COMMIT
   AS SELECT SUM(list_price - min_price), category_id
         FROM product_information
         GROUP BY category_id;
```

This materialized view is eligible for fast refresh because the log defined on its master table includes both old and new values.

Creating a Staging Log for Synchronous Refresh: Example

The following statement creates a staging log on the `sh.sales` fact table. The staging log is named `mystage_log` and is stored in the `sh` schema. It can be used for synchronous refresh.

```
CREATE MATERIALIZED VIEW LOG ON sales
   PCTFREE 5
   TABLESPACE example
   STORAGE (INITIAL 10K)
   FOR SYNCHRONOUS REFRESH USING mystage_log;
```

CREATE MATERIALIZED ZONEMAP

Purpose

Use the `CREATE MATERIALIZED ZONEMAP` statement to create a zone map.

A **zone map** is a special type of materialized view that stores information about zones. A **zone** is a set of contiguous data blocks on disk that stores the values of one or more table columns. Multiple zones are usually required to store all of the values of the table columns. A zone map tracks the minimum and maximum table column values stored in each zone.

Zone maps enable you to reduce the I/O and CPU costs of table scans. When a SQL statement contains predicates on columns in a zone map, the database compares the predicate values to the minimum and maximum table column values stored in each zone to determine which zones to read during SQL execution.

Oracle Database supports the following types of zone maps:

- A **basic zone map** is defined on a single table and maintains zone information for specified columns in that table.

  You can create a basic zone map either by specifying the *create_zonemap_on_table* clause, or by specifying the *create_zonemap_as_subquery* clause where the FROM clause of the defining subquery specifies a single table.
- A **join zone map** is defined on two or more joined tables and maintains zone information for specified columns in any of the joined tables.

  You can create a join zone map by specifying the *create_zonemap_as_subquery* clause. The FROM clause of the defining subquery must specify a table that is left outer joined with one or more other tables.

Zone maps are commonly used with star schemas in data warehousing environments. However, a star schema is not a requirement for creating a zone map. In either case, this reference uses star schema terminology to refer to the tables in a zone map. In a join zone map, the outer table of the join(s) is referred to as the **fact table**, and the tables with which this table is joined are referred to as **dimension tables**. Collectively these tables are called the **base tables** of the zone map. In a basic zone map, the single table on which the zone map is defined is referred to as both the fact table and the base table of the zone map.

A base table of a zone map can be a partitioned or composite-partitioned table. In this case, the zone map maintains minimum and maximum column values for each partition (and subpartition) as well as for each zone.

You can create zone maps for use with or without attribute clustering:

- To create a zone map for use with attribute clustering, use either of the following methods:
  - Use the CREATE MATERIALIZED ZONEMAP statement and include attribute clustered columns in the zone map. Refer to the *attribute_clustering_clause* of CREATE TABLE and the *attribute_clustering_clause* clause of ALTER TABLE for more information.
  - Specify the WITH MATERIALIZED ZONEMAP clause while creating or modifying an attribute clustered table. Refer to the zonemap_clause of CREATE TABLE and the MODIFY CLUSTERING clause of ALTER TABLE for more information.
- To create a zone map for use without attribute clustering, use the CREATE MATERIALIZED ZONEMAP statement and include columns that are not attribute clustered in the zone map.

See Also:

Oracle Database Data Warehousing Guide for more information on zone maps

Prerequisites

To create a zone map **in your own schema:**

- You must have the CREATE MATERIALIZED VIEW system privilege and either the CREATE TABLE or CREATE ANY TABLE system privilege.
- You must have access to any base tables of the zone map that you do not own, either through a READ or SELECT object privilege on each of the tables or through the READ ANY TABLE or SELECT ANY TABLE system privilege.

To create a zone map **in another user's schema:**

- You must have the CREATE ANY MATERIALIZED VIEW system privilege.

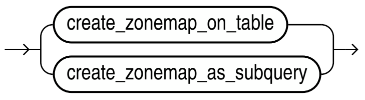
- The owner of the zone map must have the `CREATE TABLE` system privilege. The owner must also have access to any base tables of the zone map that the schema owner does not own, either through a `READ` or `SELECT` object privilege on each of the tables or through the `READ ANY TABLE` or `SELECT ANY TABLE` system privilege.

To create a refresh-on-commit zone map (`REFRESH ON COMMIT` clause), in addition to the preceding privileges, you must have the `ON COMMIT REFRESH` object privilege on any base tables that you do not own or you must have the `ON COMMIT REFRESH` system privilege. Unlike materialized views, you can create a refresh-on-commit zone map even if there are no materialized view logs on the base tables.

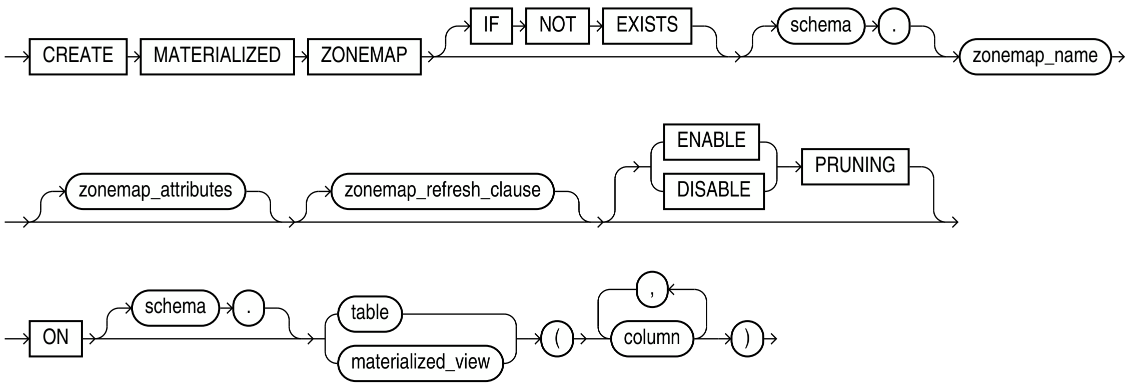
When you create a zone map, Oracle Database creates one internal table and at least one index, all in the schema of the zone map. Oracle Database uses these objects to maintain the zone map data. You must have the privileges necessary to create these objects, and you must have sufficient quota in the target tablespace to store these objects or you must have the `UNLIMITED TABLESPACE` system privilege.

Syntax

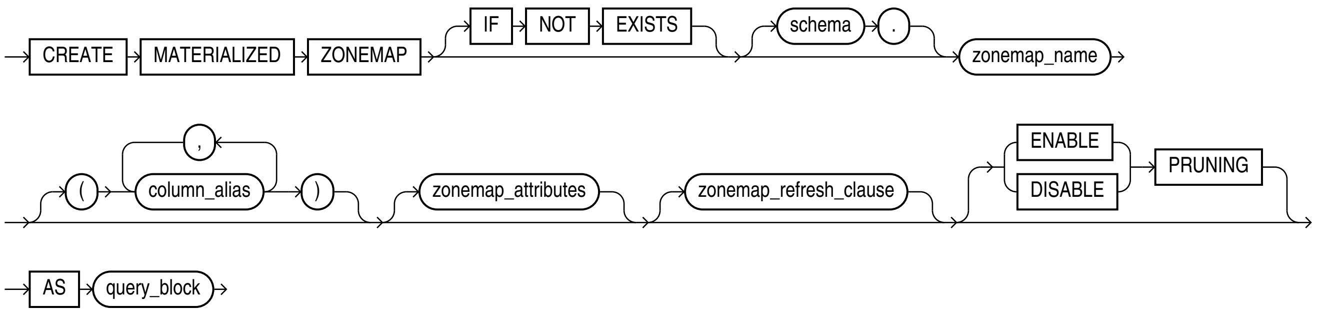
***create_materialized_zonemap*::=**



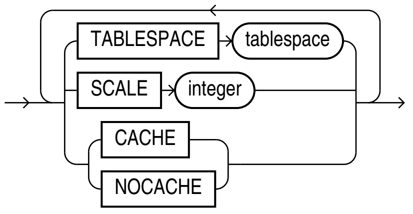
***create_zonemap_on_table*::=**



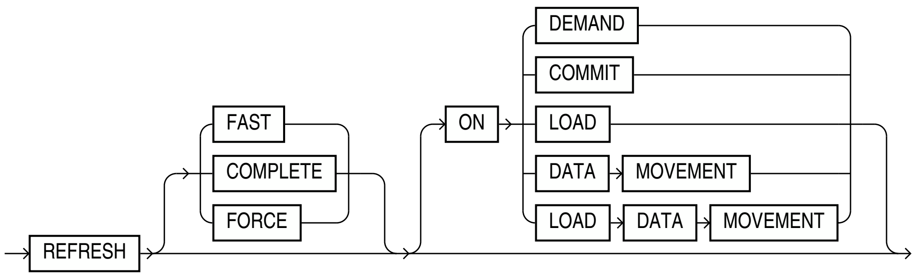
***create_zonemap_as_subquery*::=**



***zonemap_attributes*::=**



***zonemap_refresh_clause*::=**



Note:

When specifying the *`zonemap_refresh_clause`*, you must specify at least one clause after the `REFRESH` keyword.

Semantics

IF NOT EXISTS

Specifying `IF NOT EXISTS` has the following effects:

- If the materialized zonemap does not exist, a new materialized zonemap is created at the end of the statement.
- If the materialized zonemap exists, this is the materialized zonemap you have at the end of the statement. A new one is not created because the older materialized zonemap is detected.

Using `IF EXISTS` with `CREATE` results in `ORA-11543: Incorrect IF NOT EXISTS clause for CREATE statement.`

create_zonemap_on_table

Use this clause to create a basic zone map.

ON Clause

In the `ON` clause, first specify the fact table for the zone map, and then inside the parentheses specify one or more columns of the fact table to be included in the zone map.

For each specified fact table *`column`*, Oracle creates two columns in the zone map. These two columns contain the minimum and maximum values of the fact table column in each zone.

Oracle generates names for the zone map columns of the form `MIN_1_`*`column`* and `MAX_1_`*`column`* for the first specified fact table *`column`*, `MIN_2_`*`column`* and `MAX_2_`*`column`* for the second specified fact table *`column`*, and so on.

If you omit *`schema`*, then Oracle assumes the fact table is in your own schema. The fact table can be a table or a materialized view

create_zonemap_as_subquery

Use this clause to create a basic zone map or a join zone map. To create a basic zone map, specify a single base table in the `FROM` clause of the defining subquery. To create a join zone map, specify a table that is left outer joined to one or more other tables in the `FROM` clause of the defining subquery.

column_alias

You can specify a column alias for each table column to be included in the zone map. The column alias list explicitly resolves any column name conflict, eliminating the need to specify aliases in the `SELECT` list of the defining subquery. If you specify any column alias in this clause, then you must specify an alias for each column in the `SELECT` list of the defining subquery. The first column alias you specify must be `ZONE_ID$`, which corresponds to the first column in the `SELECT` list, the `SYS_OP_ZONE_ID` function expression.

AS *query_block*

Specify the defining subquery of the zone map. The subquery must consist of a single *`query_block`*. You can specify only the `SELECT`, `FROM`, `WHERE`, and `GROUP BY` clauses of *`query_block`*, and those clauses must satisfy the following requirements:

- The first column in the `SELECT` list must be the `SYS_OP_ZONE_ID` function expression. Refer to SYS_OP_ZONE_ID for more information.
- The remaining columns in the `SELECT` list must be function expressions that return minimum and maximum values for the columns you want to include in the zone map. For each column, specify a pair of function expressions of the following form:

  ```
  MIN([table.]column), MAX([table.]column)
  ```

  For *`table`*, specify the name or table alias for the table that contains the column. The table can be a fact table or dimension table. For *`column`*, specify the name or column alias for the column.
- The `FROM` clause can specify a fact table alone, or a fact table and one or more dimension tables with each dimension table left outer joined to the fact table. You can specify `LEFT [OUTER] JOIN` syntax in the `FROM` clause, or apply the outer join operator (+) to dimension table columns in the join condition in the `WHERE` clause. You can optionally specify a table alias for any of the tables in the `FROM` clause. Fact tables and dimension tables can be tables or materialized views.
- In the `WHERE` clause, you can specify only left outer join conditions using the outer join operator(+).
- You must specify a `GROUP BY` clause with the same `SYS_OP_ZONE_ID` function expression that you specified for the first column of the `SELECT` list.

schema

Specify the schema to contain the zone map. If you omit *`schema`*, then Oracle Database creates the zone map in your schema.

zonemap_name

Specify the name of the zone map to be created. The name must satisfy the requirements listed in "Database Object Naming Rules ".

zonemap_attributes

Use this clause to specify the following attributes for the zone map: `TABLESPACE`, `SCALE`, `PCTFREE`, `PCTUSED`, and `CACHE` or `NOCACHE`.

TABLESPACE

Specify the *`tablespace`* in which the zone map is to be created. If you omit this clause, then Oracle Database creates the zone map in the default tablespace of the schema containing the zone map.

SCALE

This clause lets you specify the zone map scale, which determines the number of contiguous disk blocks that form a zone. The scale is an integer value that represents a power of 2. For example, a scale of 10 means up to 2 raised to the 10th power, or 1024, contiguous disk blocks will form a zone. For *`integer`*, specify a value between 4 and 16, inclusive. The recommended value is 10; this is the default.

PCTFREE

Specify an *`integer`* representing the percentage of space in each data block of the zone map reserved for future updates to rows of the zone map. The integer value must be between 0 and 99, inclusive. The default value is 10. Refer to *physical_attributes_clause* for more information on the `PCTFREE` parameter.

PCTUSED

Specify an *`integer`* representing the minimum percentage of used space that Oracle maintains for each data block of the zone map. The integer value must be between 0 and 99, inclusive. The default value is 40. Refer to *physical_attributes_clause* for more information on the `PCTUSED` parameter.

CACHE | NOCACHE

For data that will be accessed frequently, `CACHE` specifies that the blocks retrieved for this zone map are placed at the most recently used end of the least recently used (LRU) list in the buffer cache when a full table scan is performed.

`NOCACHE` specifies that the blocks are placed at the least recently used end of the LRU list. The default is `NOCACHE`.

zonemap_refresh_clause

Use this clause to specify the default refresh method and mode for the zone map. If you do not specify a refresh method (`FAST`, `COMPLETE`, or `FORCE`), then `FORCE` is the default method. If you do not specify a refresh mode (`ON` clauses), then `ON LOAD DATA MOVEMENT` is the default mode.

FAST

Specify `FAST` to indicate the fast refresh method, which performs the refresh according to the changes that have occurred to the base tables. While zone maps are internally implemented as a type of materialized view, materialized view logs on base tables are not needed to perform a fast refresh of a zone map

COMPLETE

Specify `COMPLETE` to indicate the complete refresh method, which is implemented by executing the defining query of the zone map. If you request a complete refresh, then Oracle Database performs a complete refresh even if a fast refresh is possible.

FORCE

Specify `FORCE` to indicate that when a refresh occurs, Oracle Database will perform a fast refresh if one is possible or a complete refresh if fast refresh is not possible. This is the default.

ON DEMAND

Specify `ON DEMAND` to indicate that database will not refresh the zone map unless you manually issue an `ALTER MATERIALIZED ZONEMAP` ... `REBUILD` statement. If you specify this clause, then the zone map is referred to as a refresh-on-demand zone map. Refer to REBUILD in the documentation on `ALTER MATERIALIZED ZONEMAP` for more information on rebuilding a zone map.

ON COMMIT

Specify `ON COMMIT` to indicate that a refresh is to occur whenever the database commits a transaction that operates on a base table of the zone map. If you specify this clause, then the zone map is referred to as a refresh-on-commit zone map. This clause may increase the time taken to complete the commit, because the database performs the refresh operation as part of the commit process.

ON LOAD

Specify `ON LOAD` to indicate that a refresh is to occur at the end of a direct-path insert (serial or parallel) resulting either from an `INSERT` or a `MERGE` operation.

ON DATA MOVEMENT

Specify `ON DATA MOVEMENT` to indicate that a refresh is to occur at the end of the following data movement operations:

- Data redefinition using the `DBMS_REDEFINITION` package
- Table partition maintenance operations that are specified by the following clauses of `ALTER TABLE`: *`coalesce_table`*, *`merge_table_partitions`*, *`move_table_partition`*, and *`split_table_partition`*

ON LOAD DATA MOVEMENT

Specify `ON LOAD DATA MOVEMENT` to indicate that a refresh is to occur at the end of a direct-path insert or a data movement operation. This is the default.

ENABLE | DISABLE PRUNING

This clause lets you control the use of the zone map for pruning.

- Specify `ENABLE PRUNING` to enable use of the zone map for pruning. This is the default.
- Specify `DISABLE PRUNING` to disable use of the zone map for pruning. The optimizer will not use the zone map for pruning, but the database will continue to maintain the zone map.

If the setting is `ENABLE PRUNING`, then the optimizer will consider using the zone map for pruning during SQL operations that include any of the following conditions:

- Comparison conditions: `=`, `<=`, `<`, `>=`, `>`

The condition must be a simple comparison condition that has a column name on one side and a literal or bind variable on the other side. For example:

```
WHERE country_name = 'United States of America'
WHERE country_name = :country1
WHERE 10000 >= salary
```

- `IN` condition

  The `IN` condition must have a column name on the left side and an expression list of literals or bind variables on the right side. For example:

  ```
  WHERE country_name IN ('Germany', 'India', 'United Kingdom')
  WHERE country_name IN (:country1, :country2, :country3)
  WHERE prod_id IN (20, 48, 132, 143)
  ```

- `LIKE` condition

  The `LIKE` condition must have a column name on the left side and a text literal on the right side. The text literal is the pattern for the `LIKE` condition and it must contain at least one pattern matching character. Valid pattern matching characters are the underscore (_), which matches exactly one character, and the percent sign (`%`), which matches zero or more characters. The first character of the pattern cannot be a pattern matching character. For example:

  ```
  WHERE prod_name LIKE 'DVD%'
  WHERE prod_name LIKE 'Model%Cordless%Battery'
  WHERE prod_name LIKE 'CD%Pack of _'
  ```

See Also:

Conditions for more information on conditions

Restrictions on Zone Maps

Zone maps are subject to the following restrictions:

- A table can be a fact table for at most one zone map. A table can be a dimension table for multiple zone maps. A table can be a fact table for one zone map and a dimension table for other zone maps.
- A base table of a zone map cannot be an external table, an index-organized table, a remote table, a temporary table, or a view.
- A base table of a zone map cannot be in the schema of the user `SYS`.
- A zone map cannot be partitioned.
- You can define a zone map on a column of any scalar data type other than `BFILE`, `BLOB`, `CLOB`, `LONG`, `LONG RAW`, or `NCLOB`.
- All joins specified in the defining subquery of a zone map must be left outer equijoins with the fact table on the left side.
- If the `FROM` clause of the defining subquery for a zone map references a materialized view, then you must refresh that materialized view before refreshing the zone map.
- You cannot perform DML operations directly on a zone map.
- Each column of the zone map must have one of the following declared collations: `BINARY` or `USING_NLS_COMP`.

Examples

The following statement creates a basic zone map called `sales_zmap`. The zone map tracks columns `cust_id` and `prod_id` in the table `sales`.

```
CREATE MATERIALIZED ZONEMAP sales_zmap
  ON sales(cust_id, prod_id);
```

The following statement creates a basic zone map called `sales_zmap` that is similar to the zone map created in the previous example. However, this statement uses a defining subquery to create the zone map.

```
CREATE MATERIALIZED ZONEMAP sales_zmap
  AS SELECT SYS_OP_ZONE_ID(rowid),
            MIN(cust_id), MAX(cust_id),
            MIN(prod_id), MAX(prod_id)
     FROM sales
     GROUP BY SYS_OP_ZONE_ID(rowid);
```

The following statement creates a join zone map called `sales_zmap`. The fact table for the zone map is `sales` and the zone map has one dimension table: `customers`. The zone map tracks two columns in the dimension table: `cust_state_province` and `cust_city`.

```
CREATE MATERIALIZED ZONEMAP sales_zmap
  AS SELECT SYS_OP_ZONE_ID(s.rowid),
            MIN(cust_state_province), MAX(cust_state_province),
            MIN(cust_city), MAX(cust_city)
     FROM sales s
          LEFT OUTER JOIN customers c ON s.cust_id = c.cust_id
     GROUP BY SYS_OP_ZONE_ID(s.rowid);
```

The following statement creates a join zone map called `sales_zmap`. The fact table for the zone map is `sales` and the zone map has two dimension tables: `products` and `customers`. The zone map tracks five columns in the dimension tables: `prod_category` and `prod_subcategory` in the `products` table, and `country_id`, `cust_state_province`, and `cust_city` in the `customers` table.

```
CREATE MATERIALIZED ZONEMAP sales_zmap
  AS SELECT SYS_OP_ZONE_ID(s.rowid),
            MIN(prod_category), MAX(prod_category),
            MIN(prod_subcategory), MAX(prod_subcategory),
            MIN(country_id), MAX(country_id),
            MIN(cust_state_province), MAX(cust_state_province),
            MIN(cust_city), MAX(cust_city)
    FROM sales s
       LEFT OUTER JOIN products p ON s.prod_id = p.prod_id
       LEFT OUTER JOIN customers c ON s.cust_id = c.cust_id
    GROUP BY sys_op_zone_id(s.rowid);
```

The following statement creates a join zone map that is identical to the zone map created in the previous example. The only difference is that the previous example uses the `LEFT OUTER JOIN` syntax in the `FROM` clause and the following example uses the outer join operator (+) in the `WHERE` clause.

```
CREATE MATERIALIZED ZONEMAP sales_zmap
  AS SELECT SYS_OP_ZONE_ID(s.rowid),
            MIN(prod_category), MAX(prod_category),
            MIN(prod_subcategory), MAX(prod_subcategory),
            MIN(country_id), MAX(country_id),
            MIN(cust_state_province), MAX(cust_state_province),
            MIN(cust_city), MAX(cust_city)
     FROM sales s, products p, customers c
     WHERE s.prod_id = p.prod_id(+) AND
           s.cust_id = c.cust_id(+)
     GROUP BY sys_op_zone_id(s.rowid);
```

CREATE MLE ENV

Purpose

MLE Environments are first-class schema objects that can be managed on their own and reused across multiple execution contexts.

MLE uses execution contexts to execute MLE language code and MLE environments allow you configure properties for these execution contexts. You can set language options to customize the runtime of the MLE language and you can enable specific MLE modules to be imported to an execution context and manage dependencies.

Use `CREATE MLE ENV` to create a new MLE environment in the database in one of three ways:

You can create an MLE environment in one of three ways :

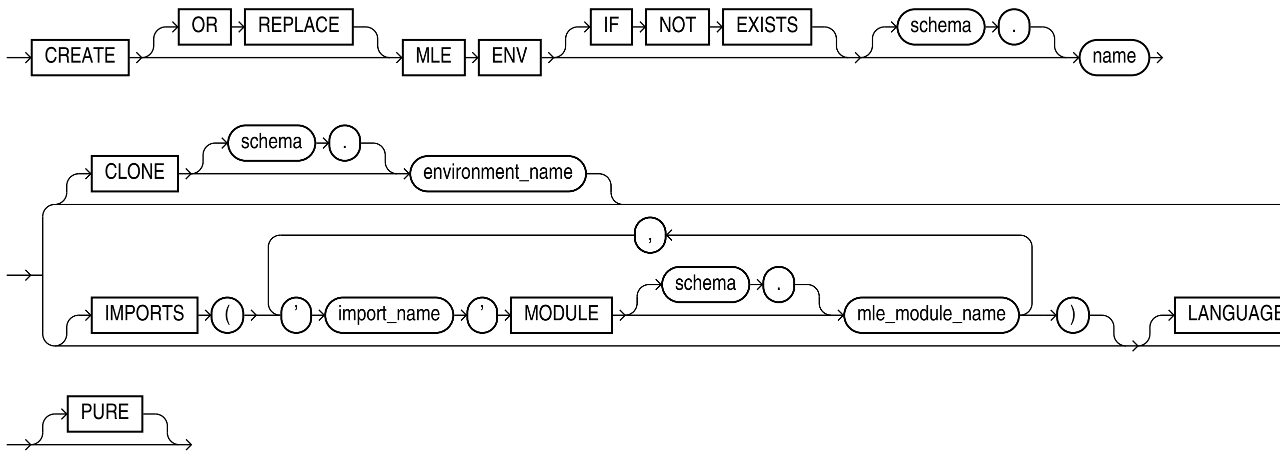
- As a fresh empty environment.
- As an environment with a list of imports and a language option string.
- By by cloning an existing environment. Cloning an environment creates an independent copy that is not affected by subsequent changes to the original environment.

MLE environments use the same namespace as tables and procedures.

Prerequisites

Users must have the `CREATE MLE` privilege to create an environment in their own schema, and the `CREATE ANY MLE` privilege to create an environment in other schemas. The user creating the environment must either own the environment being cloned or should have the `EXECUTE` privilege on it.

Syntax



Semantics

OR REPLACE

Specify `OR REPLACE` to re-create the environment, if it already exists. You can use this clause to change the definition of an existing environment without dropping, re-creating, and regranting object privileges previously granted on it.

IF NOT EXISTS

Specifying `IF NOT EXISTS` has the following effects:

- If the MLE environment does not exist, a new MLE environment is created at the end of the statement.
- If the MLE environment, this is the MLE environment you have at the end of the statement. A new one is not created because the older one is detected.

You can have one of `OR REPLACE` or `IF NOT EXISTS` in a statement at a time. Using both `OR REPLACE` with `IF NOT EXISTS` in the very same statement results in the error: `REPLACE and IF NOT EXISTS cannot coexist in the same DDL statement.`

PURE

Specify `PURE` on MLE MLE environments and JavaScript inline call specifications create restricted JavaScript execution contexts.

For more see About Restricted Execution Contexts of the *JavaScript Developer's Guide*.

Examples

The following example creates an empty MLE environment `myenv`.

```
CREATE MLE ENV scott."myenv";
```

The following example clones an existing MLE environment:

```
CREATE MLE ENV scott."myenv" CLONE "other_env";
```

See Also:

- *MLE JavaScript Modules and Environments*
- ALTER MLE ENV
- DROP MLE ENV

CREATE MLE MODULE

Purpose

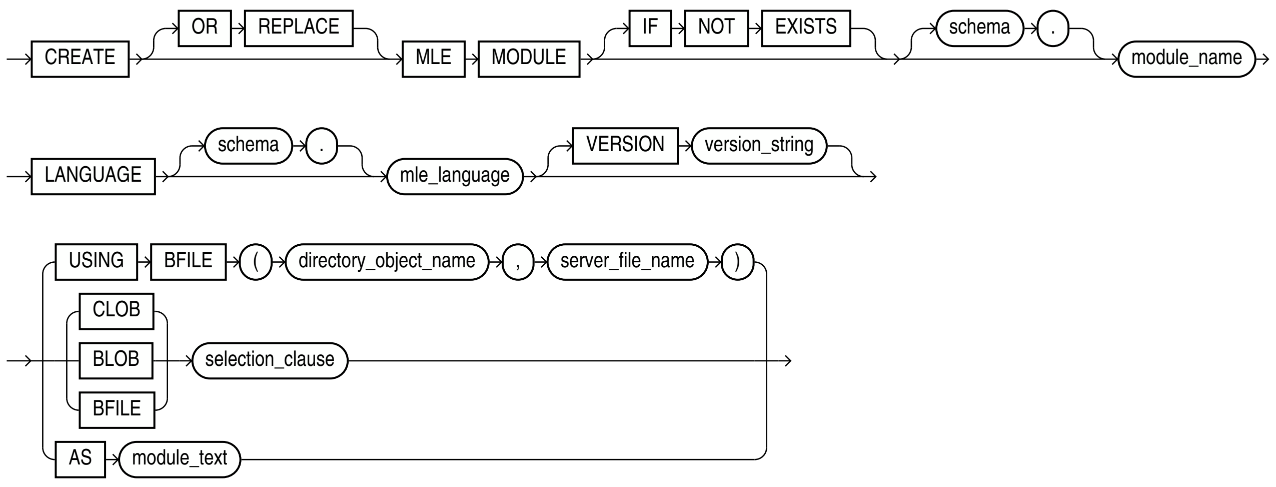
Multilingual Engine (MLE) allows developers to write, store, and execute JavaScript code in Oracle Database Release 23 on Linux `x86-64` by using MLE Modules to encapsulate JavaScript.

Use `CREATE MLE MODULE` to create a new MLE module in the database.

See Also:

JavaScript Developer's Guide

Syntax



Semantics

OR REPLACE

Specify `OR REPLACE` to re-create the module if it already exists. You can use this clause to change the definition of an existing module without dropping, re-creating, and regranting object privileges previously granted on it.

IF NOT EXISTS

Specifying `IF NOT EXISTS` has the following effects:

- If the MLE module does not exist, a new MLE module is created at the end of the statement.
- If the MLE module exists, this is the MLE module you have at the end of the statement. A new one is not created because the older one is detected.

You can have one of `OR REPLACE` or `IF NOT EXISTS` in a statement at a time. Using both `OR REPLACE` with `IF NOT EXISTS` in the very same statement results in the following error: `ORA-11541: REPLACE and IF NOT EXISTS cannot coexist in the same DDL statement.`

schema

Use *`schema`* to fully qualify the module name. If you do not specify *`schema`*, then the current schema is used.

The length of the module name must not exceed 128 bytes. MLE modules use the same namespace as tables and procedures.

LANGUAGE, VERSION

Use `LANGUAGE` to specify the MLE language of the created module. You must use the value `JAVASCRIPT` when you create JavaScript modules. An `ORA-04101` error is thrown if an unsupported MLE language is used.

The optional `VERSION` clause specifies a version string for the MLE module. Version strings are purely informational and do not influence any behavior of MLE or the RDBMS. The version string must fit into a `VARCHAR2(256)`.

CLOB, BLOB, BFILE

Use `USING` to create MLE modules from code contained in `CLOB`s, `BLOB`s, or `BFILE`s.

You can specify the `BFILE` clause with a subquery or with a directory using *`directory_object_name`* and *`server_file_name`* to specify the directory and filename of the MLE module you want to use. Note that you must create the directory object before this step using `CREATE DIRECTORY` .

The `CLOB | BLOB | BFILE` clause specifies a subquery whose result must be a single row and column of the specified type (`CLOB`, `BLOB`, or `BFILE`) that holds the contents of the MLE module to be deployed. The `CLOB` option is available only if the MLE module contains textual data. The textual data in MLE modules contained in `BLOB`s and `BFILE`s is encoded in `UTF-8`.

AS

Use `AS` to specify the contents of the MLE module as a sequence of characters inlined in the DDL statement. As with `CLOB`s, the `AS` clause is only available when the source of the MLE module contains textual data. Do not encapsulate the character sequence within quotes. The character sequence is delimited by the end of the DDL statement.

See Also:

- *JavaScript Developer's Guide*
- DROP MLE MODULE
- ALTER MLE MODULE
- CREATE MLE ENV

CREATE OPERATOR

Purpose

Use the `CREATE OPERATOR` statement to create a new operator and define its bindings.

Operators can be referenced by indextypes and by SQL queries and DML statements. The operators, in turn, reference functions, packages, types, and other user-defined objects.

See Also:

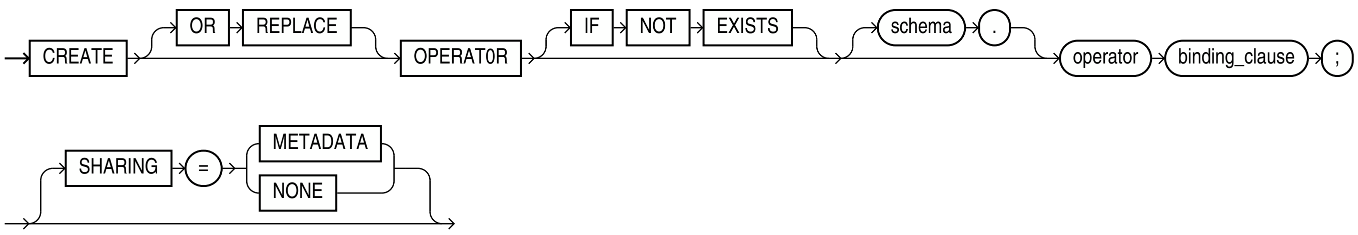
Oracle Database Data Cartridge Developer's Guide and *Oracle Database Concepts* for a discussion of these dependencies and of operators in general

Prerequisites

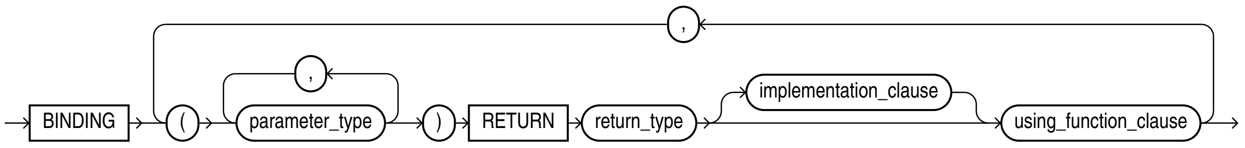
To create an operator in your own schema, you must have the `CREATE OPERATOR` system privilege. To create an operator in another schema, you must have the `CREATE ANY OPERATOR` system privilege. In either case, you must also have the `EXECUTE` object privilege on the functions and operators referenced.

Syntax

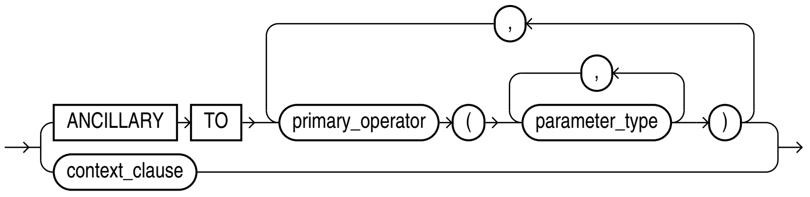
create_operator::=



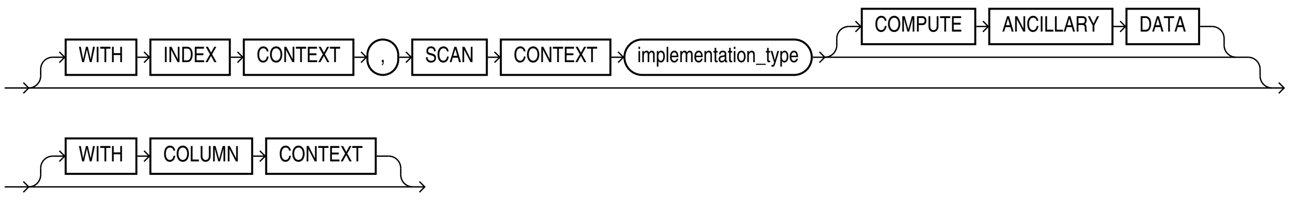
binding_clause::=



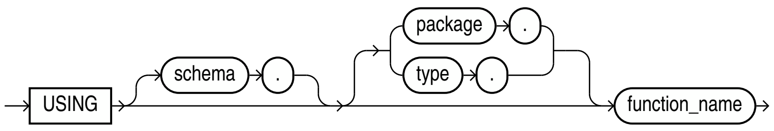
implementation_clause::=



***context_clause*::=**



***using_function_clause*::=**



Semantics

OR REPLACE

Specify `OR REPLACE` to replace the definition of the operator schema object.

Restriction on Replacing an Operator

You can replace the definition only if the operator has no dependent objects, such as indextypes supporting the operator.

IF NOT EXISTS

Specifying `IF NOT EXISTS` has the following effects:

- If the operator does not exist, a new operator is created at the end of the statement.
- If the operator exists, this is the operator you have at the end of the statement. A new one is not created because the older one is detected.

You can have *one* of `OR REPLACE` or `IF NOT EXISTS` in a statement at a time. Using both `OR REPLACE` with `IF NOT EXISTS` in the very same statement results in the following error: `ORA-11541: REPLACE and IF NOT EXISTS cannot coexist in the same DDL statement`.

Using `IF EXISTS` with `CREATE` results in `ORA-11543: Incorrect IF NOT EXISTS clause for CREATE statement`.

schema

Specify the schema containing the operator. If you omit *`schema`*, then the database creates the operator in your own schema.

operator

Specify the name of the operator to be created. The name must satisfy the requirements listed in "Database Object Naming Rules ".

binding_clause

Use the *`binding_clause`* to specify one or more parameter data types (*`parameter_type`*) for binding the operator to a function. The signature of each binding—the sequence of the data types of the arguments to the corresponding function—must be unique according to the rules of overloading.

The *`parameter_type`* can itself be an object type. If it is, then you can optionally qualify it with its schema.

Restriction on Binding Operators

You cannot specify a *`parameter_type`* of `REF`, `LONG`, or `LONG RAW`.

See Also:

Oracle Database PL/SQL Language Reference for more information about overloading

RETURN Clause

Specify the return data type for the binding.

The *`return_type`* can itself be an object type. If so, then you can optionally qualify it with its schema.

Restriction on Binding Return Data Type

You cannot specify a *`return_type`* of `REF`, `LONG`, or `LONG RAW`.

SHARING

Use the sharing clause if you want to create the object in an application root in the context of an application maintenance. This type of object is called an application common object and it can be shared with the application PDBs that belong to the application root.

You can specify how the object is shared using one of the following sharing attributes:

- `METADATA` - A metadata link shares the metadata, but its data is unique to each container. This type of object is referred to as a metadata-linked application common object.
- `NONE` - The object is not shared and can only be accessed in the application root.

implementation_clause

Use this clause to describe the implementation of the binding.

ANCILLARY TO Clause

Use the `ANCILLARY TO` clause to indicate that the operator binding is ancillary to the specified primary operator binding (*`primary_operator`*). If you specify this clause, then do not specify a previous binding with just one number parameter.

context_clause

Use the *`context_clause`* to describe the functional implementation of a binding that is not ancillary to a primary operator binding.

WITH INDEX CONTEXT, SCAN CONTEXT

Use this clause to indicate that the functional evaluation of the operator uses the index and a scan context that is specified by the implementation type.

COMPUTE ANCILLARY DATA

Specify `COMPUTE ANCILLARY DATA` to indicate that the operator binding computes ancillary data.

WITH COLUMN CONTEXT

Specify `WITH COLUMN CONTEXT` to indicate that Oracle Database should pass the column information to the functional implementation for the operator.

If you specify this clause, then the signature of the function implemented must include one extra `ODCIFuncCallInfo` structure.

See Also:

Oracle Database Data Cartridge Developer's Guide for instructions on using the `ODCIFuncCallInfo` routine

using_function_clause

The *`using_function_clause`* lets you specify the function that provides the implementation for the binding. The *`function_name`* can be a standalone function, packaged function, type method, or a synonym for any of these.

If the function is subsequently dropped, then the database marks all dependent objects `INVALID`, including the operator. However, if you then subsequently issue an `ALTER OPERATOR` ... `DROP BINDING` statement to drop the binding, then subsequent queries and DML will revalidate the dependent objects.

Examples

Creating User-Defined Operators: Example

This example creates a very simple functional implementation of equality and then creates an operator that uses the function. For a more complete set of examples, see *Oracle Database Data Cartridge Developer's Guide*.

```
CREATE FUNCTION eq_f(a VARCHAR2, b VARCHAR2) RETURN NUMBER AS
BEGIN
   IF a = b THEN RETURN 1;
   ELSE RETURN 0;
   END IF;
END;
/

CREATE OPERATOR eq_op
   BINDING (VARCHAR2, VARCHAR2)
   RETURN NUMBER
   USING eq_f;
```

CREATE OUTLINE

Purpose

Note:

Stored outlines are deprecated. They are still supported for backward compatibility. However, Oracle recommends that you use SQL plan management instead. SQL plan management creates SQL plan baselines, which offer superior SQL performance stability compared with stored outlines.

You can migrate existing stored outlines to SQL plan baselines by using the `MIGRATE_STORED_OUTLINE` function of the `DBMS_SPM` package or Enterprise Manager Cloud Control. When the migration is complete, the stored outlines are marked as migrated and can be removed. You can drop all migrated stored outlines on your system by using the `DROP_MIGRATED_STORED_OUTLINE` function of the `DBMS_SPM` package.

See Also: *Oracle Database SQL Tuning Guide* for more information about SQL plan management and *Oracle Database PL/SQL Packages and Types Reference* for information about the `DBMS_SPM` package

Use the `CREATE OUTLINE` statement to create a **stored outline**, which is a set of attributes used by the optimizer to generate an execution plan. You can then instruct the optimizer to use a set of outlines to influence the generation of execution plans whenever a particular SQL statement is issued, regardless of changes in factors that can affect optimization. You can also modify an outline so that it takes into account changes in these factors.

Note:

The SQL statement you want to affect must be an exact string match of the statement specified when creating the outline.

See Also:

- *Oracle Database SQL Tuning Guide* for information on execution plans
- ALTER OUTLINE for information on modifying an outline
- ALTER SESSION and ALTER SYSTEM for information on the `USE_STORED_OUTLINES` and `USE_PRIVATE_OUTLINES` parameters

Prerequisites

To create a public or private outline, you must have the `CREATE ANY OUTLINE` system privilege.

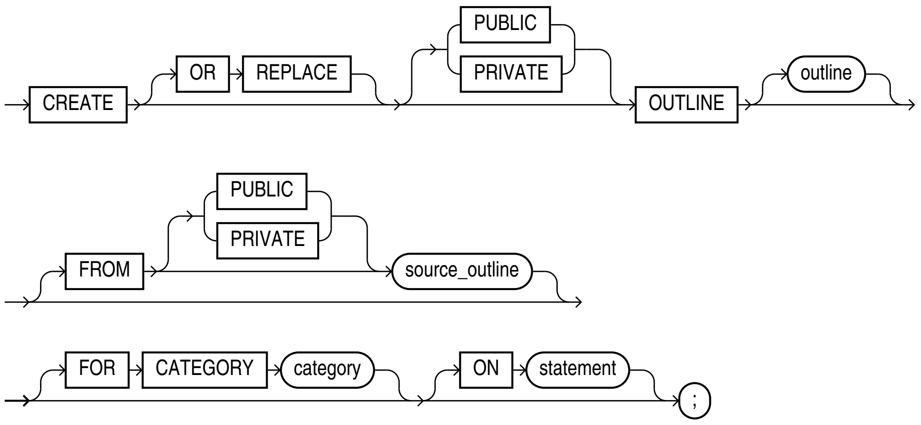
If you are creating a clone outline from a source outline, then you must also have the `SELECT_CATALOG_ROLE` role.

You can enable or disable the use of stored outlines dynamically for an individual session or for the system:

- Enable the `USE_STORED_OUTLINES` parameter to use public outlines.
- Enable the `USE_PRIVATE_OUTLINES` parameter to use private stored outlines.

Syntax

***create_outline*::=**



Note:

None of the clauses after *`outline`* are required. However, you must specify at least one clause after *`outline`*, and it must be either the `FROM` clause or the `ON` clause.

Semantics

OR REPLACE

Specify `OR REPLACE` to replace an existing outline with a new outline of the same name.

PUBLIC | PRIVATE

Specify `PUBLIC` if you are creating an outline for use by `PUBLIC`. This is the default.

Specify `PRIVATE` to create an outline for private use by the current session only. The data of this outline is stored in the current schema.

outline

Specify the unique name to be assigned to the stored outline. The name must satisfy the requirements listed in "Database Object Naming Rules ". If you do not specify *`outline`*, then the database generates an outline name.

See Also:

"Creating an Outline: Example"

FROM *source_outline* Clause

Use the `FROM` clause to create a new outline by copying an existing one. By default, Oracle Database looks for *`source_category`* in the public area. If you specify `PRIVATE`, then the database looks for the outline in the current schema.

Restriction on Copying an Outline

If you specify the `FROM` clause, then you cannot specify the `ON` clause.

> **See Also:**
>
> "Creating a Private Clone Outline: Example" and "Publicizing a Private Outline to the Public Area: Example"

FOR CATEGORY Clause

Specify an optional name used to group stored outlines. For example, you could specify a category of outlines for end-of-week use and another for end-of-quarter use. If you do not specify *`category`*, then the outline is stored in the `DEFAULT` category.

ON Clause

Specify the SQL statement for which the database will create an outline when the statement is compiled. This clause is optional only if you are creating a copy of an existing outline using the `FROM` clause.

You can specify any one of the following statements: `SELECT`, `DELETE`, `UPDATE`, `INSERT` ... `SELECT`, `CREATE TABLE` ... `AS SELECT`.

Restrictions on the ON Clause

This clause is subject to the following restrictions:

- If you specify the `ON` clause, then you cannot specify the `FROM` clause.
- You cannot create an outline on a multitable `INSERT` statement.
- The SQL statement in the `ON` clause cannot include any DML operation on a remote object.

> **Note:**
>
> In subsequent statements, you can specify additional outlines for the same SQL statement, but each outline for the same statement must specify a different category in the `CATEGORY` clause.

Examples

Creating an Outline: Example

The following statement creates a stored outline by compiling the `ON` statement. The outline is called `salaries` and is stored in the category `special`.

```
CREATE OUTLINE salaries FOR CATEGORY special
   ON SELECT last_name, salary FROM employees;
```

When this same `SELECT` statement is subsequently compiled, if the `USE_STORED_OUTLINES` parameter is set to `special`, the database generates the same execution plan as was generated when the outline `salaries` was created.

Creating a Private Clone Outline: Example

The following statement creates a stored private outline `my_salaries` based on the public category `salaries` created in the preceding example.

```
CREATE OR REPLACE PRIVATE OUTLINE my_salaries
   FROM salaries;
```

Publicizing a Private Outline to the Public Area: Example

The following statement copies back (publicizes) a private outline to the public area after private editing:

```
CREATE OR REPLACE OUTLINE public_salaries
   FROM PRIVATE my_salaries;
```

CREATE PACKAGE

Purpose

Packages are defined using PL/SQL. Therefore, this section provides some general information but refers to *Oracle Database PL/SQL Language Reference* for details of syntax and semantics.

Use the `CREATE PACKAGE` statement to create the specification for a stored **package**, which is an encapsulated collection of related procedures, functions, and other program objects stored together in the database. The **package specification** declares these objects. The **package body**, specified subsequently, defines these objects.

See Also:

- CREATE PACKAGE BODY for information on specifying the implementation of the package
- CREATE FUNCTION and CREATE PROCEDURE for information on creating standalone functions and procedures
- ALTER PACKAGE and DROP PACKAGE for information on modifying and dropping a package
- *Oracle Database Development Guide* and *Oracle Database PL/SQL Packages and Types Reference* for detailed discussions of packages and how to use them

Prerequisites

To create or replace a package in your own schema, you must have the `CREATE PROCEDURE` system privilege. To create or replace a package in another user's schema, you must have the `CREATE ANY PROCEDURE` system privilege.

To embed a `CREATE PACKAGE` statement inside an Oracle Database precompiler program, you must terminate the statement with the keyword `END-EXEC` followed by the embedded SQL statement terminator for the specific language.

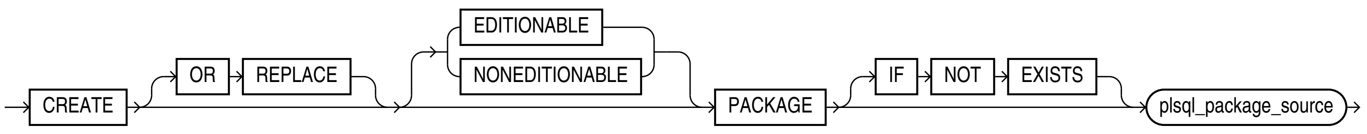
See Also:

Oracle Database PL/SQL Language Reference for more information

Syntax

Packages are defined using PL/SQL. Therefore, the syntax diagram in this book shows only the SQL keywords. Refer to *Oracle Database PL/SQL Language Reference* for the PL/SQL syntax, semantics, and examples.

***create_package*::=**



(*`plsql_package_source`*: See *Oracle Database PL/SQL Language Reference*.)

Semantics

OR REPLACE

Specify `OR REPLACE` to re-create the package specification if it already exists. Use this clause to change the specification of an existing package without dropping, re-creating, and regranting object privileges previously granted on the package. If you change a package specification, then Oracle Database recompiles it.

Users who had previously been granted privileges on a redefined package can still access the package without being regranted the privileges.

If any function-based indexes depend on the package, then the database marks the indexes `DISABLED`.

See Also:

`ALTER PACKAGE` for information on recompiling package specifications

IF NOT EXISTS

Specifying `IF NOT EXISTS` has the following effects:

- If the package does not exist, a new package is created at the end of the statement.
- If the package exists, this is the package you have at the end of the statement. A new one is not created because the older one is detected.

You can have *one* of `OR REPLACE` or `IF NOT EXISTS` in a statement at a time. Using both `OR REPLACE` with `IF NOT EXISTS` in the very same statement results in the following error: `ORA-11541: REPLACE and IF NOT EXISTS cannot coexist in the same DDL statement`.

Using `IF EXISTS` with `CREATE` results in `ORA-11543: Incorrect IF NOT EXISTS clause for CREATE statement`.

[EDITIONABLE | NONEDITIONABLE]

Use these clauses to specify whether the package is an editioned or noneditioned object if editioning is enabled for the schema object type `PACKAGE` in *`schema`*. The default is `EDITIONABLE`. For information about editioned and noneditioned objects, see *Oracle Database Development Guide*.

plsql_package_source

See *Oracle Database PL/SQL Language Reference* for the syntax and semantics of the *`plsql_package_source`*, including examples.

CREATE PACKAGE BODY

Purpose

Package bodies are defined using PL/SQL. Therefore, this section provides some general information but refers to *Oracle Database PL/SQL Language Reference* for details of syntax and semantics.

Use the `CREATE PACKAGE BODY` statement to create the body of a stored **package**, which is an encapsulated collection of related procedures, stored functions, and other program objects stored together in the database. The **package body** defines these objects. The **package specification**, defined in an earlier `CREATE PACKAGE` statement, declares these objects.

Packages are an alternative to creating procedures and functions as standalone schema objects.

See Also:

- CREATE FUNCTION and CREATE PROCEDURE for information on creating standalone functions and procedures
- CREATE PACKAGE for a discussion of packages, including how to create packages
- ALTER PACKAGE for information on modifying a package
- DROP PACKAGE for information on removing a package from the database

Prerequisites

To create or replace a package in your own schema, you must have the `CREATE PROCEDURE` system privilege. To create or replace a package in another user's schema, you must have the `CREATE ANY PROCEDURE` system privilege. In both cases, the package body must be created in the same schema as the package.

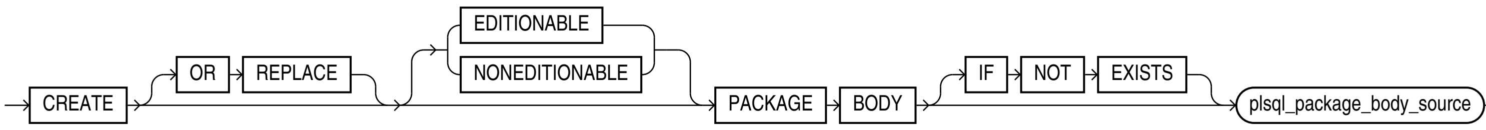
To embed a `CREATE PACKAGE BODY` statement inside an Oracle Database precompiler program, you must terminate the statement with the keyword `END-EXEC` followed by the embedded SQL statement terminator for the specific language.

> **See Also:**
>
> *Oracle Database PL/SQL Language Reference*

Syntax

Package bodies are defined using PL/SQL. Therefore, the syntax diagram in this book shows only the SQL keywords. Refer to *Oracle Database PL/SQL Language Reference* for the PL/SQL syntax, semantics, and examples.

***create_package_body*::=**



(*`plsql_package_body_source`*: See *Oracle Database PL/SQL Language Reference*.)

Semantics

OR REPLACE

Specify `OR REPLACE` to re-create the package body if it already exists. Use this clause to change the body of an existing package without dropping, re-creating, and regranting object privileges previously granted on it. If you change a package body, then Oracle Database recompiles it.

Users who had previously been granted privileges on a redefined package can still access the package without being regranted the privileges.

> **See Also:**
>
> ALTER PACKAGE for information on recompiling package bodies

IF NOT EXISTS

Specifying `IF NOT EXISTS` has the following effects:

- If the package body does not exist, a new package body is created at the end of the statement.
- If the package body exists, this is the package body you have at the end of the statement. A new one is not created because the older one is detected.

You can have *one* of `OR REPLACE` or `IF NOT EXISTS` in a statement at a time. Using both `OR REPLACE` with `IF NOT EXISTS` in the very same statement results in the following error: `ORA-11541: REPLACE and IF NOT EXISTS cannot coexist in the same DDL statement.`

Using `IF EXISTS` with `CREATE` results in `ORA-11543: Incorrect IF NOT EXISTS clause for CREATE statement`.

[EDITIONABLE | NONEDITIONABLE]

If you do not specify this clause, then the package body inherits `EDITIONABLE` or `NONEDITIONABLE` from the package specification. If you do specify this clause, then it must match that of the package specification.

plsql_package_body_source

See *Oracle Database PL/SQL Language Reference* for the syntax and semantics of the *`plsql_package_body_source`*.

CREATE PFILE

Purpose

Use the `CREATE PFILE` statement to export either a binary server parameter file or the current In-Memory parameter settings into a text initialization parameter file. Creating a text parameter file is a convenient way to get a listing of the current parameter settings being used by the database, and it lets you edit the file easily in a text editor and then convert it back into a server parameter file using the `CREATE SPFILE` statement.

Upon successful execution of this statement, Oracle Database creates a text parameter file on the server. In an Oracle Real Application Clusters environment, it will contain all parameter settings of all instances. It will also contain any comments that appeared on the same line with a parameter setting in the server parameter file.

Note on Creating Text Parameter Files in a CDB

When you create a text parameter file in a multitenant container database (CDB), the current container can be the root or a PDB.

- If the current container is the root, then the database creates a text file that contains the parameter settings for the root.
- If the current container is a PDB, then the database creates a text file that contains the parameter settings for the PDB. In this case you must specify a *`pfile_name`*.

See Also:

- CREATE SPFILE for information on server parameter files
- *Oracle Database Administrator's Guide* for additional information on text initialization parameter files and binary server parameter files
- *Oracle Real Application Clusters Administration and Deployment Guide* for information on using server parameter files in an Oracle Real Application Clusters environment

Prerequisites

You must have one of the following system privileges to execute this statement:

- `SYSDBA`

- `SYSDG`
- `SYSOPER`
- `SYSBACKUP`
- `SYSASM`
- `SYSRAC`

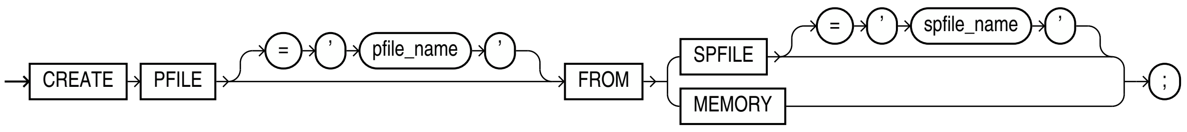
You can execute this statement either before or after instance startup.

Restrictions

You cannot overwrite OS files as a `SYSDG`, `SYSOPER`, or `SYSRAC` user.

Syntax

***create_pfile*::=**



Semantics

pfile_name

Specify the name of the text parameter file you want to create. If you do not specify *`pfile_name`*, then Oracle Database uses the platform-specific default initialization parameter file name. *`pfile_name`* can include a path prefix. If you do not specify such a path prefix, then the database adds the path prefix for the default storage location, which is platform dependent.

spfile_name

Specify the name of the binary server parameter from which you want to create a text file.

- If you specify *`spfile_name`*, then the file must exist on the server. If the file does not reside in the default directory for server parameter files on your operating system, then you must specify the full path.
- If you do not specify *`spfile_name`*, then the database uses the spfile that is currently associated with the instance, usually the one that was used a startup. If no spfile is associated with the instance, then the database looks for the platform-specific default server parameter file name. If that file does not exist, then the database returns an error.

See Also:

Creating and Configuring an Oracle Database

MEMORY

Specify `MEMORY` to create a pfile using the current system-wide parameter settings. In an Oracle RAC environment, the created file will contain the parameter settings from each instance.

Examples

Creating a Parameter File: Example

The following example creates a text parameter file `my_init.ora` from a binary server parameter file `s_params.ora`:

```
CREATE PFILE = 'my_init.ora' FROM SPFILE = 's_params.ora';
```

Note:

Typically you will need to specify the full path and filename for parameter files on your operating system. Refer to your Oracle operating system documentation for path information and default parameter file names.

CREATE PLUGGABLE DATABASE

Purpose

Use the `CREATE PLUGGABLE DATABASE` statement to create a pluggable database (PDB).

This statement enables you to perform the following tasks:

- Create a PDB by using the seed as a template

  Use the *`create_pdb_from_seed`* clause to create a PDB by using the seed in the multitenant container database (CDB) as a template. The files associated with the seed are copied to a new location and the copied files are then associated with the new PDB.
- Create a PDB by cloning an existing PDB

  Use the *`create_pdb_clone`* clause to create a PDB by copying an existing PDB and then plugging the copy into the CDB. The files associated with the existing PDB are copied to a new location and the copied files are associated with the new PDB.
- Create a PDB by plugging an unplugged PDB into a CDB

  Use the *`create_pdb_from_xml`* clause to plug an unplugged PDB into a CDB, using an XML metadata file.
- Create a proxy PDB by referencing another PDB. A proxy PDB provides fully functional access to the referenced PDB.

  Use the *`create_pdb_clone`* clause and specify `AS PROXY FROM` to create a proxy PDB.
- Create an application container, application seed, or application PDB

  Use the *`create_pdb_from_seed`*, *`create_pdb_clone`*, or *`create_pdb_from_xml clause`*. To create an application container, you must specify the `AS APPLICATION CONTAINER` clause. To create an application seed, you must specify the `AS SEED` clause.

Note:

A multitenant container database is the only supported architecture in Oracle Database 21c and later releases. While the documentation is being revised, legacy terminology may persist. In most cases, "database" and "non-CDB" refer to a CDB or PDB, depending on context. In some contexts, such as upgrades, "non-CDB" refers to a non-CDB from a previous release.

Note:

When a new PDB is established in a CDB, it is possible that the name of a service offered by the new PDB will collide with an existing service name. The namespace in which a collision can occur is that of the listener that gives access to the CDB. Within that namespace, collisions are possible among the names of CDB's default services, PDB's default services, and user-defined services. For example, if two or more CDBs on the same computer system use the same listener, and the newly established PDB has the same service name as another PDB in these CDBs, then a collision occurs.

When you create a PDB, you can specify new names for any potential colliding service names. See the clause *service_name_convert*. If you discover a service name collision after a PDB is created, you must not attempt to operate the PDB that causes a collision with an existing service name. If the colliding name is that of the PDB's default service, then you must rename the PDB. If the colliding name is that of a user-created service within the PDB, then you must drop that service and create one in its place, with a non-colliding name, that has the same purpose and properties.

See Also:

- *Oracle Multitenant Administrator’s Guide* for more information on multi-tenant architecture and concepts.
- ALTER PLUGGABLE DATABASE and DROP PLUGGABLE DATABASE for information on modifying and dropping PDBs

Prerequisites

You must be connected to a CDB. The CDB must be open and in `READ WRITE` mode.

To create a PDB or an application container, the current container must be the root and you must have the `CREATE PLUGGABLE DATABASE` system privilege, granted commonly.

To create an application seed or an application PDB, the current container must be an application root, the application container must be open and in `READ WRITE` mode, and you must have the `CREATE PLUGGABLE DATABASE` system privilege, either granted commonly or granted locally in that application container.

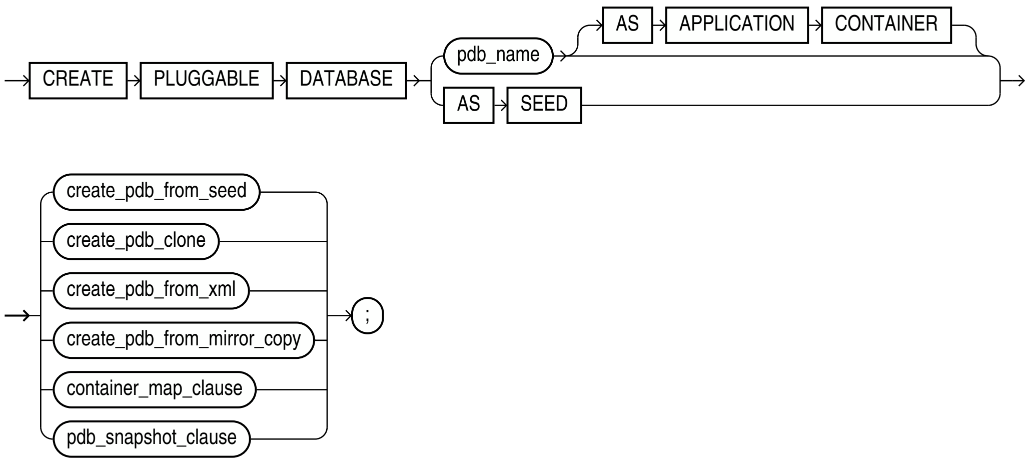
To specify the *`create_pdb_clone`* clause:

- If `src_pdb_name` refers to a PDB in the same CDB, then you must have the `CREATE PLUGGABLE DATABASE` system privilege in the root of the CDB in which the new PDB will be created and in the PDB being cloned.
- If `src_pdb_name` refers to a PDB in a remote database, then you must have the `CREATE PLUGGABLE DATABASE` system privilege in the root of the CDB in which the new PDB will be created. In addition, the remote user must have the `CREATE PLUGGABLE DATABASE` system privilege in the PDB to which `src_pdb_name` refers.

See *Oracle Multitenant Administrator's Guide* for more information on the prerequisites to PDB creation.

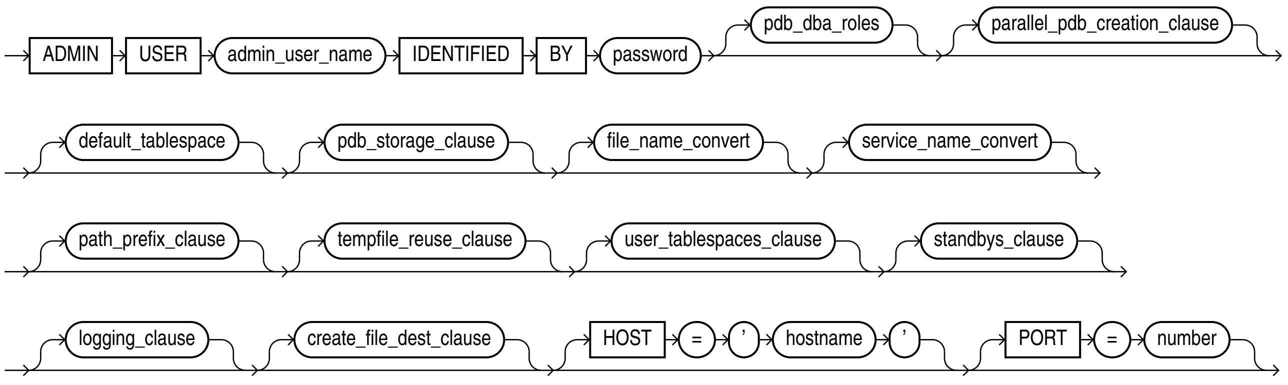
Syntax

***create_pluggable_database*::=**



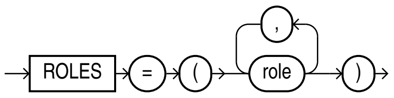
(*create_pdb_from_seed*::=, *create_pdb_clone*::=, *create_pdb_from_xml*::=)

***create_pdb_from_seed*::=**

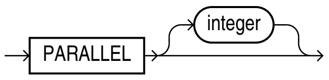


(*pdb_dba_roles*::=, *parallel_pdb_creation_clause*::=, *default_tablespace*::=, *file_name_convert*::=, *service_name_convert*::=, *pdb_storage_clause*::=, *path_prefix_clause*::=, *tempfile_reuse_clause*::=, *user_tablespaces_clause*::=, *standbys_clause*::=, *logging_clause*::=, *create_file_dest_clause*::=)

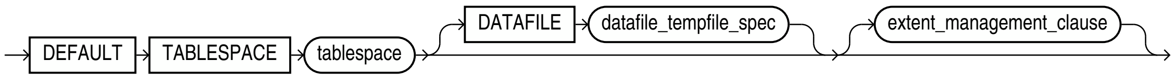
***pdb_dba_roles*::=**



***parallel_pdb_creation_clause*::=**

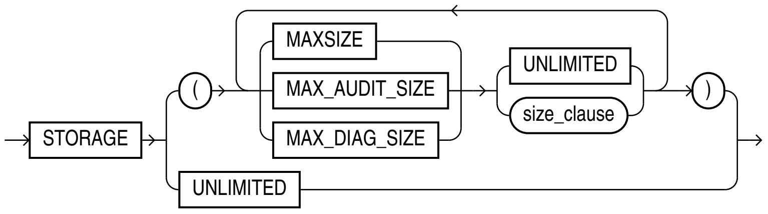


***default_tablespace*::=**



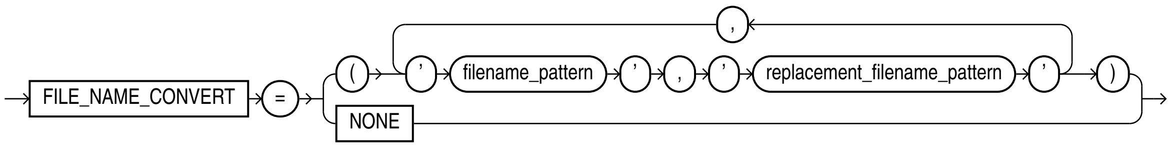
(*datafile_tempfile_spec*::=, *extent_management_clause*::=)

***pdb_storage_clause*::=**

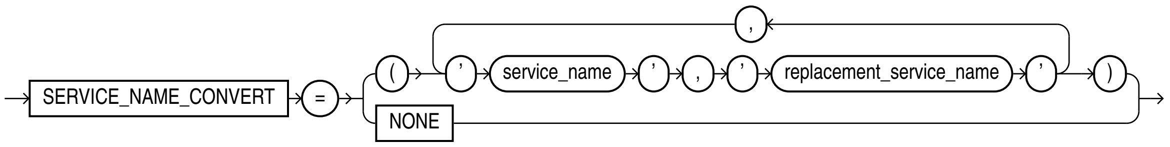


(*size_clause*::=)

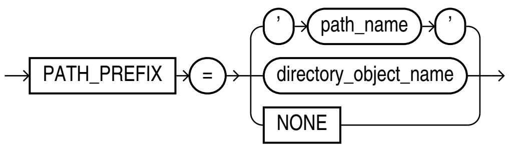
***file_name_convert*::=**



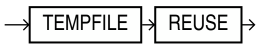
***service_name_convert*::=**



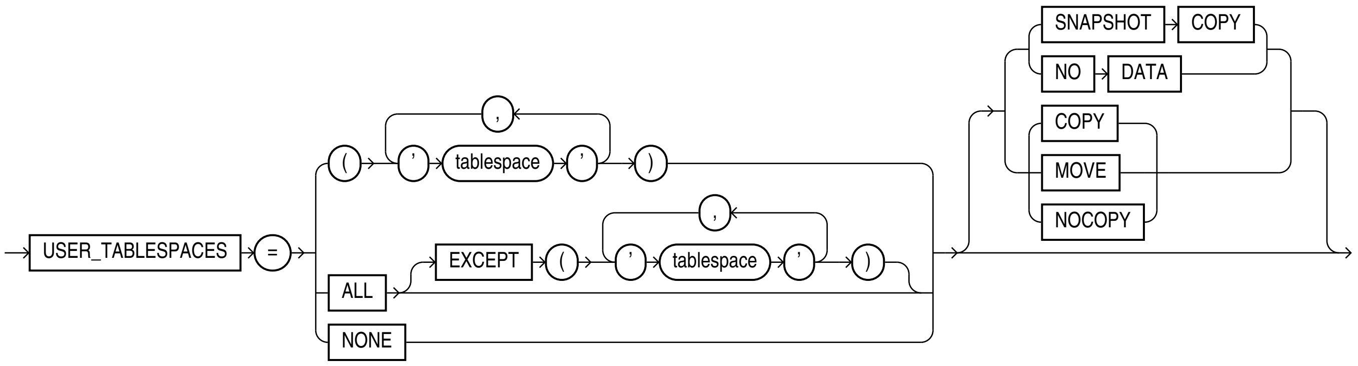
***path_prefix_clause*::=**



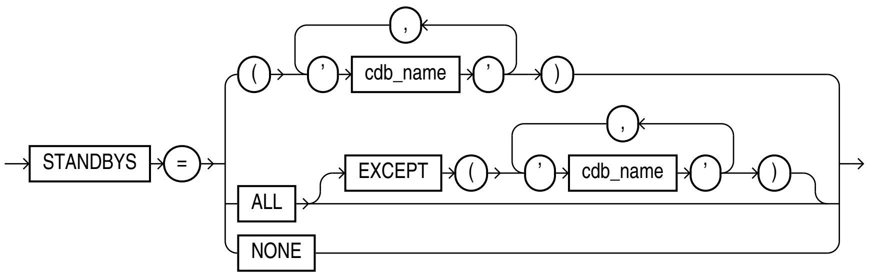
***tempfile_reuse_clause*::=**



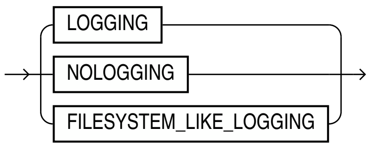
***user_tablespaces_clause*::=**



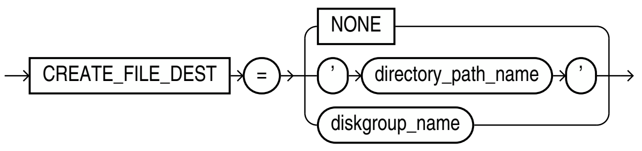
***standbys_clause*::=**



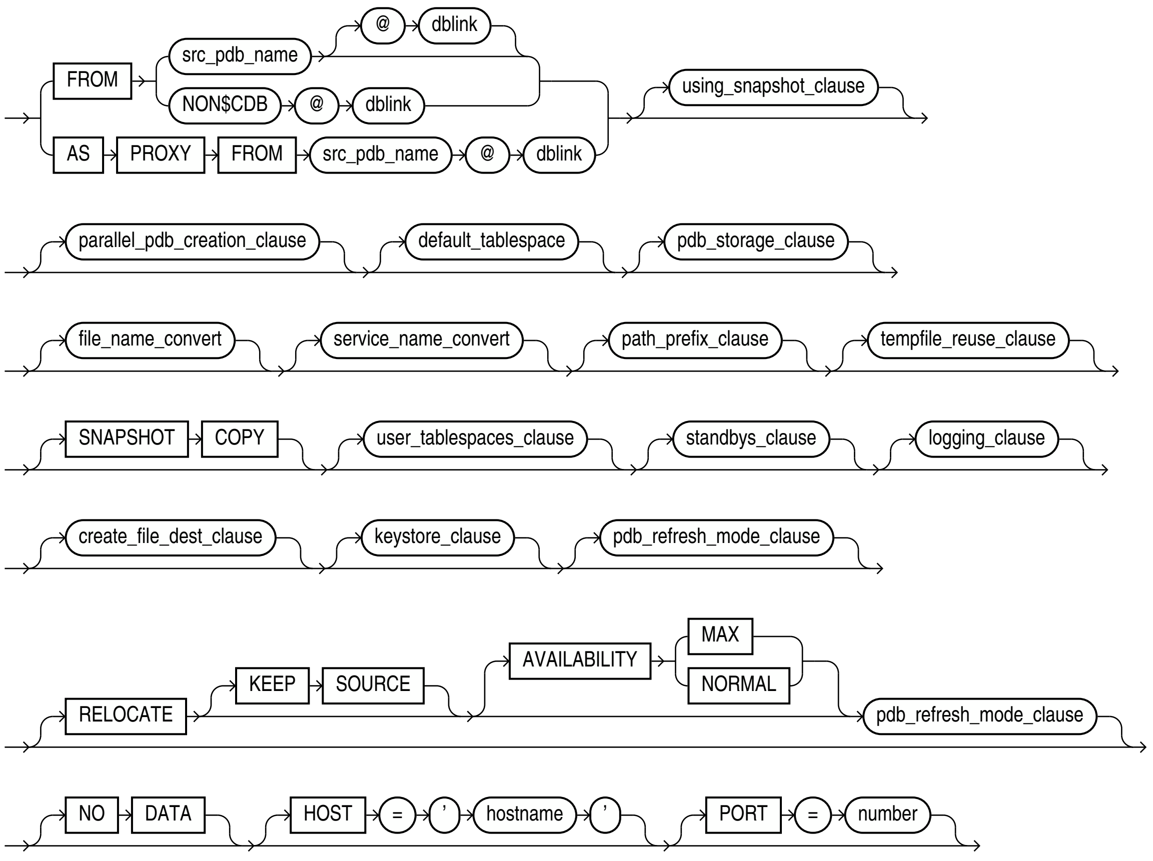
***logging_clause*::=**



***create_file_dest_clause*::=**

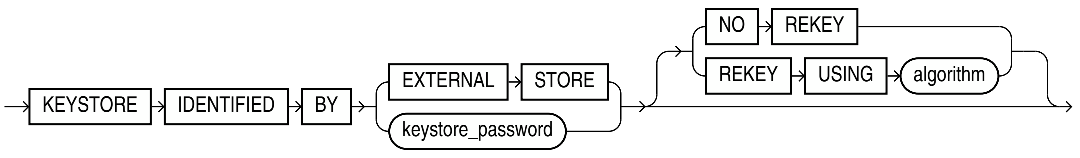


***create_pdb_clone*::=**

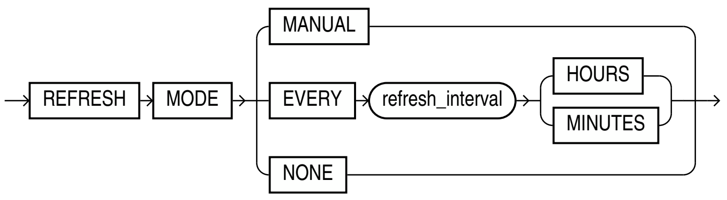


(*parallel_pdb_creation_clause*::=, *default_tablespace*::=, *pdb_storage_clause*::=, *file_name_convert*::=, *service_name_convert*::=, *path_prefix_clause*::=, *tempfile_reuse_clause*::=, *user_tablespaces_clause*::=, *standbys_clause*::=, *logging_clause*::=, *create_file_dest_clause*::=, *keystore_clause*::=, *pdb_refresh_mode_clause*::=)

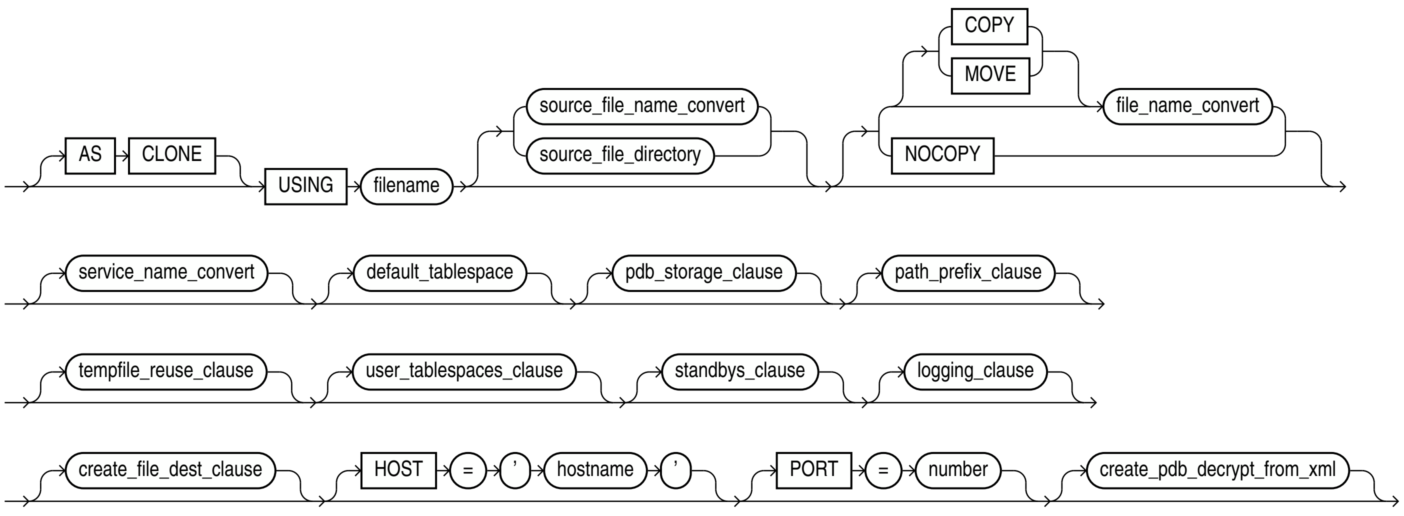
***keystore_clause*::=**



***pdb_refresh_mode_clause*::=**

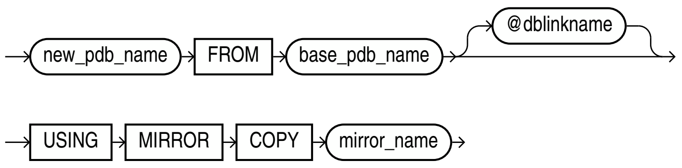


***create_pdb_from_xml*::=**

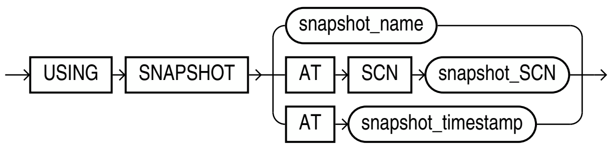


(*source_file_name_convert*::=, *source_file_directory*::=, *file_name_convert*::=, *service_name_convert*::=, *default_tablespace*::=, *pdb_storage_clause*::=, *path_prefix_clause*::=, *tempfile_reuse_clause*::=, *user_tablespaces_clause*::=, *standbys_clause*::=, *logging_clause*::=, *create_file_dest_clause*::=)

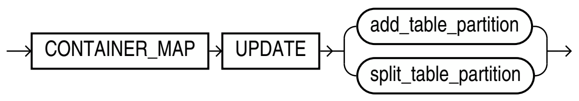
***create_pdb_from_mirror_copy*::=**



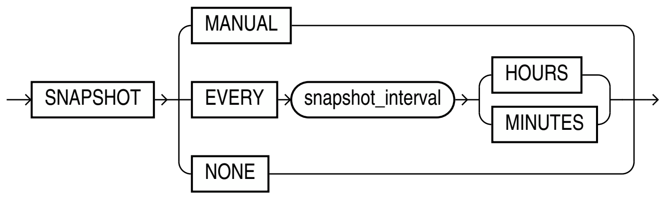
***using_snapshot_clause* ::=**



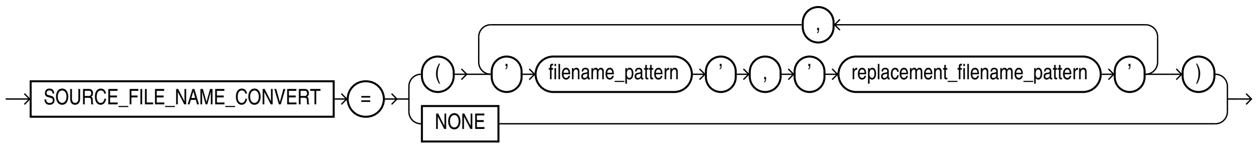
container_map_clause **::=**



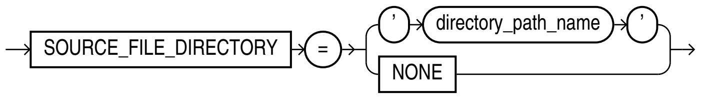
pdb_snapshot_clause **::=**



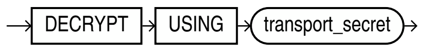
source_file_name_convert**::=**



source_file_directory**::=**



create_pdb_decrypt_from_xml**::=**



Semantics

pdb_name

Specify the name of the PDB to be created. The name must satisfy the requirements listed in "Database Object Naming Rules ". The first character of a PDB name must be an alphabet character. The remaining characters can be alphanumeric or the underscore character (_).

The PDB name must be unique in the CDB, and it must be unique within the scope of all the CDBs whose instances are reached through a specific listener.

AS APPLICATION CONTAINER

Specify this clause to create an application container.

> **See Also:**
>
> Creating and Removing Application Containers and Seeds

using_snapshot_clause

Specify this clause to create a PDB from an existing PDB snapshot that can be identified by its name, SCN, or timestamp.

If you additionally specify SNAPSHOT COPY, then the new PDB will depend on the existence of the specified PDB snapshot. This will affect your ability to drop or purge the new PDB.

AS SEED

Specify this clause to create an application seed. The database assigns the seed a name of the form `application_container_name$SEED`.

An application container can have at most one application seed. The application seed is optional, but, if it exists, you can use it to create application PDBs quickly that match the requirements of the application container. An application seed enables instant provisioning of application PDBs that are created from it.

> **See Also:**
>
> Creating and Removing Application Containers and Seeds

create_pdb_from_seed

This clause enables you to create a PDB by using the seed in the CDB as a template.

> **See Also:**
>
> Creating a PDB from Scratch

ADMIN USER

Use this clause to create an administrative user who can be granted the privileges required to perform administrative tasks on the PDB. For *`admin_user_name`*, specify name of the user to be created. Use the `IDENTIFIED BY` clause to specify the password for *`admin_user_name`*. Oracle Database creates a local user in the PDB and grants the `PDB_DBA` local role to that user.

pdb_dba_roles

This clause lets you grant one or more roles to the `PDB_DBA` role. Use this clause to grant roles that have the privileges required by the administrative user of the PDB. For *`role`*, specify a predefined role. For a list of predefined roles, refer to *Oracle Database Security Guide*.

You can also use the `GRANT` statement to grant roles to the `PDB_DBA` role after the PDB has been created. Until you have granted the appropriate privileges to the `PDB_DBA` role, the `SYS` and `SYSTEM` users can perform administrative tasks on a PDB.

parallel_pdb_creation_clause

This clause instructs the CDB to use parallel execution servers to copy the new PDB's data files to a new location. This may result in faster creation of the PDB.

PARALLEL

If you specify `PARALLEL`, then the CDB automatically chooses the number of parallel execution servers to use. This is the default if the `COMPATIBLE` initialization parameter is set to `12.2` or higher.

PARALLEL *integer*

Use integer to specify the number of parallel execution servers to use. The CDB can ignore this setting, depending on the current database load and the number of available parallel execution servers. If you specify a value of 0 or 1, then the CDB does not parallelize the creation of the PDB. This can result in a longer PDB creation time.

default_tablespace

If you specify this clause, then Oracle Database creates a smallfile tablespace and sets it as the default permanent tablespace for the PDB. Oracle Database will assign the default tablespace to any non-`SYSTEM` user for whom a different permanent tablespace is not specified. The *`default_tablespace`* clause has the same semantics that it has for the `CREATE DATABASE` statement. For full information, refer to *default_tablespace* in the documentation on `CREATE DATABASE`.

pdb_storage_clause

Use this clause to specify storage limits for the PDB.

- Use `MAXSIZE` to limit the amount of storage that can be used by all tablespaces in the PDB to the value specified with *`size_clause`*. This limit includes the size of data files and temporary files for tablespaces belonging to the PDB. Specify `MAXSIZE UNLIMITED` to enforce no limit.
- Use `MAX_AUDIT_SIZE` to limit the amount of storage that can be used by unified audit OS spillover (`.bin` format) files in the PDB to the value specified with *`size_clause`*. Specify `MAX_AUDIT_SIZE UNLIMITED` to enforce no limit.
- Use `MAX_DIAG_SIZE` to limit the amount of storage for diagnostics (trace files and incident dumps) in the Automatic Diagnostic Repository (ADR) that can be used by the PDB to the value specified with *`size_clause`*. Specify `MAX_DIAG_SIZE UNLIMITED` to enforce no limit.

If you omit this clause, or specify `STORAGE UNLIMITED`, then there are no storage limits for the PDB. This is equivalent to specifying `STORAGE (MAXSIZE UNLIMITED MAX_AUDIT_SIZE UNLIMITED MAX_DIAG_SIZE UNLIMITED)`.

file_name_convert

Use this clause to determine how the database generates the names of files (such as data files and wallet files) for the PDB.

- For *`filename_pattern`*, specify a string found in names of files associated with the seed (when creating a PDB by using the seed), associated with the source PDB (when cloning a PDB), or listed in the XML file (when plugging a PDB into a CDB).
- For *`replacement_filename_pattern`*, specify a replacement string.

Oracle Database will replace *`filename_pattern`* with *`replacement_filename_pattern`* when generating the names of files associated with the new PDB.

File name patterns cannot match files or directories managed by Oracle Managed Files.

You can specify `FILE_NAME_CONVERT = NONE`, which is the same as omitting this clause. If you omit this clause, then the database first attempts to use Oracle Managed Files to generate file names. If you are not using Oracle Managed Files, then the database uses the `PDB_FILE_NAME_CONVERT` initialization parameter to generate file names. If this parameter is not set, then an error occurs.

service_name_convert

Use this clause to rename the user-defined services of the new PDB based on the service names of the source PDB. When the service name of a new PDB conflicts with an existing service name in the CDB, plug-in violations can result. This clause enables you to avoid these violations.

- For *`service_name`*, specify the name of a service found in the PDB seed (when creating a PDB in an application container by using the application seed) or in the source PDB (when cloning a PDB or plugging a PDB into a CDB).
- For *`replacement_service_name`*, specify the replacement name for the service.

Oracle Database will use the replacement service name for the service in the PDB being created.

You can specify `SERVICE_NAME_CONVERT = NONE`, which is the same as omitting this clause.

Restrictions on *service_name_convert*

The *`service_name_convert`* clause is subject to the following restrictions:

- You cannot change the name of the default service for a PDB. The default service has the same name as the PDB.
- You cannot specify this clause when you use the *`create_pdb_from_seed`* clause to create a PDB from the CDB seed, because the CDB seed does not have user-defined services. You can, however, specify this clause when you use the *`create_pdb_from_seed`* clause to create an application PDB from the application seed.

path_prefix_clause

Use this clause to ensure that file paths for directory objects associated with the PDB are restricted to the specified directory or its subdirectories. This clause also ensures that the following files associated with the PDB are restricted to the specified directory: the Oracle XML repository for the PDB, files created with a `CREATE PFILE` statement, and the export directory for Oracle wallets. You cannot modify the setting of this clause after you create the PDB. This clause does not affect files created by Oracle Managed Files.

- For *path_name*, specify the absolute path name of an operating system directory. The single quotation marks are required, with the result that the path name is case sensitive. Oracle Database uses *path_name* as a prefix for all file paths associated with the PDB.

  Be sure to specify *path_name* so that the resulting path name will be properly formed when relative paths are appended to it. For example, on UNIX systems, be sure to end *path_name* with a forward slash (/), such as:

  ```
  PATH_PREFIX = '/disk1/oracle/dba/salespdb/'
  ```
- For *directory_object_name*, specify the name of a directory object that exists in the CDB root (CDB$ROOT). The directory object points to the absolute path to be used for PATH_PREFIX.
- If you specify PATH_PREFIX = NONE, then the relative paths for directory objects associated with the PDB are treated as absolute paths and are not restricted to a particular directory.

Omitting the *path_prefix_clause* is equivalent to specifying PATH_PREFIX = NONE.

After the *path_prefix_clause* is specified for a PDB, existing directory objects might not work as expected, since the PATH_PREFIX string is always added as a prefix to all local directory objects in the PDB. The *path_prefix_clause* only applies to user-created directory objects. It does not apply to Oracle-supplied directory objects.

tempfile_reuse_clause

When you create a PDB, Oracle Database associates temp files with the new PDB. Depending on how you create the PDB, the temp files may already exist and may have been previously used.

Specify TEMPFILE REUSE to instruct the database to format and reuse a temp file associated with the new PDB if it already exists. If you specify this clause and a temp file does not exist, then the database creates the temp file.

If you do not specify TEMPFILE REUSE and a temp file to be associated with the new PDB already exists, then the database returns an error and does not create the PDB.

user_tablespaces_clause

This clause lets you specify the tablespaces to be made available in the new PDB. The SYSTEM, SYSAUX, and TEMP tablespaces are available in all PDBs and cannot be specified in this clause.

You can use this clause to separate the data for multiple schemas into different PDBs.

- Specify *tablespace* to make the tablespace available in the new PDB. You can specify more than one tablespace in a comma-separated list.
- Specify ALL to make all tablespaces available in the new PDB. This is the default.
- Specify ALL EXCEPT to make all tablespaces available in the new PDB, except the specified tablespaces.
- Specify NONE to make only the SYSTEM, SYSAUX, and TEMP tablespaces available in the new PDB.

When the compatibility level of the CDB is 12.2 or higher, the tablespaces that are excluded by this clause are created offline in the new PDB, and they have no data files associated with them. When the compatibility level of the CDB is lower than 12.2, the tablespaces that are excluded by this clause are offline in the new PDB, and all data files that belong to these tablespaces are unnamed and offline.

{ SNAPSHOT COPY | NO DATA }

These clauses apply only when cloning a PDB with the `create_pdb_clone` clause. By default, the database creates each tablespace to be made available in the new PDB according to the settings specified for cloning the PDB. These clauses allow you to override those settings as follows:

- `SNAPSHOT COPY` - Clone the tablespace using storage snapshots.
- `NO DATA` - Clone the data model definition of the tablespace, but not the tablespace's data.

{ COPY | MOVE | NOCOPY }

These clauses apply when you plug in a PDB with the `create_pdb_from_xml` clause. By default, the database creates each tablespace to be made available in the new PDB according to the settings specified for plugging in the PDB. These clauses allow you to override those settings as follows:

- `COPY` - Copy the tablespace files to the new location.
- `MOVE` - Move the tablespace files to the new location.
- `NOCOPY` - Do not copy or move the tablespace files to the new location.

standbys_clause

Use this clause to specify whether the new PDB is included in one or more standby CDBs. If you include a PDB in a standby CDB, then during standby recovery the standby CDB will search for the data files for the PDB. If the data files are not found, then standby recovery will stop and you must copy the data files to the correct location before you can restart recovery.

- Specify `cdb_name` to include the new PDB in the specified standby CDB. You can specify more than one standby CDB name in a comma-separated list.
- Specify `ALL` to include the new PDB in all standby CDBs. This is the default.
- Specify `ALL EXCEPT` to include the new PDB in all standby CDBs, except the specified standby CDBs.
- Specify `NONE` to exclude the new PDB from all standby CDBs. When a PDB is excluded from all standby CDBs, the PDB's data files are unnamed and marked offline on all of the standby CDBs. Standby recovery will not stop if the data files for the PDB are not found on the standby. If you instantiate a new standby CDB after the PDB is created, then you must explicitly disable the PDB for recovery on the new standby CDB.

You can enable a PDB on a standby CDB after it was excluded on that standby CDB by copying the data files to the correct location, bringing the PDB online, and marking it as enabled for recovery.

logging_clause

Use this clause to specify the default logging attribute for tablespaces created within the PDB. The logging attribute controls whether certain DML operations are logged in the redo log file (`LOGGING`) or not (`NOLOGGING`).The default is `LOGGING`.

When creating a tablespace, you can override the default logging attribute by specifying the *logging_clause* of the `CREATE TABLESPACE` statement.

Refer to *logging_clause* for a full description of this clause.

create_file_dest_clause

By default, a newly created PDB inherits its Oracle Managed Files settings from the root. If the root uses Oracle Managed Files, then the PDB also uses Oracle Managed Files. The PDB shares the same base file system directory for Oracle Managed Files with the root and has its own subdirectory named with the GUID of the PDB. If the root does not use Oracle Managed Files, then the PDB also does not use Oracle Managed Files.

This clause lets you override the default behavior. You can enable or disable Oracle Managed Files for the PDB and you specify a different base file system directory or Oracle ASM disk group for the PDB's files.

- Specify `NONE` to disable Oracle Managed Files for the PDB.
- Specify either *`directory_path_name`* or *`diskgroup_name`* to enable Oracle Managed Files for the PDB.

  Specify *`directory_path_name`* to designate the base file system directory for the PDB's files. Specify the full path name of the operating system directory. The directory must exist and Oracle processes must have appropriate permissions on the directory. The single quotation marks are required, with the result that the path name is case sensitive.

  Specify *`diskgroup_name`* to designate the default Oracle ASM disk group for the PDB's files.

If you specify a value other than `NONE`, then the database implicitly sets the `DB_CREATE_FILE_DEST` initialization parameter with `SCOPE=SPFILE` in the PDB.

HOST and PORT

These clauses are useful only if you are creating a PDB that you plan to reference from a proxy PDB. This type of PDB is called a referenced PDB.

When creating a referenced PDB:

- If the name of the listener is different from the host name of the PDB, then you must specify the `HOST` clause. For *`hostname`*, specify the fully qualified domain name of the listener. Enclose *`hostname`* in single quotation marks. For example: `'myhost.example.com'`.

  In an Oracle Real Application Clusters (Oracle RAC) environment, you can specify for *`hostname`* any of the hosts for the PDB.
- If the port number of the listener is not 1521, then you must specify the `PORT` clause. For *`number`*, specify the port number for the listener.

A proxy PDB uses a database link to establish communication with its referenced PDB. After communication is established, the proxy PDB communicates directly with the referenced PDB without using a database link. The host name and port number of the listener for the referenced PDB must be correct for the proxy PDB to function properly.

See Also:

The clause AS PROXY FROM of *`create_pdb_clone`* for information on creating a proxy PDB

create_pdb_clone

This clause enables you to create a new PDB by cloning a source to a target PDB. The source can be a PDB in the local CDB, or a PDB in a remote CDB. The target PDB is the clone of the source.

If the source is a PDB in the local CDB, then the source PDB can be plugged in or unplugged. If the source is a PDB in a remote CDB, then the source PDB must be plugged in.

If the source is a PDB in a remote CDB, then the source and the CDB that contains the target PDB must meet the following requirements:

- They must have the same endian format.
- They must have compatible character sets and national character sets, which means:
  - Every character in the source character set is available in the local CDB character set.
  - Every character in the source character set has the same code point value in the local CDB character set.
- They must have the same set of database options installed.

Users in the PDB who used the default temporary tablespace of the source PDB use the default temporary tablespace of the new PDB. Users who used non-default temporary tablespaces in the PDB continue to use the same local temporary tablespaces in the new PDB.

You can clone a united PDB or an isolated PDB with the same command. The only difference is that the keystore password you must provide are for different keystores.

Hot Clone a PDB: Example

```
CREATE PLUGGABLE DATABASE CDB1_PDB2_CLONE FROM CDB1_PDB2
 KEYSTORE IDENTIFIED BY keystore_password
```

For a **united** PDB:

- `keystore_password` is the `ROOT` keystore password.
- The wallet must be open in `ROOT`.

For an **isolated** PDB:

- `keystore_password` is the **new** keystore password for the PDB `CDB1_PDB2_CLONE`.
- The wallet must be open in `CDB1_PDB2_CLONE`.

Clone a PDB: Example

United PDB

```
CREATE PLUGGABLE DATABASE CDB1_PDB1_C AS CLONE USING '/tmp/cdb1_pdb3.pdb'
 KEYSTORE IDENTIFED BY keystore_password DECRYPT USING transport_secret
```

- The wallet must be open in `ROOT`, if TDE is in use.
- If there are TDE keys in the `.pdb` file, you must specify `KEYSTORE IDENTIFED BY` and provide *`transport_secret`*.
- *`keystore_password`* is the `ROOT` keystore password.

Isolated PDB

```
CREATE PLUGGABLE DATABASE CDB1_PDB2_C AS CLONE USING '/tmp/cdb1_pdb2.pdb'
```

- You need not specify `KEYSTORE IDENTIFED BY` or *`transport_secret`*. If specified, they are ignored.
- The wallet need *not* be open in `ROOT`.

> **See Also:**
>
> Cloning a PDB

FROM

Use this clause to specify the source PDB. The files associated with the source are copied to a new location and these copied files are then associated with the new PDB.

The source PDB cannot be closed. It can be open as follows:

- If the CDB that contains the source PDB (the source CDB) is in `ARCHIVELOG` mode and local undo mode, then the source PDB can be open in `READ WRITE` mode and fully functional during the cloning operation. This is called hot PDB cloning.
- If the source CDB is not in `ARCHIVELOG` mode, then the source PDB must be open `READ ONLY`.

Specify the source PDBas follows:

- If the source is a PDB in the local CDB, then use *`src_pdb_name`* to specify the name of the source PDB. You cannot specify `PDB$SEED` for *`src_pdb_name`*. Instead, use the *create_pdb_from_seed* clause to create a PDB by using the seed as a template.
- If the source is a PDB in a remote CDB, then use *`src_pdb_name`* to specify the name of the source PDB and *`dblink`* to specify the name of the database link to use to connect to the remote CDB.

AS PROXY FROM

Use this clause to create a proxy PDB by referencing a different PDB, which is referred to as the referenced PDB. The referenced PDB can be in the same CDB as the proxy PDB or in a different CDB. A local proxy PDB is in the same CDB as its referenced PDB, and a remote proxy PDB is in a different CDB than its referenced PDB.

For *`src_pdb_name@dblink`*, specify the referenced PDB.

> **See Also:**
>
> Creating a PDB as a Proxy PDB

default_tablespace

Use this clause to specify a permanent default tablespace for the PDB. Oracle Database will assign the default tablespace to any non-`SYSTEM` user for whom a different permanent tablespace is not specified. The tablespace must already exist in the source PDB. Because the tablespace already exists, you cannot specify the `DATAFILE` clause or the *`extent_management_clause`* when creating a PDB with the *`create_pdb_clone`* clause.

pdb_storage_clause

Use this clause to specify storage limits for the new PDB. Refer to *pdb_storage_clause* for the full semantics of this clause.

file_name_convert

Use this clause to determine how the database generates the names of files for the new PDB. Refer to *file_name_convert* for the full semantics of this clause.

service_name_convert

Use this clause to determine how the database renames services for the new PDB. Refer to *service_name_convert*::= for the full semantics of this clause.

path_prefix_clause

Use this clause to ensure that all directory object paths associated with the PDB are restricted to the specified directory or its subdirectories. Refer to *path_prefix_clause* for the full semantics of this clause.

tempfile_reuse_clause

Specify `TEMPFILE REUSE` to instruct the database to format and reuse a temp file associated with the new PDB if it already exists. Refer to *tempfile_reuse_clause* for the full semantics of this clause.

SNAPSHOT COPY

You can specify `SNAPSHOT COPY` only when cloning a PDB. The source PDB can be in the local CDB or a remote CDB. The `SNAPSHOT COPY` clause instructs the database to clone the source PDB using storage snapshots. This reduces the time required to create the clone because the database does not need to make a complete copy of the source data files.

When you use the `SNAPSHOT COPY` clause to create a clone of a source PDB and the `CLONEDB` initialization parameter is set to `FALSE`, the underlying file system for the source PDB's files must support storage snapshots. Such file systems include Oracle Advanced Cluster File System (Oracle ACFS) and Direct NFS Client storage.

When you use the `SNAPSHOT COPY` clause to create a clone of a source PDB and the `CLONEDB` initialization parameter is set to `TRUE`, the underlying file system for the source PDB's files can be any local file system, network file system (NFS), or clustered file system that has Direct NFS enabled. However, the source PDB must remain in open read-only mode as long as any clones exist.

Direct NFS Client enables an Oracle database to access network attached storage (NAS) devices directly, rather than using the operating system kernel NFS client. If the PDB files are stored on Direct NFS Client storage, then the following additional requirements must be met:

- The source PDB files must be located on an NFS volume.
- Storage credentials must be stored in a Transparent Data Encryption keystore.
- The storage user must have the privileges required to create and destroy snapshots on the volume that hosts the source PDB files.
- Credentials must be stored in the keystore using an `ADMINISTER KEY MANAGEMENT ADD SECRET` SQL statement.

When you use the `SNAPSHOT COPY` clause to create a clone of a source PDB, the following restrictions apply to the source PDB as long as any clones exist:

- It cannot be unplugged.
- It cannot be dropped.

PDB clones created using the `SNAPSHOT COPY` clause cannot be unplugged. They can only be dropped. Attempting to unplug a clone created using the `SNAPSHOT COPY` clause results in an error.

For a PDB created using the `SNAPSHOT COPY` clause in an Oracle Real Application Clusters (Oracle RAC) environment, each node that must access the PDB's files must be mounted. For Oracle RAC databases running on Linux or UNIX platforms, the underlying NFS volumes must be mounted. If the Oracle RAC database is running on a Windows platform and using Direct NFS for shared storage, then you must update the `oranfstab` file on all nodes with the created volume `export` and `mount` entries.

Storage clones are named and tagged using the new PDB GUID. You can query the `CLONETAG` column of `DBA_PDB_HISTORY` view to view clone tags for storage clones.

keystore_clause

Specify this clause if the source database has encrypted data or a keystore set.

If you want to create the PDB by cloning another PDB, and if the source database has encrypted data or a TDE master encryption key that has been set, then you must provide the keystore password of the target keystore in *`keystore_password`* .

You can find if the source database has encrypted data by querying the `DBA_ENCRYPTED_COLUMNS` data dictionary view or the `V$ENCRYPTED_TABLESPACES` dynamic performance view.

You can use the `EXTERNAL STORE` clause instead of *`keystore_password`* to clone a PDB that is using a united keystore. Note that you must configure the TDE SEPS wallet first before you use this option.

You cannot use the `EXTERNAL STORE` clause for a PDB that is using an isolated keystore.

You can set the password to a maximum length of 1024 bytes.

pdb_refresh_mode_clause

The `REFRESH MODE` clause applies only when cloning a PDB. The source PDB must be in a remote CDB, that is, you must specify the source PDB using the `FROM` *`src_pdb_name@dblink`* clause.

This clause lets you specify the refresh mode of the PDB. You can use this clause to create a **refreshable PDB**. Changes in the source PDB can be propagated to the refreshable PDB, either manually or automatically. This operation is called a refresh. You can specify the following refresh modes:

- `MANUAL` - This mode allows you to refresh the refreshable PDB manually at any time by issuing an `ALTER PLUGGABLE DATABASE REFRESH` statement.
- `EVERY` *`refresh_interval`* `MINUTES` or `HOURS` – This mode instructs the database to refresh the refreshable PDB every *`refresh_interval`* of selected time units, minutes or hours. If you select `MINUTES`, the `refresh_interval` must be less than 3000. If you select `HOURS`, the `refresh_interval` must be less than 2000. This mode also allows you to refresh the PDB manually at any time by issuing an `ALTER PLUGGABLE DATABASE REFRESH` statement.

- `NONE` - If you specify this mode, then the clone PDB is not a refreshable PDB. The database cannot refresh the PDB automatically and you cannot refresh the PDB manually. If you specify this mode, then you cannot later change the PDB into a refreshable PDB. This is the default.

A refreshable PDB can be opened only in `READ ONLY` mode. A refreshable PDB must be closed in order for a refresh to occur. If it is not closed when you attempt to perform a manual refresh, then an error will occur. If it is not closed when the database attempts an automatic refresh, then the refresh will be deferred until the next scheduled refresh.

See Also:

- `ALTER PLUGGABLE DATABASE` REFRESH for information on refreshing a PDB manually
- `ALTER PLUGGABLE DATABASE` *pdb_refresh_mode_clause* for information on changing the refresh mode of a PDB
- *Oracle Database Administrator's Guide* for more information on refreshable PDBs

RELOCATE

Use this clause to relocate a PDB from one CDB to another. The database first clones the source PDB to the target PDB, and then removes the source PDB. The database also moves the files associated with the PDB to a new location. This operation is the fastest way to relocate a PDB with minimal down time. The down time for the PDB is approximately the time required to copy the PDB's files from their old location to their new location. The source PDB can be open in `READ WRITE` mode and fully functional during the relocation operation.

Specify `REFRESH MODE` to keep the PDB current during the relocate process.

You can specify the availability level with the `AVAILABILITY` keyword. The default availability is `NORMAL`. If you specify `AVAILABILITY MAX`, then additional operations are performed to ensure a smooth migration of the workload in a persistent connection between source and target.

In the *`create_pdb_clone`* clause, you must use the `FROM` *`src_pdb_name@dblink`* syntax to identify the location of the source PDB. For *`src_pdb_name`*, specify the name of the source PDB. For *`dblink`*, specify a database link that indicates the location of the source PDB. The database link must have been created in the CDB to which the PDB will be relocated. It can connect either to the root of the remote CDB or to the remote PDB.

KEEP SOURCE

Specify `KEEP SOURCE` if you want to keep the source PDB and preserve it in an unplugged state.

See Also:

Relocating a PDB

NO DATA

The NO DATA clause applies only when cloning a PDB. This clause specifies that the source PDB's data model definition is cloned, but not the PDB's data. The dictionary data in the source PDB is cloned, but all user-created table and index data from the source PDB is discarded.

Restrictions on the NO DATA Clause

The following restrictions apply to the NO DATA clause:

- The source PDB should be open in read only mode when you use the NO DATA clause to clone a PDB.
- You cannot specify NO DATA if the source PDB contains clustered tables, Advanced Queuing (AQ) tables, index-organized tables, or tables that contain abstract data type columns.

HOST and PORT

These clauses are useful only if you are creating a PDB that you plan to reference from a proxy PDB. This type of PDB is called a referenced PDB. Refer to HOST and PORT for the full semantics of these clauses.

create_pdb_from_xml

This clause enables you to create a PDB by plugging an unplugged PDB (the source database) into a CDB (the target CDB). If the source database is an unplugged PDB, then it may have been unplugged from the target CDB or a different CDB.

The source database and the target CDB must meet the following requirements:

- They must have the same endian format.
- They must have compatible character sets and national character sets, which means:
  - Every character in the source database character set is available in the target CDB character set.
  - Every character in the source database character set has the same code point value in the target CDB character set.
- They must have the same set of database options installed.

See Also:

- Plugging In an Unplugged PDB
- *Oracle Database PL/SQL Packages and Types Reference* for more information on the DBMS_PDB package

AS CLONE

Specify this clause only if the target CDB already contains a PDB that was created using the same set of data files. The source files remain as an unplugged PDB and can be used again. Specifying AS CLONE also ensures that Oracle Database generates new identifiers, such as DBID and GUID, for the new PDB.

USING

This clause lets you specify a file that contains information about the source database that your are plugging in. For *filename*, specify the full path name of the file. You can obtain this file in one of the following ways:

- If the source database is an unplugged PDB, then the file was created by the *pdb_unplug_clause* of `ALTER PLUGGABLE DATABASE` as follows:
  - If the filename ends with the extension .xml, then it is an XML file containing metadata about the PDB. In this case, you must ensure that the XML metadata file, as well as the PDB's data files, are in a location that is accessible to the CDB.
  - If the filename ends with the extension .pdb, then it is a PDB archive file. This is a compressed file that includes an XML file containing metadata about the PDB, as well as the PDB's data files. The PDB archive file must exist in a location that is accessible to the CDB. When you use a .pdb archive file, this file is extracted when you plug in the PDB, and the PDB’s files are placed in the same directory as the .pdb archive file. Therefore, the *source_file_directory* clause is not required.
- If the source database is a non-CDB, then you must create the XML metadata file using the `DBMS_PDB` package, and ensure that the XML metadata file, as well as the source non-CDB's data files, are in a location that is accessible to the CDB.

See Also:

- *pdb_unplug_clause* of `ALTER PLUGGABLE DATABASE`
- *Oracle Database PL/SQL Packages and Types Reference* for more information on the `DBMS_PDB` package

source_file_name_convert

Specify this clause only if the contents of the XML file do not accurately describe the locations of the source files. If the files that must be used to plug in the source database are no longer in the location specified in the XML file, then use this clause to map the specified file names to the actual file names.

- For *filename_pattern*, specify the string for the location of the files as specified in the XML file.
- For *replacement_filename_pattern*, specify the string for the actual location that contains the files that must be used to create the PDB.

Oracle Database will replace *filename_pattern* with *replacement_filename_pattern* when searching for the source database files.

File name patterns cannot match files or directories managed by Oracle Managed Files.

If the files that must be used to create the PDB exist in the location specified in the XML file, you can either omit this clause or specify `SOURCE_FILE_NAME_CONVERT=NONE`.

source_file_directory

Specify this clause only if the contents of the XML file do not accurately describe the locations of the source files *and* the source files are all present in a single directory. This clause is convenient when you have a large number of data files and specifying a replacement file name pattern for each file using the *source_file_name_convert* clause is not feasible.

- For *directory_path_name*, specify the absolute path of the directory that contains the source files. The directory is scanned to find the appropriate files based on the unplugged PDB's XML file.

You can specify this clause for configurations that use Oracle Managed Files and for configurations that do not use Oracle Managed Files.

If the files that must be used to create the PDB exist in the location specified in the XML file, you can either omit this clause or specify `SOURCE_FILE_DIRECTORY=NONE`.

COPY

Specify `COPY` if you want the files listed in the XML file to be copied to the new location and used for the new PDB. This is the default. You can use the optional *file_name_convert* clause to use pattern replacement in the new file names. Refer to *file_name_convert* for the full semantics of this clause.

MOVE

Specify `MOVE` if you want the files listed in the XML file to be moved, rather than copied, to the new location and used for the new PDB. You can use the optional *file_name_convert* clause to use pattern replacement in the new file names. Refer to *file_name_convert* for the full semantics of this clause.

If the storage locations are different mounts, or if the storage locations do not support move at the OS or storage level, then the `MOVE` clause first copies the files then deletes the originals.

NOCOPY

Specify `NOCOPY` if you want the files for the PDB to remain in their current locations. Use this clause if there is no need to copy or move the files required to plug in the PDB.

service_name_convert

Use this clause to determine how the database renames services for the new PDB. Refer to *service_name_convert*::= for the full semantics of this clause.

default_tablespace

Use this clause to specify a permanent default tablespace for the PDB. Oracle Database will assign the default tablespace to any non-`SYSTEM` user for whom a different permanent tablespace is not specified. The *tablespace* must already exist in the source database. Because the tablespace already exists, you cannot specify the `DATAFILE` clause or the *extent_management_clause* when creating a PDB with the *create_pdb_from_xml* clause.

pdb_storage_clause

Use this clause to specify storage limits for the new PDB. Refer to *pdb_storage_clause* for the full semantics of this clause.

path_prefix_clause

Use this clause to ensure that all directory object paths associated with the PDB are restricted to the specified directory or its subdirectories. Refer to *path_prefix_clause* for the full semantics of this clause.

tempfile_reuse_clause

Specify `TEMPFILE REUSE` to instruct the database to format and reuse a temp file associated with the new PDB if it already exists. Refer to *tempfile_reuse_clause* for the full semantics of this clause.

HOST and PORT

These clauses are useful only if you are creating a PDB that you plan to reference from a proxy PDB. This type of PDB is called a referenced PDB. Refer to HOST and PORT for the full semantics of these clauses.

create_pdb_from_mirror_copy

Specify this clause to create a pluggable database *`new_pdb_name`* using the prepared files of the mirror copy *`mirror_name`*. The new PDB will be split from the source database using the prepared files created by the *`prepare_clause`*.

- You must execute this clause from the root container.
- The meaning of the other optional parameters remains unchanged by this clause.
- You can only split one database from a prepared mirror copy. If you want to create additional splits, you must prepare a new mirror copy.
- You can specify the database link name after you have specifed the mirror copy name in the *`prepare_clause`* of the `ALTER PLUGGABLE DATABASE` statement. In addition, the current CDB name should match the target CDB name specified in the *`prepare_clause`*. You must be a valid user in the CDB being referenced by the database link with the system privileges `CREATE SESSION` and `CREATE PLUGGABLE DATABASE`.
- If the database link name is omitted, then the base PDB name is looked up in the current CDB.

using_snapshot_clause

Specify this clause to create a PDB using an existing PDB snapshot that can be identified by its name, SCN, or timestamp.

If you create a PDB specifying `SNAPSHOT COPY`, then the new PDB will depend on the existence of the PDB snapshot. This will affect your ability to drop or purge the PDB.

container_map_clause

Specify this clause in CDB Root, Application Root or both to dynamically update changes as they happen to the new PDB.

You must note the following points with container maps:

- The `container_map_clause` is optional.
- The `add_partition_clause` will add a new partition to the container map defined in the Root (CDB Root and/or Application Root) of the new PDB.
- The `split_partition_clause` will split an existing partition of the container map defined in the Root (CDB Root and/or Application Root) of the new PDB.
- In the absence of `add_partition_clause` and `split_partition_clause`, container map defined in the Root of the new PDB is not updated.

- For PDB relocate, container map defined in the Root (CDB Root and/or Application Root) of the source PDB are automatically updated to reflect the “drop” of the source PDB.
- Dynamic maintenance of container map defined using hash partitioning is not supported

Add a New Partition to a Range-Partitioned Container Map: Example

```
CREATE PLUGGABLE DATABASE cdb1_pdb3
  ADMIN USER IDENTIFIED BY manager
  FILE_NAME_CONVERT=('cdb1_pdb0, cdb1_pdb3')
  CONTAINER_MAP UPDATE (ADD PARTITION cdb1_pdb3 VALUES LESS THAN (100));
  ALTER PLUGGABLE DATABASE cdb1_pdb3 OPEN
```

Split an Existing Partition of a Range-Partitioned Container Map to Create a New Partition: Example

```
CREATE PLUGGABLE DATABASE cdb1_pdb4
  ADMIN USER IDENTIFIED BY manager
  FILE_NAME_CONVERT=('cdb1_pdb0, cdb1_pdb4')
  CONTAINER_MAP UPDATE (SPLIT PARTITION cdb1_pdb3
                        AT (50)
                        INTO
                        (PARTITION cdb1_pdb3, PARTITION cdb1_pdb3)
  ALTER PLUGGABLE DATABASE cdb1_pdb4 OPEN
```

Verify Updated in Range-Partitioned Container Map : Example

```
SELECT partition_name, high_value
  FROM dba_tab_partitions
  WHERE table_name='MAP' AND table_owner='SYS'
```

pdb_snapshot_clause

Specify this clause if you want to be able to create PDB snapshots.

- `NONE` is the default. It means that no snapshots of the PDB can be created.
- `MANUAL` means that the PDB snapshot can *only* be created manually.
- If snapshot interval is specified, PDB snapshots will be created automatically at specified interval. In addition, a user will also be able to create PDB snapshots manually
- If expressed in minutes, `snapshot_interval` must be less than 3000.
- If expressed in hours, `snapshot_interval` must be less than 2000.

create_pdb_decrypt_from_xml

You must have the `SYSKM` privilege to execute this command.

For PDBs in **united** mode, the following restrictions apply:

- You must specify the clause if you are using a TDE protected database. Otherwise it is optional.
- You need not specify the clause for an isolated PDB.
- The wallet must be open in `ROOT`.
- The wallet file is copied in all cases: `NOCOPY`, `COPY`, and `MOVE`.

Plugging a PDB from an XML Metadata File: Example

```
CREATE PLUGGABLE DATABASE CDB1_PDB2 USING '/tmp/cdb1_pdb2.xml' NOCOPY
KEYSTORE IDENTIFIED BY keystore_password DECRYPT USING transport_secret
```

Plugging a PDB from an Archive File: Example

```
CREATE PLUGGABLE DATABASE CDB1_PDB1_1_C USING '/tmp/cdb1_pdb3.pdb' DECRYPT USING
transport_secret
```

For PDBs in **isolated** mode, you need not specify `DECRYPT USING transport_secret`. This is not required because the wallet file is copied during the creation of an unplugged PDB from an XML file. if you are creating a PDB from an archive file with the `.pdb` extension, the wallet file of the PDB is available in the zipped archive.

If the `ewallet.p12` file already exists at the destination, a backup is automatically initiated. The backup file has the following format: `ewallet_PLGDB_2017090517455564.p12`.

Examples

Creating a PDB by Using the Seed: Example

The following statement creates a PDB `salespdb` by using the seed in the CDB as a template. The administrative user `salesadm` is created and granted the `dba` role. The default tablespace assigned to any non-`SYSTEM` users for whom no permanent tablespace is assigned is `sales`. File names for the new PDB will be constructed by replacing `/disk1/oracle/dbs/pdbseed/` in the file names in the seed with `/disk1/oracle/dbs/salespdb/`. All tablespaces that belong to `sales` must not exceed 2G. The location of all directory object paths associated with `salespdb` are restricted to the directory `/disk1/oracle/dbs/salespdb/`.

```
CREATE PLUGGABLE DATABASE salespdb
  ADMIN USER salesadm IDENTIFIED BY password
  ROLES = (dba)
  DEFAULT TABLESPACE sales
    DATAFILE '/disk1/oracle/dbs/salespdb/sales01.dbf' SIZE 250M AUTOEXTEND ON
  FILE_NAME_CONVERT = ('/disk1/oracle/dbs/pdbseed/',
                       '/disk1/oracle/dbs/salespdb/')
  STORAGE (MAXSIZE 2G)
  PATH_PREFIX = '/disk1/oracle/dbs/salespdb/';
```

Cloning a PDB From an Existing PDB: Example

The following statement creates a PDB `newpdb` by cloning PDB `salespdb`. PDBs `newpdb` and `salespdb` are in the same CDB. Because no storage limits are explicitly specified, there is no limit on the amount of storage for `newpdb`. The files are copied from `/disk1/oracle/dbs/salespdb/` to `/disk1/oracle/dbs/newpdb/`. The location of all directory object paths associated with `newpdb` are restricted to the directory `/disk1/oracle/dbs/newpdb/`.

```
CREATE PLUGGABLE DATABASE newpdb FROM salespdb
  FILE_NAME_CONVERT = ('/disk1/oracle/dbs/salespdb/', '/disk1/oracle/dbs/newpdb/')
  PATH_PREFIX = '/disk1/oracle/dbs/newpdb';
```

Plugging a PDB into a CDB: Example

The following statement plugs the PDB `salespdb`, which was previously unplugged, into the CDB. The details about the metadata describing `salespdb` are stored in the XML file `/disk1/usr/salespdb.xml`. The XML file does not accurately describe the current locations of the files. Therefore, the `SOURCE_FILE_NAME_CONVERT` clause is used to indicate that the files are in `/disk2/oracle/dbs/salespdb/`, not `/disk1/oracle/dbs/salespdb/`. The `NOCOPY` clause indicates that the files are already in the correct location. All tablespaces that belong to `sales` must not exceed 2G. A file with the same name as the temp file specified in the XML file exists in the target location. Therefore, the `TEMPFILE REUSE` clause is required.

```
CREATE PLUGGABLE DATABASE salespdb
  USING '/disk1/usr/salespdb.xml'
```

```
SOURCE_FILE_NAME_CONVERT =
  ('/disk1/oracle/dbs/salespdb/', '/disk2/oracle/dbs/salespdb/')
NOCOPY
STORAGE (MAXSIZE 2G)
TEMPFILE REUSE;
```

CREATE PMEM FILESTORE

Purpose

You can create a persistent memory file store with this statement.

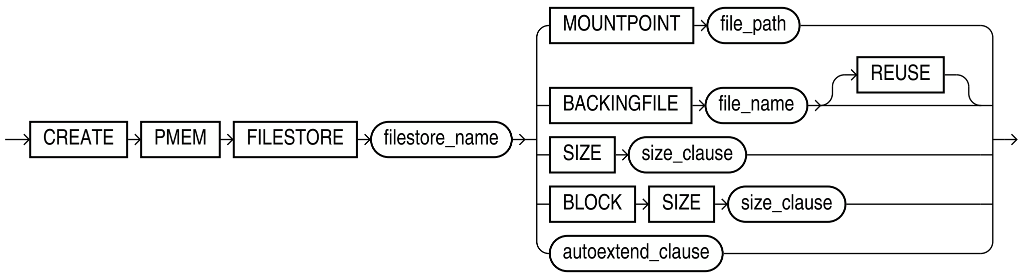
Prerequistes

You must have `SYSDBA` privileges to execute `CREATE PMEM FILESTORE` .

You must execute this statement from `CDB$ROOT`.

Syntax

***create_pmem_filestore*::=**



Semantics

MOUNTPOINT

`file_path` contains the final directory name and must match the PMEM file store name. If there is no match, the statement will fail.

You must start database instance with at least `NOMOUNT` mode.

It is recommeded to use a `spfile` for the database `init.ora` file.

When you use a `spfile` , the `CREATE PMEM FILESTORE` command automatically writes the necessary `init.ora` parameters into the `spfile` to remember the configuration. If you do not use a `spfile` , you must explicitly add the required parameters to `init.ora` so that the next database instance startup will automatically mount the PMEM file store.

Example

```
CREATE PMEM FILESTORE cloud_db_1 MOUNTPOINT ‘/corp/db/cloud_db_1’
    BACKINGFILE ‘/var/pmem/foo_1.’ SIZE 2T BLOCKSIZE 8K
    AUTOEXTEND ON NEXT 10G MAXSIZE 3T
```

CREATE PROCEDURE

Purpose

Procedures are defined using PL/SQL. Therefore, this section provides some general information but refers to *Oracle Database PL/SQL Language Reference* for details of syntax and semantics.

Use the `CREATE PROCEDURE` statement to create a standalone stored procedure or a call specification.

A **procedure** is a group of PL/SQL statements that you can call by name. A **call specification** (sometimes called call spec) declares a Java method, a JavaScript method, or a third-generation language (3GL) routine so that it can be called from SQL and PL/SQL. The call spec tells Oracle Database which Java method, JavaScript function, or third-generation language (3GL) routine to invoke when a call is made. It also tells the database what type conversions to make for the arguments and return value.

Stored procedures offer advantages in the areas of development, integrity, security, performance, and memory allocation.

See Also:

- *JavaScript Developer's Guide*
- CREATE MLE MODULE
- CREATE MLE ENV
- *Oracle Database Development Guide* for more information on stored procedures, including how to call stored procedures and for information about registering external procedures.
- CREATE FUNCTION for information specific to functions, which are similar to procedures in many ways.
- CREATE PACKAGE for information on creating packages. The `CREATE PROCEDURE` statement creates a procedure as a standalone schema object. You can also create a procedure as part of a package.
- ALTER PROCEDURE and DROP PROCEDURE for information on modifying and dropping a standalone procedure.
- CREATE LIBRARY for more information about shared libraries.

Prerequisites

To create or replace a procedure in your own schema, you must have the `CREATE PROCEDURE` system privilege. To create or replace a procedure in another user's schema, you must have the `CREATE ANY PROCEDURE` system privilege.

To invoke a call spec, you may need additional privileges, for example, the `EXECUTE` object privilege on the C library for a C call spec.

To embed a `CREATE PROCEDURE` statement inside an Oracle precompiler program, you must terminate the statement with the keyword `END-EXEC` followed by the embedded SQL statement terminator for the specific language.

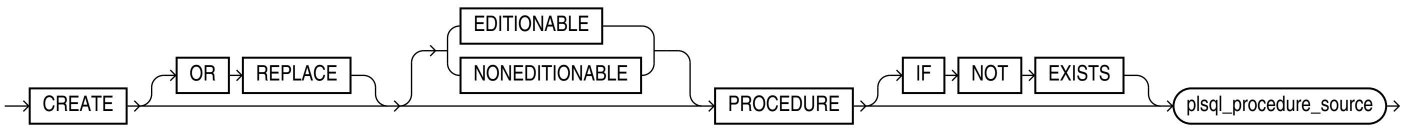
See Also:

Oracle Database PL/SQL Language Reference or *Oracle Database Java Developer's Guide* for more information

Syntax

Procedures are defined using PL/SQL. Alternatively they can refer to non-PL/SQL code such as Java, JavaScript, C, and others by means of call specifications. Therefore, the syntax diagram in this book shows only the SQL keywords. Refer to *Oracle Database PL/SQL Language Reference* for the PL/SQL syntax, semantics, and examples.

***create_procedure*::=**



(*`plsql_procedure_source`*: See *Oracle Database PL/SQL Language Reference*.)

Semantics

OR REPLACE

Specify `OR REPLACE` to re-create the procedure if it already exists. Use this clause to change the definition of an existing procedure without dropping, re-creating, and regranting object privileges previously granted on it. If you redefine a procedure, then Oracle Database recompiles it.

Users who had previously been granted privileges on a redefined procedure can still access the procedure without being regranted the privileges.

If any function-based indexes depend on the procedure, then Oracle Database marks the indexes `DISABLED`.

See Also:

ALTER PROCEDURE for information on recompiling procedures

IF NOT EXISTS

Specifying `IF NOT EXISTS` has the following effects:

- If the procedure does not exist, a new procedure is created at the end of the statement.
- If the procedure exists, this is the procedure you have at the end of the statement. A new one is not created because the older one is detected.

You can have *one* of `OR REPLACE` or `IF NOT EXISTS` in a statement at a time. Using both `OR REPLACE` with `IF NOT EXISTS` in the very same statement results in the following error: `ORA-11541: REPLACE and IF NOT EXISTS cannot coexist in the same DDL statement`.

Using `IF EXISTS` with `CREATE` results in `ORA-11543: Incorrect IF NOT EXISTS clause for CREATE statement`.

[EDITIONABLE | NONEDITIONABLE]

Use these clauses to specify whether the procedure is an editioned or noneditioned object if editioning is enabled for the schema object type `PROCEDURE` in *`schema`*. The default is `EDITIONABLE`. For information about editioned and noneditioned objects, see *Oracle Database Development Guide*.

plsql_procedure_source

See *Oracle Database PL/SQL Language Reference* for the syntax and semantics of the *`plsql_procedure_source`*.

CREATE PROFILE

Note:

Oracle recommends that you use the Database Resource Manager rather than this SQL statement to establish resource limits. The Database Resource Manager offers a more flexible means of managing and tracking resource use. For more information on the Database Resource Manager, refer to *Oracle Database Administrator's Guide.*

Purpose

Use the `CREATE PROFILE` statement to create a **profile**, which is a set of limits on database resources. If you assign the profile to a user, then that user cannot exceed these limits.

To specify resource limits for a user, you must:

- Enable resource limits dynamically with the `ALTER SYSTEM` statement or with the initialization parameter `RESOURCE_LIMIT`. This parameter does not apply to password resources. Password resources are always enabled.
- Create a profile that defines the limits using the `CREATE PROFILE` statement
- Assign the profile to the user using the `CREATE USER` or `ALTER USER` statement

In a multitenant environment, different profiles can be assigned to a common user in the root and in a PDB. When the common user logs in to the PDB, a profile whose setting applies to the session depends on whether the settings are password-related or resource-related.

- Password-related profile settings are fetched from the profile that is assigned to the common user in the root. For example, suppose you assign a common profile `c##prof` (in which `FAILED_LOGIN_ATTEMPTS` is set to 1) to common user `c##admin` in the root. In a PDB that user is assigned a local profile`local_prof` (in which `FAILED_LOGIN_ATTEMPTS` is set to 6.) Common user `c##admin` is allowed only one failed login attempt when he or she tries to log in to the PDB where `loc_prof` is assigned to him.
- Resource-related profile settings specified in the profile assigned to a user in a PDB get used without consulting resource-related settings in a profile assigned to the common user

in the root. For example, if the profile `local_prof` that is assigned to user `c##admin` in a PDB has `SESSIONS_PER_USER` set to 2, then `c##admin` is only allowed only 2 concurrent sessions when he or she logs in to the PDB `loc_prof` is assigned to him, regardless of value of this setting in a profile assigned to him in the root.

See Also:

Oracle Database Security Guide for a detailed description and explanation of how to use password management and protection

Prerequisites

To create a profile, you must have the `CREATE PROFILE` system privilege.

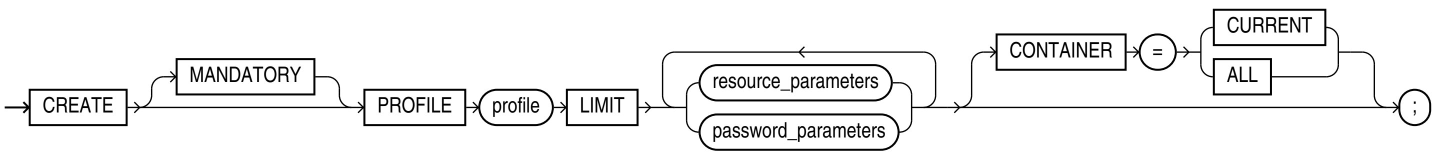
To specify the `CONTAINER` clause, you must be connected to a multitenant container database (CDB). To specify `CONTAINER = ALL`, the current container must be the root. To specify `CONTAINER = CURRENT`, the current container must be a pluggable database (PDB).

See Also:

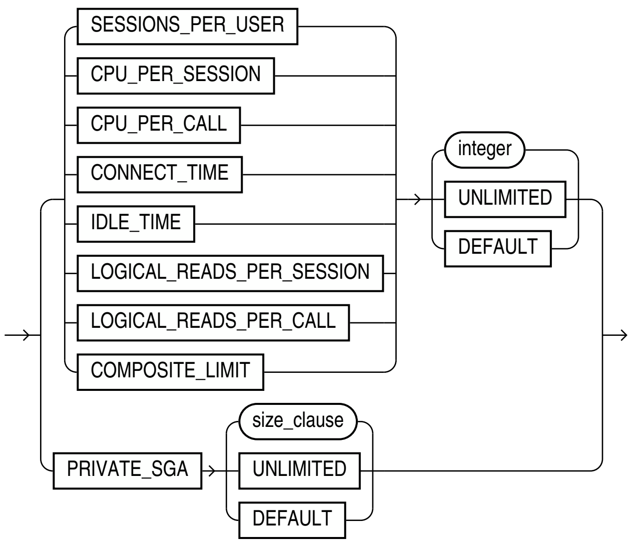
- ALTER SYSTEM for information on enabling resource limits dynamically
- *Oracle Database Reference* for information on the `RESOURCE_LIMIT` parameter
- CREATE USER and ALTER USER for information on profiles

Syntax

create_profile::=

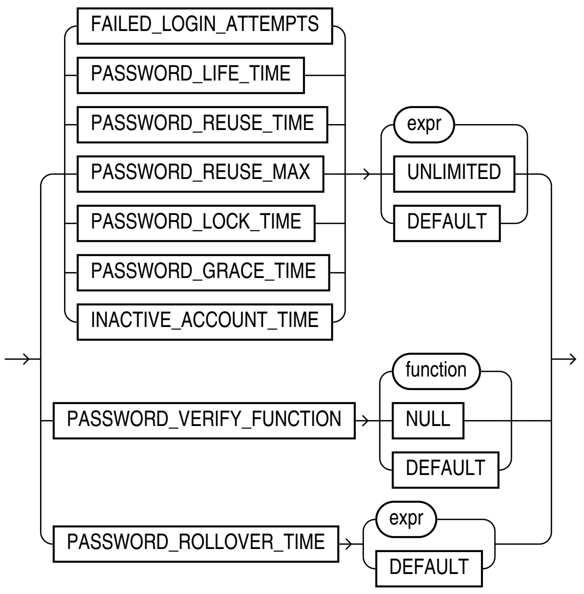


resource_parameters::=



(*size_clause*::=

***password_parameters*::=**



Semantics

profile

Specify the name of the profile to be created. The name must satisfy the requirements listed in "Database Object Naming Rules ". Use profiles to limit the database resources available to a user for a single call or a single session.

In a non-CDB, a profile name cannot begin with `C##` or `c##`.

Note:

A multitenant container database is the only supported architecture in Oracle Database 21c and later releases. While the documentation is being revised, legacy terminology may persist. In most cases, "database" and "non-CDB" refer to a CDB or PDB, depending on context. In some contexts, such as upgrades, "non-CDB" refers to a non-CDB from a previous release.

In a CDB, the requirements for a profile name are as follows:

- The name of a **common profile** must begin with characters that are a case-insensitive match to the prefix specified by the `COMMON_USER_PREFIX` initialization parameter. By default, the prefix is `C##`.
- The name of a **local profile** must not begin with characters that are a case-insensitive match to the prefix specified by the `COMMON_USER_PREFIX` initialization parameter. Regardless of the value of `COMMON_USER_PREFIX`, the name of a local profile can never begin with `C##` or `c##`.

Note:

If the value of `COMMON_USER_PREFIX` is an empty string, then there are no requirements for common or local profile names with one exception: the name of a local profile can never begin with `C##` or `c##`. Oracle recommends against using an empty string value because it might result in conflicts between the names of local and common profiles when a PDB is plugged into a different CDB, or when opening a PDB that was closed when a common user was created.

Oracle Database enforces resource limits in the following ways:

- If a user exceeds the `CONNECT_TIME` or `IDLE_TIME` session resource limit, then the database rolls back the current transaction and ends the session. When the user process next issues a call, the database returns an error.
- If a user attempts to perform an operation that exceeds the limit for other session resources, then the database aborts the operation, rolls back the current statement, and immediately returns an error. The user can then commit or roll back the current transaction, and must then end the session.
- If a user attempts to perform an operation that exceeds the limit for a single call, then the database aborts the operation, rolls back the current statement, and returns an error, leaving the current transaction intact.

MANDATORY

Specify the keyword `MANDATORY` to create a generic mandatory profile in `CDB$ROOT`. You can use the mandatory profile to enforce password complexity requirements for database user accounts across the entire CDB or individual PDBs using the profile parameter *`password_verify_function`*.

The mandatory profile adds the password complexity requirement in addition to existing profile limits for common and local users. A PDB administrator cannot remove the password complexity requirement and allow users to set insecure shorter passwords, because mandatory profiles, just like common profiles, can only be altered in `CDB$ROOT` .

You can only use *password_verify_function* and *password_grace_time* profile parameters to define the limits for the mandatory profile.

Use the profile parameter *password_grace_time* to specify a grace period for user accounts in violation of mandatory password complexity requirements and whose passwords have to be changed.

The default value for *password_verify_function* is null. The default value for *password_grace_time* is 0.

User accounts imported using Oracle Data Pump are checked for password compliance against the mandatory profile and forced to change their passwords. If the password is not changed within the grace period, further connections are rejected. On import, the password is not checked for compliance against the mandatory profile because the password is hashed and cannot be decrypted. So the password is marked to expire after a configurable period set in the parameter PASSWORD_GRACE_TIME of the mandatory profile. Once the password expires, the new password is checked for compliance against the mandatory profile. Note that, post import, the mandatory password verification check can be performed ONLY when the user logs into the database. If the user does not login, the verification does not happen. In this case there is no way for the system to know that the password complies with mandatory profile's password complexity checks and MANDATORY_PROFILE_VIOLATION will continue to show up as NO for such users.

User-Created Password Complexity Function: Example

The example creates a password complexity function *my_mandatory_function* as the argument to PASSWORD_VERIFY_FUNCTION.

```
SQL> create or replace function my_mandatory_verify_function
 ( username     varchar2,
   password     varchar2,
   old_password varchar2)
 return boolean IS
begin
   -- mandatory verify function will always be evaluated regardless of the
   -- password verify function that is associated to a particular profile/user
   -- requires the minimum password length to be 8 characters
   if not ora_complexity_check(password, chars => 8) then
      return(false);
   end if;
   return(true);
end;
/
  2    3    4    5    6    7    8    9   10   11   12   13   14   15   16   17   18
Function created.
```

Create a Mandatory Profile: Example

The example creates mandatory profile c##cdb_profile. LIMIT restricts the profile to use the only profile parameter allowed, the PASSWORD_VERIFY_FUNCTION. The PASSWORD_VERIFY_FUNCTION specifies the user-created password complexity function my_mandatory_function.

```
CREATE MANDATORY PROFILE c##cdb_profile LIMIT PASSWORD_VERIFY_FUNCTION
my_mandatory_function
     CONTAINER = ALL ;
```

If you want to apply the mandatory user profile for all PDBs in the CDB, then you must do so in the CDB root using the ALTER SYSTEM statement.

Apply the Mandatory Profile to the Entire CDB: Example

You must be in `CDB$ROOT` to execute this statement.

```
ALTER SYSTEM SET MANDATORY_USER_PROFILE=c##cdb_profile;
```

If you want to apply the mandatory user profile for individual PDBs, then you must configure the `MANDATORY_USER_PROFILE` parameter in the `init.ora` file that is associated with the PDB.

Apply the Mandatory Profile to an Individual PDB: Example

Open the `init.ora` file associated with the PDB and set the `MANDATORY_USER_PROFILE`.

```
MANDATORY_USER_PROFILE=c##cdb_profile;
```

You can use `SHOW PARAMETER` to find the current `MANDATORY_USER_PROFILE` setting.

The mandatory profile that you set in `init.ora` takes precedence over the mandatory profile that you set with the `ALTER SYSTEM` statement in the CDB root.

Restrictions

- Only common users who have been commonly granted the `ALTER PROFILE` system privilege can alter or drop the mandatory profile, and only from the CDB root.
- Only a common user who has been commonly granted the `ALTER SYSTEM` privilege or has the `SYSDBA` administrative privilege can modify the `MANDTORY_USER_PROFILE` in the `init.ora` file.

Note:

- You can use fractions of days for all parameters that limit time, with days as units. For example, 1 hour is 1/24 and 1 minute is 1/1440.
- You can specify resource limits for users regardless of whether the resource limits are enabled. However, Oracle Database does not enforce the limits until you enable them.

See Also:

- Managing Security for Database Users
- "Creating a Profile: Example"

UNLIMITED

When specified with a resource parameter, `UNLIMITED` indicates that a user assigned this profile can use an unlimited amount of this resource. When specified with a password parameter, `UNLIMITED` indicates that no limit has been set for the parameter.

DEFAULT

Specify `DEFAULT` if you want to omit a limit for this resource in this profile. A user assigned this profile is subject to the limit for this resource specified in the `DEFAULT` profile. The `DEFAULT` profile initially defines unlimited resources. You can change those limits with the `ALTER PROFILE` statement.

Any user who is not explicitly assigned a profile is subject to the limits defined in the `DEFAULT` profile. Also, if the profile that is explicitly assigned to a user omits limits for some resources or specifies `DEFAULT` for some limits, then the user is subject to the limits on those resources defined by the `DEFAULT` profile.

resource_parameters

SESSIONS_PER_USER

Specify the number of concurrent sessions to which you want to limit the user.

CPU_PER_SESSION

Specify the CPU time limit for a session, expressed in hundredth of seconds.

CPU_PER_CALL

Specify the CPU time limit for a call (a parse, execute, or fetch), expressed in hundredths of seconds.

CONNECT_TIME

Specify the total elapsed time limit for a session, expressed in minutes.

IDLE_TIME

Specify the permitted periods of continuous inactive time during a session, expressed in minutes. Long-running queries and other operations are not subject to this limit.

When you set an idle timeout of X minutes, note that the session will take X minutes, plus a couple of additional minutes to be terminated.

On the client application side, the error message shows up the next time, when the idle client attempts to issue a new command.

LOGICAL_READS_PER_SESSION

Specify the permitted number of data blocks read in a session, including blocks read from memory and disk.

LOGICAL_READS_PER_CALL

Specify the permitted number of data blocks read for a call to process a SQL statement (a parse, execute, or fetch).

PRIVATE_SGA

Specify the amount of private space a session can allocate in the shared pool of the system global area (SGA). Refer to *size_clause* for information on that clause.

> **Note:**
>
> This limit applies only if you are using shared server architecture. The private space for a session in the SGA includes private SQL and PL/SQL areas, but not shared SQL and PL/SQL areas.

COMPOSITE_LIMIT

Specify the total resource cost for a session, expressed in **service units**. Oracle Database calculates the total service units as a weighted sum of `CPU_PER_SESSION`, `CONNECT_TIME`, `LOGICAL_READS_PER_SESSION`, and `PRIVATE_SGA`.

> **See Also:**
>
> - ALTER RESOURCE COST for information on how to specify the weight for each session resource
> - "Setting Profile Resource Limits: Example"

password_parameters

Use the following clauses to set password parameters. Parameters that set lengths of time—that is, all the password parameters except `FAILED_LOGIN_ATTEMPTS` and `PASSWORD_REUSE_MAX`—are interpreted in number of days. For testing purposes you can specify minutes (*n*/1440) or even seconds (*n*/86400) for these parameters. You can also use a decimal value for this purpose (for example .0833 for approximately one hour). The minimum value is 1 second. The maximum value is 24855 days. For `FAILED_LOGIN_ATTEMPTS` and `PASSWORD_REUSE_MAX`, you must specify an integer.

FAILED_LOGIN_ATTEMPTS

Specify the number of consecutive failed attempts to log in to the user account before the account is locked. If you omit this clause, then the default is 10 times.

PASSWORD_LIFE_TIME

Specify the number of days the same password can be used for authentication. If you also set a value for `PASSWORD_GRACE_TIME`, then the password expires if it is not changed within the grace period, and further connections are rejected. If you omit this clause, then the default is 180 days.

> **See Also:**
>
> *Oracle Database Security Guide* for information on setting `PASSWORD_LIFE_TIME` to a low value

PASSWORD_REUSE_TIME and PASSWORD_REUSE_MAX

These two parameters must be set in conjunction with each other. `PASSWORD_REUSE_TIME` specifies the number of days which need to pass before a user having this profile can reuse one of their earlier passwords. `PASSWORD_REUSE_MAX` specifies the number of password changes required before the current password can be reused. For these parameters to have any effect, you must specify a value for both of them.

- If you specify a value for both of these parameters, then the user cannot reuse a password until the password has been changed the number of times specified for `PASSWORD_REUSE_MAX` during the number of days specified for `PASSWORD_REUSE_TIME`.

  For example, if you specify `PASSWORD_REUSE_TIME` to 30 and `PASSWORD_REUSE_MAX` to 10, then the user can reuse the password after 30 days if the password has already been changed 10 times.

- If you specify a value for either of these parameters and specify `UNLIMITED` for the other, then the user can never reuse a password.
- If you specify `DEFAULT` for either parameter, then Oracle Database uses the value defined in the `DEFAULT` profile. By default, the `PASSWORD_REUSE_TIME` and `PASSWORD_REUSE_MAX` parameters are set to `UNLIMITED` in the `DEFAULT` profile. If you have not changed the default setting of `UNLIMITED` in the `DEFAULT` profile, then the database treats the value for that parameter as `UNLIMITED`.
- If you set both of these parameters to `UNLIMITED`, then the database ignores both of them. This is the default if you omit both parameters.

PASSWORD_LOCK_TIME

Specify the number of days an account will be locked after the specified number of consecutive failed login attempts. If you omit this clause, then the default is 1 day.

PASSWORD_GRACE_TIME

Specify the number of days after the grace period begins during which a warning is issued and login is allowed. If you omit this clause, then the default is 7 days.

INACTIVE_ACCOUNT_TIME

Specify the permitted number of consecutive days of no logins to the user account, after which the account will be locked. The minimum value is 15 days. The maximum value is 24855. If you omit this clause, then the default is `UNLIMITED`.

PASSWORD_VERIFY_FUNCTION

You can pass a PL/SQL password complexity verification script as an argument to `CREATE PROFILE` by specifying `PASSWORD_VERIFY_FUNCTION`. Oracle Database provides a default script, but you can write your own function or use third-party software instead.

- For *`function`*, specify the name of the password complexity verification function. The function must exist in the `SYS` schema, and you must have `EXECUTE` privilege on the function.
- Specify `NULL` to indicate that no password verification is performed.

If you specify *`expr`* for any of the password parameters, then the expression can be of any form except scalar subquery expression.

Restriction on Password Parameters

When you assign a profile to an external user or a global user, the password parameters do not take effect for that user.

See Also:

"Setting Profile Password Limits: Example"

PASSWORD_ROLLOVER_TIME

You must configure a non-zero limit for the `PASSWORD_ROLLOVER_TIME` user profile parameter in order to enable the gradual database password rollover. You can configure this parameter using `CREATE PROFILE` or `ALTER PROFILE`.

Use *expr* to specify a value for PASSWORD_ROLLOVER_TIME in days. You must specify hours as a fraction of one day. For example, if you want to set the limit to four hours, *expr* would be 4/24 .

The granularity of the PASSWORD_ROLLOVER_TIME limit value is one second. For example, you can have a limit of one hour plus three minutes and five seconds by providing an *expr* like this: (1/24) + (3/1440) + (5/86400)) .

The default setting for PASSWORD_ROLLOVER_TIME is 0, which means that gradual password rollover is disabled.

Example

The example sets the gradual password rollover time period to 1 day:

```
CREATE PROFILE usr_prof LIMIT PASSWORD_ROLLOVER_TIME 1
```

Limits on PASSWORD_ROLLOVER_TIME:

- Specify a value of 0 for PASSWORD_ROLLOVER_TIME if you want to disable the password rollover period.
- Specify a positive value for PASSWORD_ROLLOVER_TIME to enable the password rollover feature for all users who are members of the profile.
- The minimum value you can specify for PASSWORD_ROLLOVER_TIME is one hour. You do this by entering 1/24. If you want to set the password rollover time to six hours, you enter 6/24 as the value for PASSWORD_ROLLOVER_TIME .
- The value for PASSWORD_ROLLOVER_TIME cannot exceed either 60 days, or the current value of the PASSWORD_GRACE_TIME limit of the profile, or the current value of the PASSWORD_LIFE_TIME limit of the profile; whichever is lowest.

To find user accounts that are currently in the password rollover period, query the ACCOUNT_STATUS column of the DBA_USERS data dictionary view. The status will be IN ROLLOVER.

The password rollover period begins the moment the user changes their password.

See Also:

Configuring Authentication

CONTAINER Clause

The CONTAINER clause applies when you are connected to a CDB. However, it is not necessary to specify the CONTAINER clause because its default values are the only allowed values.

- To create a common profile, you must be connected to the root. You can optionally specify CONTAINER = ALL, which is the default when you are connected to the root.
- To create a local profile, you must be connected to a PDB. You can optionally specify CONTAINER = CURRENT, which is the default when you are connected to a PDB.

Examples

Creating a Profile: Example

The following statement creates the profile new_profile:

```
CREATE PROFILE new_profile
  LIMIT PASSWORD_REUSE_MAX 10
        PASSWORD_REUSE_TIME 30;
```

Setting Profile Resource Limits: Example

The following statement creates the profile `app_user`:

```
CREATE PROFILE app_user LIMIT
   SESSIONS_PER_USER          UNLIMITED
   CPU_PER_SESSION            UNLIMITED
   CPU_PER_CALL               3000
   CONNECT_TIME               45
   LOGICAL_READS_PER_SESSION  DEFAULT
   LOGICAL_READS_PER_CALL     1000
   PRIVATE_SGA                15K
   COMPOSITE_LIMIT            5000000;
```

If you assign the `app_user` profile to a user, then the user is subject to the following limits in subsequent sessions:

- The user can have any number of concurrent sessions.
- In a single session, the user can consume an unlimited amount of CPU time.
- A single call made by the user cannot consume more than 30 seconds of CPU time.
- A single session cannot last for more than 45 minutes.
- In a single session, the number of data blocks read from memory and disk is subject to the limit specified in the `DEFAULT` profile.
- A single call made by the user cannot read more than 1000 data blocks from memory and disk.
- A single session cannot allocate more than 15 kilobytes of memory in the SGA.
- In a single session, the total resource cost cannot exceed 5 million service units. The formula for calculating the total resource cost is specified by the `ALTER RESOURCE COST` statement.
- Since the `app_user` profile omits a limit for `IDLE_TIME` and for password limits, the user is subject to the limits on these resources specified in the `DEFAULT` profile.

Setting Profile Password Limits: Example

The following statement creates the `app_user2` profile with password limits values set:

```
CREATE PROFILE app_user2 LIMIT
   FAILED_LOGIN_ATTEMPTS 5
   PASSWORD_LIFE_TIME 60
   PASSWORD_REUSE_TIME 60
   PASSWORD_REUSE_MAX 5
   PASSWORD_VERIFY_FUNCTION ora12c_verify_function
   PASSWORD_LOCK_TIME 1/24
   PASSWORD_GRACE_TIME 10
   INACTIVE_ACCOUNT_TIME 30;
```

This example uses the default Oracle Database password verification function, `ora12c_verify_function`. Refer to *Oracle Database Security Guide* for information on using this verification function provided or designing your own verification function.

CREATE PROPERTY GRAPH

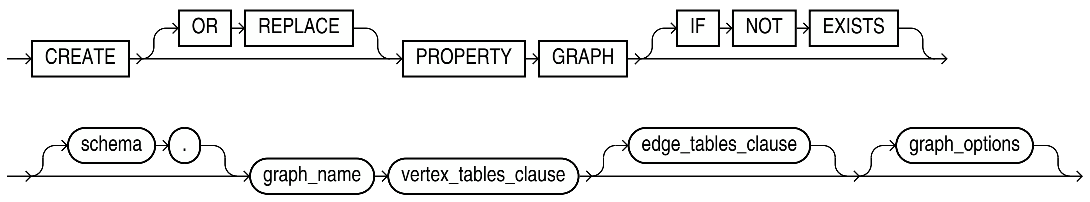
Purpose

Use `CREATE PROPERTY GRAPH` to create a property graph from existing schema objects. The schema object can be a table, an external table, a materialized view or a synonym of the table, external table, or materialized view.

Prerequistes

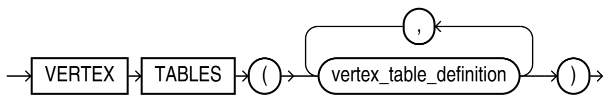
You need the `CREATE PROPERTY GRAPH` privilege to create a property graph in your own schema. To create a property graph in any schema except `SYS` and `AUDSYS`, you must have the `CREATE ANY PROPERTY GRAPH` privilege.

Syntax

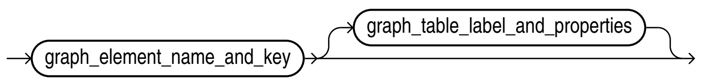


(edge_tables_clause::= , graph_options)

***vertex_tables_clause*::=**

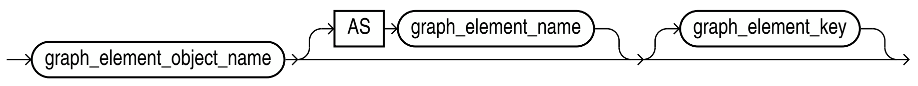


***vertex_table_definition*::=**

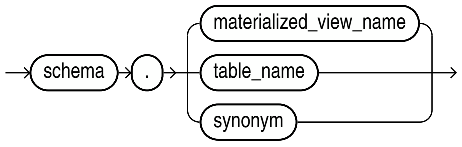


graph_table_level_and_properties

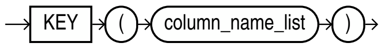
***graph_element_name_and_key*::=**



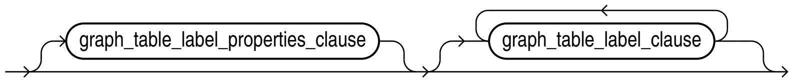
***graph_element_object_name*::=**



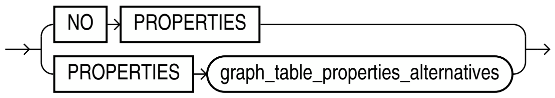
***graph_element_key*::=**



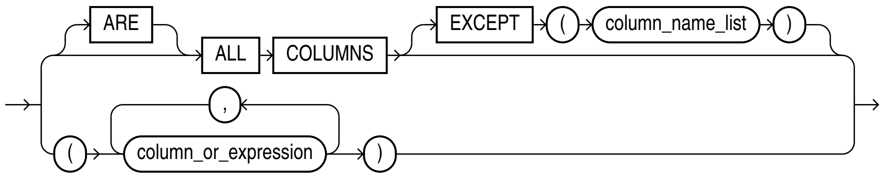
***graph_table_label_and_properties*::=**



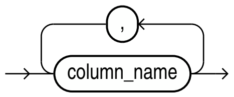
***graph_table_label_properties_clause*::=**



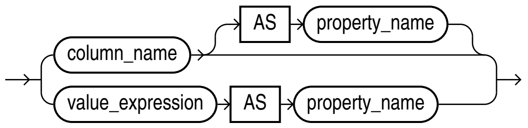
***graph_table_properties_alternatives*::=**



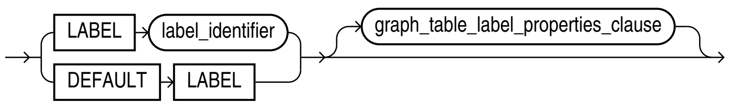
***column_name_list*::=**



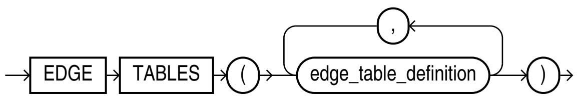
***column_or_expression*::=**



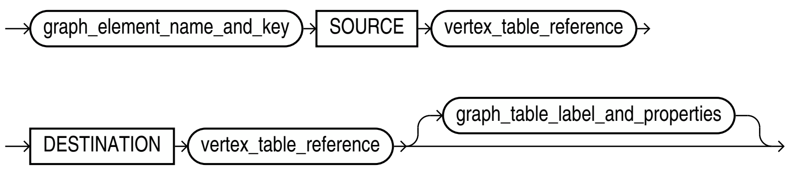
***graph_table_label_clause*::=**



***edge_tables_clause*::=**

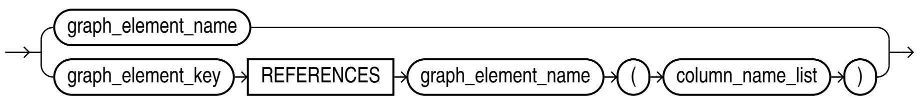


***edge_tables_definition*::=**

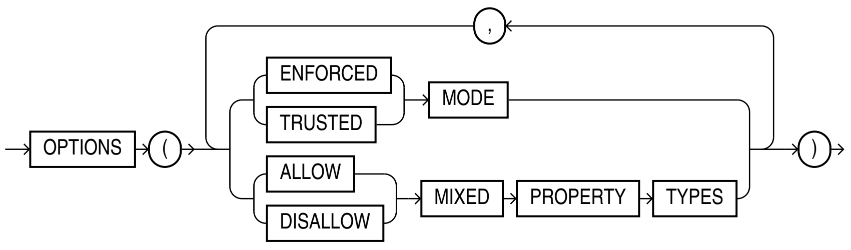


graph_table_level_and_properties

***vertex_table_reference*::=**



***graph_options*::=**



Semantics

IF NOT EXISTS

Specifying `IF NOT EXISTS` has the following effects:

- If the property graph does not exist, a new property graph is created at the end of the statement.
- If the property graph exists, the existing property graph is what you have at the end of the statement. A new one is not created because the older one is detected.

Using `IF EXISTS` with `CREATE` results in the following error: Incorrect `IF NOT EXISTS` clause for `CREATE` statement .

You must create a property graph with the *`vertex_tables_clause`* .

Specify the schema to contain the property graph. If you omit schema, then Oracle Database creates the graph in your own schema.

The name of the property graph must not be used by any other object in the same schema, because property graphs share the name space used for tables and views. `ORA-00955` is raised in the case of name conflicts.

Specify `OR REPLACE` to re-create the property graph, if it already exists. You can use this clause to change the definition of an existing property graph without dropping, re-creating, and regranting object privileges previously granted on it.

If any materialized views are dependent on the property graph, then those materialized views will be marked `UNUSABLE` and will require a full refresh to restore them to a usable state. Invalid materialized views cannot be used by query rewrite and cannot be refreshed until they are recompiled.

vertex_tables_clause

The *`vertex_tables_clause`* lets you define one or more vertex tables for the property graph. A *`vertex_table_definition`* needs to specify the underlying object name. It can optionally specify more items (like labels and properties) as explained below.

You can define a property graph by specifying just the name of the underlying object used to define the graph element table. In this case a default label with the same name as the underlying table is created and all the columns are exposed as graph properties.

The object name can be the name of a table, an external table, a materialized view, or a synonym of a table or materialized view.

The object name can be qualified by specifying the schema it resides in. This means that you can use objects from other schemas to define a graph element table. If no option is specified, the name of the specified object is used as the name of the graph element table.

Example: Create a Property Graph with Vertex Table

In the following example, the vertex table name `my_table_1` is the name of underlying object `my_table_1`.

```
CREATE PROPERTY GRAPH “myGraph” VERTEX TABLES (my_table_1);
```

Example: Create a Property Graph with a Schema-Qualified Vertex Table

In the following example, the name `my_table_1` is qualified by the schema `other_schema` and the vertex table name is the name of underlying object `my_table_1`.

```
CREATE PROPERTY GRAPH “myGraph” VERTEX TABLES (other_schema.my_table_1);
```

graph_element_name_and_key

The *`graph-element-name-and-key`* clause lets you specify:

- The name of the schema object to be used for defining a graph element table. The name of the graph element table defaults to the *graph_element_object_name* without the schema qualification, if an `AS` clause is not used to provide an alternative name.
- One or more column names used to explicitly specify what columns of the underlying object are used to identify a row in that underlying object.

Graph element table names are defined in a name space specific to the property graph: they do not conflict with the names of schema objects, nor with the names of graph element tables defined in other property graphs. This implies also that an edge table cannot use the name of a vertex table. Graph element table names follow the same rules as other identifiers: they may be quoted to indicate case-sensitivity, and are limited by default to 128 characters. Any subsequent symbolic references to a graph element table in the DDL statement must use the graph element table name, not the name of its underlying object. In particular, you must use the graph element table name when you define the source and destination vertex table of an edge table.

graph_element_object_name identifies the table or the materialized view directly or indirectly using a synonym for the table or the materialized view.

You can omit the clause *graph-element-key* in the following cases:

- The clause *graph_element_object_name* identifies a base table with a single primary key constraint. The primary key constraint takes precedence over any unique key that might also be defined.
- The clause *graph_element_object_name* identifies a base table without primary keys and a single unique key constraint where all columns are not nullable.

You must specify *graph-element-key* when *object_name* identifies a materialized view.

Example

The example shows the ways *graph-element-key* is used. In vertex table `VT3` it is used to specify a composite key made of multiple columns of the underlying table. Vertex table `ALTVT2` is defined from the same underlying object used to define vertex table `VT2`, but a different column of that object (`PK4`) is specified as identifier for its vertices. It is assumed that vertex table `VT1` has a primary key constraint or unique key with not null columns.

```
CREATE PROPERTY GRAPH "myGraph"
     VERTEX TABLES (
       VT1,
       VT2 KEY(PK2),
       VT3 KEY(PK31, PK32),
       VT2 AS ALTVT2 KEY(PK4)
     );
```

Note that a same underlying object `VT2` can be used to define another graph element table using the same primary key used to define other graph element tables from `VT2`. So in the example above, specifying `PK2` instead of `PK4` for `ALTVT2` is allowed.

When the *graph-element-key* clause is present, all the declared column names must match column names of the underlying object of the graph element table.

When the *graph-element-key* clause is omitted, the database will infer the columns from the constraints of the underlying object. If multiple primary or unique key constraints are defined for that object, inferring a key fails and an error is raised. Note that the primary constraint is only used to infer the key for the graph element table, no dependency to the constraint is created as a result of this inference. This means that the constraint may be dropped later without invalidating the graph or impact to its definition.

You can change this behavior and create a dependency to the constraint with the `ENFORCED MODE` option.

You can define multiple graph element tables from the same object. For example, a table may act as a different edge table in the same graph. You can also define a graph from tables from different schemas but with the same name. In both cases you must take care to avoid name collisions by specifying an alternative graph element table name with the optional `AS` clause.

Example: Create a Property Graph with the AS Clause

```
CREATE PROPERTY GRAPH "myGraph" VERTEX TABLES (my_table_1, other_schema.my_table_1 AS
my_table2);
```

Restrictions

Restrictions that apply on primary key constraints also apply on vertex and edge table keys:

- Columns of the following built-in data types can be used to define keys of vertex or edge tables: `VARCHAR2`, `NVARCHAR2`, `NUMBER`, `BINARY_FLOAT`, `BINARY_DOUBLE`, `CHAR`, `NCHAR`, `DATE`, `INTERVAL` (both `YEAR TO MONTH` and `DAY TO SECOND`), and `TIMESTAMP` (but not `TIMESTAMP WITH TIME ZONE`).
- A composite key cannot exceed 32 columns.

edge_tables_clause

Use *`edge_tables_clause`* to specify one or more *`edge_table_definition`* clauses. Each *`edge_table_definition`* clause specifies the underlying object used to define the edge table of the graph.

edge_table_definition

Use *`edge_table_definition`* to explicitly define the vertex table that acts as the source of the edge, and the vertex table that acts as the destination of the edge using the keywords `SOURCE` and `DESTINATION`.

vertex_table_reference

The source and destination of the edge specify a *`vertex_table_reference`*. The *`vertex_table_reference`* specifies three components:

- The graph element name of a vertex table, *`graph_element_name`*
- A list of columns of the edge table to be treated as foreign key *`graph_element_key`*.
- A list of columns of the referenced vertex table to be treated as referenced keys *`column_name_list`*

The *`graph_element_name`* of a *`vertex_table_reference`* clause must be defined by a preceding *`vertex_tables_definition`* clause. A vertex table name defined in the *`vertex_tables_definition`* may be used to define multiple edge table definitions, either as a source for the edge, a destination, or both.

Example

Given the following vertex tables defined as follows:

```
CREATE PROPERTY GRAPH "myGraph"
     VERTEX TABLES (
       VT1,
       VT2 KEY(PK2),
       VT3 KEY(PK31, PK32),
       VT2 AS ALTVT2 KEY(PK4)
     )
     EDGE TABLES (
       E1 SOURCE VT1
          DESTINATION VT2,
       E2 SOURCE      KEY(FK1) REFERENCES VT1 (PK1)
          DESTINATION KEY(FK2) REFERENCES VT2 (PK2),
       E3 SOURCE      KEY(FK1) REFERENCES VT1 (PK1)
          DESTINATION VT2,
       E4 SOURCE VT1
          DESTINATION KEY(FK5) REFERENCES VT2(RK5))
;
```

Both vertex-table-reference from edge table `E1` to, respectively, source table `VT1` and destination table `VT2` are declared implicitly. When using this syntax, the user relies on the database to infer source and destination keys from existing foreign-key constraints between `E1` and, respectively, `VT1` and `VT2`. In this case, there must be exactly only foreign-key constraints between `E1` and `VT1` (respectively, `VT2`).

If that is not the case an error is raised. Note that the foreign key constraint is only used to infer the foreign key relationships between an edge table and its source and destination vertex tables. No dependency to the foreign key constraint is created as a result of this inference. This means that the constraint may be dropped later without invalidating the graph or impact to its definition.

In contrast, both vertex-table-references from edge table `E2`, respectively, source vertex table `VT1` and destination vertex table `VT2` are declared explicitly. This syntax is mandatory when there are no or multiple foreign constraints defined between `E2` and its referenced vertex tables.

Implicit and explicit syntax can be mixed, as shown for edge table `E3` and `E4`, wherein the former uses an explicit syntax only of the source table, while the latter uses it only for the destination table.

Note that the *`column_name_list`* clause that specifies the columns of the underlyng object for the referenced vertex table to be treated as referenced key don't necessarily match the columns specified in the graph-element-key sub-clause of the vertex-definition clause that defined the referenced vertex table. This is illustrated with edge table E4 from the example above: the referenced key specified for `VT2` is `RK5`, whereas the key that was specified in the vertex table definition clause for `VT2` was `PK2`.

graph_table_label_and_properties

Use *`graph_table_label_and_properties`* to specify the labels and properties of a graph. Then you can formulate graph queries using the labels of a graph and the properties defined by these labels.

Graph Labels and Properties

You can associate graph element tables with labels that expose the columns of the underlying object as properties. A label has a name and declares a mapping of property names to columns of the underlying object for a given graph element table. Labels give you a way to refer to one or more graph element tables in a graph query using a same label name. Properties give you a way to refer to columns of one or more graph element tables using a same, possibly label qualified, property name.

You can associate one label to multiple graph element tables, provided that all the graph element tables that share this label declare the same property name. The columns or value expressions exposed by the same property name must have union compatible types.

A graph element table may be associated with multiple labels.

Graph element tables are always associated with at least one label. If none is defined explicitly, a label is assigned automatically with the same name as the graph element table.

Declaring labels and properties is optional. All the following ways to explicitly declare properties and labels are valid:

- Only properties using *`graph_table_label_properties_clause`*
- Only labels using *`graph_table_label_clause`*
- Both properties and labels using *`graph_table_label_properties_clause`*
- No properties or labels

graph_table_label_properties_clause

The properties are derived from columns or SQL value expressions of columns of the underlying object used to define the graph element table. By default, all visible columns are mapped to properties and the names of the properties default to the names of these columns. Pseudo-columns cannot be exposed as a property.

The *`graph_table_label_properties_clause`* provides the following options:

- `PROPERTIES [ ARE ] ALL COLUMNS`

  All visible columns of the graph element table are exposed as properties of the label with the same names as the column names. (This is the default when no properties are specified.) Note that all visible columns that are used as keys will also be exposed as properties.

- `PROPERTIES [ ARE ] ALL COLUMNS EXCEPT(`*`column_name_list`*`)`

  All visible columns are exposed as properties of the label except for the ones explicitly listed. This option is useful, if the number of columns not supposed to be exposed as properties is small compared to the number of columns exposed as properties.

- `PROPERTIES` (*`column_name_list`*)

  Only the columns explicitly listed become properties of the label with the same names as the column names. This option is useful, if the number of columns exposed as properties is small compared to the number of columns not supposed to be exposed as properties, or if the user wants to expose invisible columns. It is also useful when renaming some or all of the properties is necessary, as shown in the following:

- `PROPERTIES`(*`column_name_list AS property_name`*)

  Only the columns explicitly listed become properties of the label. If `AS` *`property_name`* is appended to the *`column_name`*, then the *`property_name`* is used as the property name, otherwise the property name defaults to the column name. A property name can only be defined once per label. The `AS` clause is useful to enable association of one label to multiple graph element tables.

- `PROPERTIES` (*`value_expression AS property_name, ...`*)

  It is possible to define a property as an expression over columns of the underlying object used to define a graph element table. The `AS` clause is mandatory in this case. A value expression can be a scalar expression, or a function expression, or expression list. It can contain only the following forms of expression:

  – Columns of the underlying object
  – Constants: strings or numbers

- – Deterministic functions — either SQL built-in functions or PL/SQL functions

  No other expression forms are valid (in particular, sub-query expressions and aggregate functions are invalid). The expression can only return a scalar data type. SQL operator used in the expression must be deterministic

- `NO PROPERTIES`

  The label does not expose any column of the underlying object associated with the graph element table.

Note that that for a given vertex or edge table, the properties exposed in the various labels applied to this vertex or edge table must have the same definition.

graph_table_properties_alternatives

You can control explicitly what columns are exposed as properties using the options of *`graph_table_properties_alternatives`* clause.

Note that for implicit clauses, for example `ALL COLUMNS`, the list of exposed columns is determined when the graph is created. If you add additional columns to a table after you create the graph, for example you add a virtual column, the graph will not reflect the virtual column.

Examples

The following example illustrates various uses of *`graph_table_label_and_properties`* for declaring labels associated to graph element tables (here only vertex tables) and their properties:

```
CREATE PROPERTY GRAPH “myGraph”
  VERTEX TABLES (
    HR.VT1,
    VT1  AS ALTVT1,
    VT2  LABEL “foo” ,
    VT3  NO PROPERTIES,
    VT4  PROPERTIES(C1),
    VT5  PROPERTIES(C1, C2 as P2),
    VT6  LABEL “bar” LABEL “weighted” NO PROPERTIES,
    VT7  LABEL “bar2” PROPERTIES ARE ALL COLUMNS EXCEPT (C3),
    VT8  LABEL “weighted” NO PROPERTIES DEFAULT LABEL,
    VT9  PROPERTIES(Cx + Cy * 0.15 AS PX, Cz AS PZ),
    VT10 PROPERTIES(JSON_VALUE(JCOL,
      ‘$.person.creditScore[0]’ returning number) AS CREDITSCORE,
    VT11 PROPERTIES(XMLCAST(XMLQUERY(‘/purchaseOrder/poDate’
       PASSING XCOL RETURNING CONTENT) AS DATE) AS PURCHASEDATE
 );
```

The meaning of each vertex table definition of the example:

- Vertex table `VT1` defines (implicitly) a single label `VT1` that exposes all the visible columns of the underlying object `HR.VT1`. This is the default when no options to specify label or property are used.
- Vertex table `ALTVT1` defines (implicitly) a single label `ALTVT1` that exposes all the visible columns of the underlying object `VT1`. If object name `VT1` resolves to `HR.VT1`, both vertex tables `ALTVT1` and `VT1` exposes the same columns from the same underlying object `HR.VT1`
- Vertex table `VT2` defines a single label `foo` that exposes all the visible columns of the underlying object `VT2`.
- Vertex table `VT3` defines (implicitly) a single label `VT3` without any properties. No columns from the underlying object `VT3` are exposed.

- Vertex table VT4 defines (implicitly) a single label VT4 with a single property C1 that exposes the column C1 of the underlying object VT4.
- Vertex table VT5 defines (implicitly) a single label VT5 with a two properties C1 and P2, that exposes, respectively, column C1 and C2 of the underlying object VT5.
- Vertex table VT6 defines two labels, bar and weighted, such that label bar exposes all visible columns of underlying object VT6 as properties, while label "weighted" has no properties.
- Vertex table VT7 defines a single label bar that exposes all columns of the underlying object VT3 but its column C3.
- Vertex table VT8 defines two labels, bar2 and VT8 (via the DEFAULT LABEL). The former has no properties while the later exposes all columns as properties.
- Vertex table VT9 defines (implicitly) a single label VT9 with two properties PX and PZ, with PX exposing an expression over columns Cx and Cy of the underlying object VT9, while PZ exposes its column Cz.
- Vertex table VT10 defines one property CREDITSCORE that extracts creditScore value as number data type from the JSON type column JCOL.
- Vertex table VT11 defines one property PURCHASEDATE that extracts purchase order date value as date data type from the XMLtype column XCOL.

graph_options

Use the OPTIONS clause to specify a comma separated list of options. Each option can appear only once. You can specify the mode of the graph, one of ENFORCED or TRUSTED. You can either allow or disallow mixed types in properties with the same name.

ENFORCED or TRUSTED Mode

Option ENFORCED on the property graph means that guarantees are enforced over the entire graph via constraints in the ENABLE VALIDATE state.

If you do not specify ENFORCED , the mode is TRUSTED. This is the default mode.

A property graph is in ENFORCED mode if :

- All of its graph element tables are defined with a primary key that matches an existing ENABLE VALIDATE primary key constraint, or a unique key constraint in the ENABLE VALIDATE state where all columns are not nullable.
- All vertex table references from edge tables are defined with a foreign key that matches an existing ENABLE VALIDATE foreign key constraint between the underlying objects for the edge and the vertex table respectively, that (foreign key constraint) defines the source or destination vertex table reference. Further, the foreign key columns must have a NOT NULL constraint, and a ENABLE VALIDATE primary key constraint, or both a unique and not null constraints, must be defined on the referenced keys for each of the source and destination table.

If neither one of these conditions is true, then the property graph is in TRUSTED mode. This is the default mode.

Example: Creation of a Property Graph in Enforced Mode

```
CREATE PROPERTY GRAPH "mygraph"
  VERTEX TABLES (VT1, VT2 KEY(PK2)),
  EDGE TABLES (
    ET1 SOURCE VT1 DESTINATION VT2,
    ET2 SOURCE KEY(FK2) REFERENCES VT2 (PK2) DESTINATION VT1)
 OPTIONS(ENFORCED MODE);
```

The DDL in the example fails if any of the following is true:

- If neither a primary key constraint, or exacly one unique key constraints on non nullable columns can be found for vertex table `VT1`, edge table `ET1` or edge table `ET2`, regardless of the `ENFORCED MODE` option.
- If there is not exactly one foreign key between `ET1` and its referenced tables `VT1` and `VT2`, or between `ET2` and its referenced table `VT1`, regardless of the `ENFORCED MODE` option.
- If neither a single primary key constraint on `VT2.PK2`, or a unique key constraint and `NOT NULL` constraint on `VT2.PK2` can be found, as a result of the `ENFORCED MODE` option.
- If no foreign key constraint can be found between `ET2.FK2` and its referenced table `VT2.PK2`, and there is neither a primary key constraint, or both a unique key and a `NOT NULL` constraint on `VT2.PK2`, as a result of the `ENFORCED MODE` option.

DDL operations on constraints on tables that form the underlying objects of a property graph can invalidate the graph if this one was successfully created with the `ENFORCED MODE` option and have no effect on the graph if this one was successfully created with the `TRUSTED MODE` option.

Table 14-1 DDL Operations on Constraints that Causes Graph Created with the ENFORCED MODE Option to become Invalid

Operations	Description

`pkc` is a `PRIMARY KEY` constraint in the statements below.
 	
`ALTER TABLE t DROP CONSTRAINT pkc;`
 	
`ALTER TABLE t DISABLE CONSTRAINT pkc;`
 	
`ALTER TABLE t ENABLE NOVALIDATE CONSTRAINT pkc;`
	If `t` is a graph element table `e` of `G`, and `pkc` is a primary or unique key constraint on columns used as keys for `e`, and `G` was in `ENFORCED` mode, `G` is changed to be in `TRUSTED` mode.
	
	If `t` is a vertex table `e` of `G` and `pkc` is a primary or unique key constraint on columns used to define a referenced key of a foreign key constraint with one or more edge tables, and `G` was in `ENFORCED` mode, `G` is changed to be in `TRUSTED` mode.

`fkc` is a `FOREIGN KEY` constraint in the statements below.
 	
`ALTER TABLE t DROP CONSTRAINT fkc;`
 	
`ALTER TABLE t DISABLE CONSTRAINT fkc;`
 	
`ALTER TABLE t ENABLE NOVALIDATE CONSTRAINT fkc;`
	If `t` is an edge table `e` of `G`, and `fkc` is a foreign key constraint on columns used as source or destination keys for `e` referencing columns of vertex table `v`, and `G` was in `ENFORCED` mode, `G` is changed to be in `TRUSTED` mode

ALLOW or DISALLOW MIXED PROPERTY TYPES

`DISALLOW` means that the types of properties with same name should be exactly the same, regardless of the labels where they come. Use `DISALLOW` when you want to ensure that a given property has the same type across all labels.

`ALLOW` means that the types of properties with same name exposed in different labels can be distinct and of properties with same name coming from same label should be `UNION` compatible.

If you specify `DISALLOW MIXED PROPERTY TYPES`, the properties of a given name must have exactly the same type in every label. Note that this option also requires that you define a label associated with multiple graph element tables with the same data type.

The table summarizes the compatibility rules.

Table 14-2 Compatibility Rules for Mixed Property Types

Options	ALLOW	DISALLOW
Properties with same name in different definition of the same labels	Types must be `UNION` Compatible	Types must match
Properties with same name from different labels	Any	Types must match

`DISALLOW MIXED PROPERTY TYPES` is the default.

Dependencies Between Property Graph and its Underlying Objects

A property graph depends on the underlying objects it is based upon, tables, materialized views, or synonyms of tables or materialized views. Changes in these underlying objects can render the property graph invalid. Cursors that depend on the underlying objects are also invalidated. Queries against an invalid property graph in invalid state will error.

The following summarizes operations on dependent objects that cause a property graph to become invalid:

Table 14-3 Operations on Dependent Objects that Invalidate a Property Graph

Operations	Result

`DROP TABLE t ;`
`DROP [PUBLIC] SYNONYM t;`
`DROP MATERIALIZED VIEW t;`
`CREATE OR REPLACE [PUBLIC] SYNONYM t;`
	If `t` is used to define a graph element table `e` of graph `G`, then `G` becomes invalid.

`RENAME t TO t2;`	If `t` is used to defined a graph element table `e` of graph `G`, then `G` becomes invalid

`ALTER TABLE` for dropping an unused column `C` of table `t`
 	
`ALTER TABLE`
	If `t` is used to defined a graph element table `e` of graph `G`, then `G` becomes invalid

`ALTER TABLE t RENAME C TO C2;`	If `t` is used to defined a graph element table `e` of graph `G` and at least one label applied to *e* define a property as an SQL operator expression, then `G` becomes invalid
`ALTER TABLE` for modifying the type of a column `C` of table `t`	If `t` is used to defined a graph element table `e` of graph `G`, then `G` becomes invalid

Other DDL operations to alter dependent tables, views, or synonyms do not invaliate the property graph.

Note also that using a materialized view to define vertex or edge tables in a property graph creates a dependency to the container table for the view, not directly to the materialized view schema object. This has the following implications:

- When dropping a materialized view but preserving its table (i.e., using `PRESERVE TABLE`), the property graph remains valid.
- When dropping one of the tables, views, or synonyms used in the definition of the materialized view, the materialized view becomes invalid, but the property graph remains valid as it only depends on the container table.

This behavior is similar to the behavior of views defined over a materialized view.

Revalidating a Property Graph

Changes to the underlying objects of a property graph may invalidate the graph. An invalid state indicates that the metadata of the property graph describes an incorrect definition with respect to the property graph data model.

You can revalidate the property graph by redefining it with `CREATE OR REPLACE PROPERTY GRAPH` .

Sometimes however a graph may report an invalid state when it is actually valid. This can happen when the dependencies of the graph to its underlying objects are too coarse. When this happens you can revalidate the graph using `ALTER PROPERTY GRAPH COMPILE` instead of redefining it.

See Also:

ALTER PROPERTY GRAPH

Examples

Example: Property Graph Without Explicit Labels or Properties

In the example a property graph `myGraph` is created without labels or properties. A label with name `mytable` is automatically associated with the vertex table `mytable` defined over the object `myschema`. A label with name `T2` is automatically associated with the vertex table `T2` defined over the object `mytable2`.

All the columns of the underlying tables `mytable` and `T2` are exposed as properties. The property name is the column name.

```
CREATE PROPERTY GRAPH "myGraph"
      VERTEX TABLES ("myschema". "mytable", "mytable2" AS T2);
```

Example: Property Graph With An Explicit Label

In the example the vertex table `mytable` is associated with the label `person`.

All the columns of `mytable` are exposed as properties.

```
CREATE PROPERTY GRAPH "myGraph"
     VERTEX TABLES ("myschema". "mytable" LABEL "person");
```

If a label with the name of the graph element table is also needed, you have to explicitly declare it in addition to the `person` label. You can do this by declaring another explicit label that has the name of the graph element table, or by adding a `DEFAULT LABEL`.

The following vertex table declarations are semantically equivalent and all associate the graph element table `mytable` with the label `mytable`:

```
CREATE PROPERTY GRAPH “myGraph”
  VERTEX TABLES (“myschema”. “mytable” LABEL “mytable”);

CREATE PROPERTY GRAPH
  VERTEX TABLES (“myschema”. “mytable” DEFAULT LABEL);

CREATE PROPERTY GRAPH
  VERTEX TABLES (“myschema”. “mytable” AS “mytable”);

CREATE PROPERTY GRAPH
  VERTEX TABLES (“myschema”. “mytable”);
```

Example: Property Graph With Multipe Labels

You can associate multiple labels to the same graph element. The example vertex table `mytable` is associated with two labels `foo` and `bar`.

All the columns of `mytable` are exposed as properties.

```
CREATE PROPERTY GRAPH “myGraph”
  VERTEX TABLES (“myschema”. “mytable” LABEL “foo” LABEL “bar”);
```

Example: Property Graph With One Label Associating Multiple Graph Elements

The example shows a property graph `mygraph` where the shared `weighted` label associates two vertex tables `mytable1` and `mytable2` and two edge tables `E1` and `E2` .

All the columns of `mytable1` and `mytable2`are exposed as properties.

```
CREATE PROPERTY GRAPH “myGraph”
    VERTEX TABLES (
      “mytable1” LABEL “foo” LABEL “weighted”,
      “mytable2” LABEL “weighted”),
    EDGE TABLES (
      "E1" SOURCE  “mytable1” DESTINATION “mytable2” LABEL “weighted”
      "E2" SOURCE  “mytable2” DESTINATION “mytable1” LABEL “weighted”
  );
```

CREATE RESTORE POINT

Purpose

Use the `CREATE RESTORE POINT` statement to create a **restore point**, which is a name associated with a timestamp or an SCN of the database. A restore point can be used to flash back a table or the database to the time specified by the restore point without the need to determine the SCN or timestamp. Restore points are also useful in various RMAN operations, including backups and database duplication. You can use RMAN to create restore points in the process of implementing an archival backup.

See Also:

- *Oracle Database Backup and Recovery User's Guide* for more information on creating and using restore points and guaranteed restore points, for information on database duplication, and for information on archival backups
- FLASHBACK DATABASE, FLASHBACK TABLE , and DROP RESTORE POINT for information on using and dropping restore points

Prerequisites

To create a normal restore point, you must have the SELECT ANY DICTIONARY, FLASHBACK ANY TABLE, SYSDBA, SYSBACKUP, or SYSDG system privilege.

To create a guaranteed restore point, you must fulfill *one* of the following conditions:

- You must connect AS SYSDBA, or AS SYSBACKUP, or AS SYSDG.
- You must have been granted the SYSDBA privilege and be using a multitenant database.
- You must be running as user SYS, and be using a a multitenant database.

To view or use a restore point, you must have the SELECT ANY DICTIONARY, FLASHBACK ANY TABLE, SYSDBA, SYSBACKUP, or SYSDG system privilege or the SELECT_CATALOG_ROLE role.

You can create a restore point on a primary or standby database. The database can be open, or mounted but not open. If the database is mounted, then it must have been shut down consistently before being mounted unless it is a physical standby database.

You must have created a fast recovery area before creating a guaranteed restore point. You need not enable flashback database before you create the guaranteed restore point. The database must be in ARCHIVELOG mode if you are creating a guaranteed restore point.

You need not enable flashback database before you create a normal restore point, because normal restore points have other applications besides FLASHBACK DATABASE. However, you would need to have enabled flashback database before you create a normal restore point, if you intend to perform a FLASHBACK DATABASE to that normal restore point.

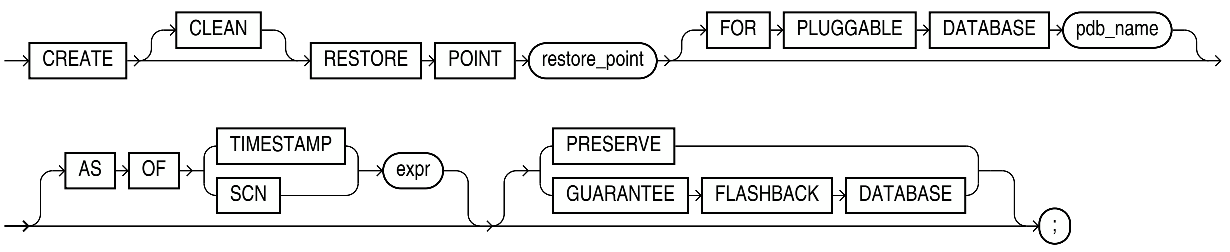
You can create, use, or view a restore point when connected to a multitenant container database (CDB) as follows:

- To create a normal CDB restore point, the current container must be the root and you must have the SELECT ANY DICTIONARY or FLASHBACK ANY TABLE system privilege, either granted commonly or granted locally in the root, or the SYSDBA, SYSBACKUP, or SYSDG system privilege granted commonly.
- To create a guaranteed CDB restore point, the current container must be the root and you must have the SYSDBA, SYSBACKUP, or SYSDG system privilege granted commonly.
- To view a CDB restore point, the current container must be the root and you must have the SELECT ANY DICTIONARY or FLASHBACK ANY TABLE system privilege or the SELECT_CATALOG_ROLE role, either granted commonly or granted locally in the root, or the SYSDBA, SYSBACKUP, or SYSDG system privilege granted commonly, or the current container must be a PDB and you must have the SELECT ANY DICTIONARY, FLASHBACK ANY TABLE, SYSDBA, SYSBACKUP, or SYSDG system privilege, granted commonly or granted locally in that PDB.
- To use a CDB restore point, you must have the SELECT ANY DICTIONARY or FLASHBACK ANY TABLE system privilege or the SELECT_CATALOG_ROLE role, either granted commonly or granted locally in the root, or the SYSDBA, SYSBACKUP, or SYSDG system privilege granted commonly.
- To create a normal PDB restore point, the current container must be the root and you must have the SELECT ANY DICTIONARY or FLASHBACK ANY TABLE system privilege, either granted commonly or granted locally in the root, or the SYSDBA, SYSBACKUP, or SYSDG system privilege granted commonly, or the current container must be the PDB for which you want to create the restore point and you must have the SELECT ANY DICTIONARY, FLASHBACK ANY TABLE, SYSDBA, SYSBACKUP, or SYSDG system privilege, granted commonly or granted locally in that PDB.

- To create a guaranteed PDB restore point, the current container must be the root and you must have the `SYSDBA`, `SYSBACKUP`, or `SYSDG` system privilege, granted commonly, or the current container must be the PDB for which you want to create the restore point and you must have the `SYSDBA`, `SYSBACKUP`, or `SYSDG` system privilege, granted commonly or granted locally in that PDB.
- To view a PDB restore point, the current container must be the root and you must have the `SELECT ANY DICTIONARY` or `FLASHBACK ANY TABLE` system privilege or the `SELECT_CATALOG_ROLE` role, either granted commonly or granted locally in the root, or the `SYSDBA`, `SYSBACKUP`, or `SYSDG` system privilege granted commonly, or the current container must be the PDB for the restore point and you must have the `SELECT ANY DICTIONARY`, `FLASHBACK ANY TABLE`, `SYSDBA`, `SYSBACKUP`, or `SYSDG` system privilege, granted commonly or granted locally in that PDB.
- To use a PDB restore point, the current container must be the PDB for the restore point and you must have the `SELECT ANY DICTIONARY`, `FLASHBACK ANY TABLE`, `SYSDBA`, `SYSBACKUP`, or `SYSDG` system privilege, granted commonly or granted locally in that PDB.

Syntax

***create_restore_point*::=**



Semantics

CLEAN

You can specify `CLEAN` only when creating a PDB restore point. The PDB must use shared undo and must be closed with no outstanding transactions. Flashing back a PDB using shared undo to a clean PDB restore point does not require restoring backups or creating a clone instance. Therefore, it is faster than flashing back a PDB using shared undo to an SCN or other type of restore point.

restore_point

Specify the name of the restore point. The name must satisfy the requirements listed in "Database Object Naming Rules ".

In a multitenant environment, the CDB and PDBs have their own namespaces for restore points. Therefore, the CDB and each PDB can have a restore point with the same name. When you specify a restore point name in a PDB or for a PDB operation, the name is first interpreted as a PDB restore point for the concerned PDB. If a PDB restore point with the specified name is not found, then it is interpreted as a CDB restore point.

The database can retain at least 2048 normal restore points. In a Multitenant environment, a CDB can retain at least 2048 normal restore points across the entire CDB, including PDB restore points. Normal restore points are retained in the database for at least the number of days specified for the `CONTROL_FILE_RECORD_KEEP_TIME` initialization parameter. The default value of that parameter is 7 days. Guaranteed restore points are retained in the database until explicitly dropped by the user.

If you specify neither `PRESERVE` nor `GUARANTEE FLASHBACK DATABASE`, then the resulting restore point enables you to flash the database back to a restore point within the time period determined by the `DB_FLASHBACK_RETENTION_TARGET` initialization parameter. The database automatically manages such restore points. When the maximum number of restore points is reached, according to the rules described in *`restore_point`* above, the database automatically drops the oldest restore point. Under some circumstances the restore points will be retained in the RMAN recovery catalog for use in restoring long-term backups. You can explicitly drop a restore point using the `DROP RESTORE POINT` statement.

FOR PLUGGABLE DATABASE

This clause enables you to create a PDB restore point when you are connected to the root. For *`pdb_name`*, specify the name of the PDB.

If you are connected to the PDB for which you want to create the restore point, then it is not necessary to specify this clause. However, if you specify this clause, then you must specify the name of the PDB to which you are connected.

AS OF Clause

Use this clause to create a restore point at a specified datetime or SCN in the past. If you specify `TIMESTAMP`, then *`expr`* must be a valid datetime expression resolving to a time in the past. If you specify SCN, then *`expr`* must be a valid SCN in the database in the past. In either case, *`expr`* must refer to a datetime or SCN in the current incarnation of the database.

PRESERVE

Specify `PRESERVE` to indicate that the restore point must be explicitly deleted. Such restore points are useful for preserving a flashback database.

GUARANTEE FLASHBACK DATABASE

A guaranteed restore point enables you to flash the database back deterministically to the restore point regardless of the `DB_FLASHBACK_RETENTION_TARGET` initialization parameter setting. The guaranteed ability to flash back depends on sufficient space being available in the fast recovery area.

Guaranteed restore points guarantee only that the database will maintain enough flashback logs to flashback the database to the guaranteed restore point. It does not guarantee that the database will have enough undo to flashback any table to the same restore point.

Guaranteed restore points are always preserved. They must be dropped explicitly by the user using the `DROP RESTORE POINT` statement. They do not age out. Guaranteed restore points can use considerable space in the fast recovery area. Therefore, Oracle recommends that you create guaranteed restore points only after careful consideration.

Examples

Creating and Using a Restore Point: Example

The following example creates a normal restore point, updates a table, and then flashes back the altered table to the restore point. The example assumes the user `hr` has the appropriate system privileges to use each of the statements.

```
CREATE RESTORE POINT good_data;
```

```
SELECT salary FROM employees WHERE employee_id = 108;

    SALARY
----------
     12000

UPDATE employees SET salary = salary*10
   WHERE employee_id = 108;

SELECT salary FROM employees
   WHERE employee_id = 108;

    SALARY
----------
    120000

COMMIT;

FLASHBACK TABLE employees TO RESTORE POINT good_data;

SELECT salary FROM employees
   WHERE employee_id = 108;

    SALARY
----------
     12000
```

CREATE ROLE

Purpose

Use the `CREATE ROLE` statement to create a **role**, which is a set of privileges that can be granted to users or to other roles. You can use roles to administer database privileges. You can add privileges to a role and then grant the role to a user. The user can then enable the role and exercise the privileges granted by the role.

A role contains all privileges granted to the role and all privileges of other roles granted to it. A new role is initially empty. You add privileges to a role with the `GRANT` statement.

If you create a role that is `NOT IDENTIFIED` or is `IDENTIFIED EXTERNALLY` or `BY` *`password`*, then Oracle Database grants you the role with `ADMIN OPTION`. However, if you create a role `IDENTIFIED GLOBALLY`, then the database does not grant you the role. A global role cannot be granted to a user or role directly. Global roles can be granted through EUS enterprise roles, mapped group memberships, and mapped app roles.

See Also:

- GRANT for information on granting roles
- ALTER USER for information on enabling roles
- ALTER ROLE and DROP ROLE for information on modifying or removing a role from the database
- SET ROLE for information on enabling and disabling roles for the current session
- *Oracle Database Security Guide* for general information about roles
- *Oracle Database Enterprise User Security Administrator's Guide* for details on enterprise roles

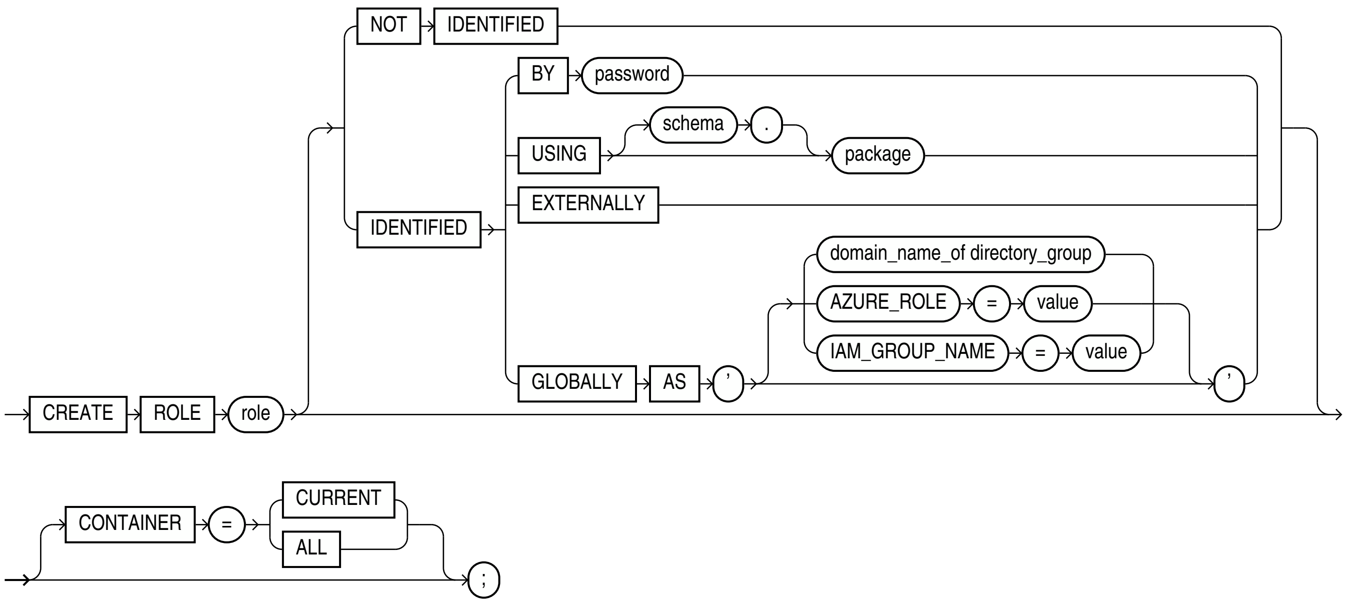
Prerequisites

You must have the `CREATE ROLE` system privilege.

To specify the `CONTAINER` clause, you must be connected to a multitenant container database (CDB). To specify `CONTAINER = ALL`, the current container must be the root. To specify `CONTAINER = CURRENT`, the current container must be a pluggable database (PDB).

Syntax

create_role::=



Semantics

role

Specify the name of the role to be created. The name must satisfy the requirements listed in "Database Object Naming Rules ". Oracle recommends that the role contain at least one single-byte character regardless of whether the database character set also contains multibyte characters. The maximum length of the role name is 128 bytes. The maximum number of user-defined roles that can be enabled for a single user at one time is 148.

In a non-CDB, a role name cannot begin with `C##` or `c##`.

> **Note:**
>
> A multitenant container database is the only supported architecture in Oracle Database 21c and later releases. While the documentation is being revised, legacy terminology may persist. In most cases, "database" and "non-CDB" refer to a CDB or PDB, depending on context. In some contexts, such as upgrades, "non-CDB" refers to a non-CDB from a previous release.

In a CDB, the requirements for a role name are as follows:

- The name of a **common role** must begin with characters that are a case-insensitive match to the prefix specified by the `COMMON_USER_PREFIX` initialization parameter. By default, the prefix is `C##`.
- The name of a **local role** must not begin with characters that are a case-insensitive match to the prefix specified by the `COMMON_USER_PREFIX` initialization parameter. Regardless of the value of `COMMON_USER_PREFIX`, the name of a local role can never begin with `C##` or `c##`.

> **Note:**
>
> If the value of `COMMON_USER_PREFIX` is an empty string, then there are no requirements for common or local role names with one exception: the name of a local role can never begin with `C##` or `c##`. Oracle recommends against using an empty string value because it might result in conflicts between the names of local and common roles when a PDB is plugged into a different CDB, or when opening a PDB that was closed when a common user was created.

Some roles are defined by SQL scripts provided on your distribution media.

> **See Also:**
>
> GRANT for a list of these predefined roles and SET ROLE for information on enabling and disabling roles for a user

NOT IDENTIFIED Clause

Specify `NOT IDENTIFIED` to indicate that this role is authorized by the database and that no password is required to enable the role.

IDENTIFIED Clause

Use the `IDENTIFIED` clause to indicate that a user must be authorized by the specified method before the role is enabled with the `SET ROLE` statement.

BY *password*

You can create a **local role** with a password with the `BY` *`password`* clause. This means that you must specify the password to the database at the time you enable the role.

The password can contain any characters from the database character set except the `NULL` character `(CHR(0))` and the double-quote. The maximum length of the password is 1024 bytes. The password is syntactically an identifier, and may need to be enclosed in double-quotes as required by the "Database Object Naming Rules ". You must ensure that your database, and the clients that need to enable the role are configured to support all the characters comprising the password.

You can enable password-protected roles in a proxy session. Both secure application role and password-protected roles provide a secure method for enabling a role in a session. Oracle recommends using secure password roles instead of password protected roles in instances where the password has to be maintained and transmitted over insecure channels, or if more than one person needs to know the password. Password-protected roles in a proxy session are suitable for situations where automation is used to set the role.

USING *package*

The `USING` *`package`* clause lets you create a **secure application role**, which is a role that can be enabled only by applications using an authorized package. If you do not specify *`schema`*, then the database assumes the package is in your own schema.

See Also:

Oracle Database Security Guide for information on creating a secure application role

EXTERNALLY

Specify `EXTERNALLY` to create an **external role.** An external user must be authorized by an external service, such as an operating system or third-party service, before enabling the role.

Depending on the operating system, the user may have to specify a password to the operating system before the role is enabled.

GLOBALLY

Specify `GLOBALLY` to create a **global role** . A global user must be authorized to use the role by the enterprise directory service before the role is enabled at login.

Specify `GLOBALLY` with `AS` to map a directory group to a global role when using centrally managed users. The directory group is identified by its domain name.

Example: Map a Directory User to a Global User

```
  CREATE USER scott_global IDENTIFIED GLOBALLY AS ‘cn=scott
taylor,ou=sales,dc=abccorp,dc=com’;
```

This effectively maps a directory user named ‘scott taylor’ in the ‘sales’ organization unit of the abccorp.com domain to a database global user ‘scott_global’.

You can map an Oracle Database global role to an Azure app role in order to give Azure users and applications additional privileges and roles beyond those that they have through their login schemas.

Example: Map an Oracle Database Global Role to an App Role

The example creates a new database global role *`widget_sales_role`* and maps it to an existing Azure AD application role *`WidgetManagerGroup`*:

```
CREATE ROLE widget_sales_role IDENTIFIED GLOBALLY AS 'AZURE_ROLE=WidgetManagerGroup';
```

See Also:

Authenticating and Authorizing Microsoft Azure Active Directory Users for Oracle Autonomous Databases

CONTAINER Clause

The `CONTAINER` clause applies when you are connected to a CDB. However, it is not necessary to specify the `CONTAINER` clause because its default values are the only allowed values.

- To create a common role, you must be connected to the root. You can optionally specify `CONTAINER = ALL`, which is the default when you are connected to the root.
- To create a local role, you must be connected to a PDB. You can optionally specify `CONTAINER = CURRENT`, which is the default when you are connected to a PDB.

Examples

Creating a Role: Example

The following statement creates the role `dw_manager`:

```
CREATE ROLE dw_manager;
```

Users who are subsequently granted the `dw_manager` role will inherit all of the privileges that have been granted to this role.

You can add a layer of security to roles by specifying a password, as in the following example:

```
CREATE ROLE dw_manager
   IDENTIFIED BY warehouse;
```

Users who are subsequently granted the `dw_manager` role must specify the password `warehouse` to enable the role with the `SET ROLE` statement.

The following statement creates global role `warehouse_user`:

```
CREATE ROLE warehouse_user IDENTIFIED GLOBALLY;
```

The following statement creates the same role as an external role:

```
CREATE ROLE warehouse_user IDENTIFIED EXTERNALLY;
```

The following statement creates local role `role1` in the current PDB. The current container must be a PDB when you issue this statement:

```
CREATE ROLE role1 CONTAINER = CURRENT;
```

The following statement creates common role `c##role1`. The current container must be the root when you issue this statement:

```
CREATE ROLE c##role1 CONTAINER = ALL;
```

CREATE ROLLBACK SEGMENT

Note:

Oracle strongly recommends that you run your database in automatic undo management mode instead of using rollback segments. Do not use rollback segments unless you must do so for compatibility with earlier versions of Oracle Database. Refer to *Oracle Database Administrator's Guide* for information on automatic undo management.

Purpose

Use the `CREATE ROLLBACK SEGMENT` statement to create a **rollback segment**, which is an object that Oracle Database uses to store data necessary to reverse, or undo, changes made by transactions.

The information in this section assumes that your database is not running in automatic undo mode (the `UNDO_MANAGEMENT` initialization parameter is set to `MANUAL` or not set at all). If your database is running in automatic undo mode (the `UNDO_MANAGEMENT` initialization parameter is set to `AUTO`, which is the default), then rollback segments are not permitted. However, errors generated in rollback segment operations are suppressed.

Further, if your database has a locally managed `SYSTEM` tablespace, then you cannot create rollback segments in any dictionary-managed tablespace. Instead, you must either use the automatic undo management feature or create locally managed tablespaces to hold the rollback segments.

Note:

A tablespace can have multiple rollback segments. Generally, multiple rollback segments improve performance.

- The tablespace must be online for you to add a rollback segment to it.
- When you create a rollback segment, it is initially offline. To make it available for transactions by your Oracle Database instance, bring it online using the `ALTER ROLLBACK SEGMENT` statement. To bring it online automatically whenever you start up the database, add the segment name to the value of the `ROLLBACK_SEGMENT` initialization parameter.

To use objects in a tablespace other than the `SYSTEM` tablespace:

- If you are using rollback segments for undo, then at least one rollback segment (other than the `SYSTEM` rollback segment) must be online.
- If you are running the database in automatic undo mode, then at least one `UNDO` tablespace must be online.

See Also:

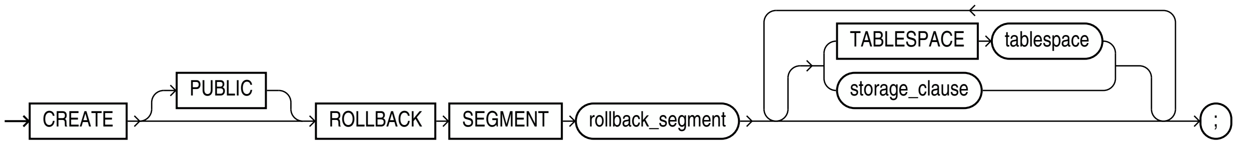
- ALTER ROLLBACK SEGMENT for information on altering a rollback segment
- DROP ROLLBACK SEGMENT for information on removing a rollback segment
- *Oracle Database Reference* for information on the `UNDO_MANAGEMENT` parameter
- *Oracle Database Administrator's Guide* for information on automatic undo mode

Prerequisites

To create a rollback segment, you must have the `CREATE ROLLBACK SEGMENT` system privilege.

Syntax

***create_rollback_segment*::=**



(*storage_clause*)

Semantics

PUBLIC

Specify `PUBLIC` to indicate that the rollback segment is public and is available to any instance. If you omit this clause, then the rollback segment is private and is available only to the instance naming it in its initialization parameter `ROLLBACK_SEGMENTS`.

rollback_segment

Specify the name of the rollback segment to be created. The name must satisfy the requirements listed in "Database Object Naming Rules ".

TABLESPACE

Use the `TABLESPACE` clause to identify the tablespace in which the rollback segment is created. If you omit this clause, then the database creates the rollback segment in the `SYSTEM` tablespace.

Note:

Oracle Database must access rollback segments frequently. Therefore, Oracle strongly recommends that you do not create rollback segments in the `SYSTEM` tablespace, either explicitly or implicitly by omitting this clause. In addition, to avoid high contention for the tablespace containing the rollback segment, it should not contain other objects such as tables and indexes, and it should require minimal extent allocation and deallocation.

To achieve these goals, create rollback segments in locally managed tablespaces with autoallocation disabled—in tablespaces created with the `EXTENT MANAGEMENT LOCAL` clause with the `UNIFORM` setting. The `AUTOALLOCATE` setting is not supported.

See Also:

CREATE TABLESPACE

storage_clause

The *`storage_clause`* lets you specify storage characteristics for the rollback segment.

- The `OPTIMAL` parameter of the *`storage_clause`* is of particular interest, because it applies only to rollback segments.
- You cannot specify the `PCTINCREASE` parameter of the *`storage_clause`* with `CREATE ROLLBACK SEGMENT`.

See Also:

storage_clause

Examples

Creating a Rollback Segment: Example

The following statement creates a rollback segment with default storage values in an appropriately configured tablespace:

```
CREATE TABLESPACE rbs_ts
   DATAFILE 'rbs01.dbf' SIZE 10M
   EXTENT MANAGEMENT LOCAL UNIFORM SIZE 100K;

/* This example and the next will fail if your database is in
   automatic undo mode.
*/
CREATE ROLLBACK SEGMENT rbs_one
   TABLESPACE rbs_ts;
```

The preceding statement is equivalent to the following:

```
CREATE ROLLBACK SEGMENT rbs_one
   TABLESPACE rbs_ts
```

```
STORAGE
( INITIAL 10K );
```

CREATE SCHEMA

Purpose

Use the `CREATE SCHEMA` statement to create multiple tables and views and perform multiple grants in your own schema in a single transaction.

To execute a `CREATE SCHEMA` statement, Oracle Database executes each included statement. If all statements execute successfully, then the database commits the transaction. If any statement results in an error, then the database rolls back all the statements.

Note:

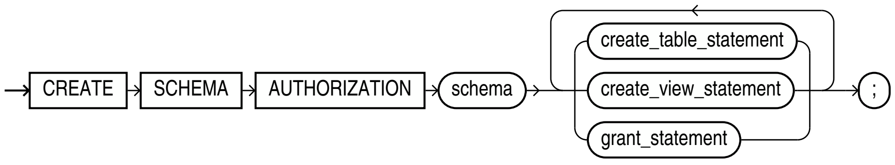
This statement does not actually create a schema. Oracle Database automatically creates a schema when you create a user (see CREATE USER). This statement lets you populate your schema with tables and views and grant privileges on those objects without having to issue multiple SQL statements in multiple transactions.

Prerequisites

The `CREATE SCHEMA` statement can include `CREATE TABLE`, `CREATE VIEW`, and `GRANT` statements. To issue a `CREATE SCHEMA` statement, you must have the privileges necessary to issue the included statements.

Syntax

***create_schema*::=**



Semantics

schema

Specify the name of the schema. The schema name must be the same as your Oracle Database username.

Restrictions

While `CREATE SCHEMA` supports `CREATE TABLE` , `CREATE BLOCKCHAIN TABLE` is unsupported.

create_table_statement

Specify a `CREATE TABLE` statement to be issued as part of this `CREATE SCHEMA` statement. Do not end this statement with a semicolon (or other terminator character).

See Also:

CREATE TABLE

create_view_statement

Specify a `CREATE VIEW` statement to be issued as part of this `CREATE SCHEMA` statement. Do not end this statement with a semicolon (or other terminator character).

See Also:

CREATE VIEW

grant_statement

Specify a `GRANT` statement to be issued as part of this `CREATE SCHEMA` statement. Do not end this statement with a semicolon (or other terminator character). You can use this clause to grant object privileges on objects you own to other users. You can also grant system privileges to other users if you were granted those privileges `WITH ADMIN OPTION`.

See Also:

GRANT

The `CREATE SCHEMA` statement supports the syntax of these statements only as defined by standard SQL, rather than the complete syntax supported by Oracle Database.

The order in which you list the `CREATE TABLE`, `CREATE VIEW`, and `GRANT` statements is unimportant. The statements within a `CREATE SCHEMA` statement can reference existing objects or objects you create in other statements within the same `CREATE SCHEMA` statement.

Restriction on Granting Privileges to a Schema

The syntax of the *`parallel_clause`* is allowed for a `CREATE TABLE` statement in `CREATE SCHEMA`, but parallelism is not used when creating the objects.

See Also:

The *parallel_clause* in the `CREATE TABLE` documentation

Examples

Creating a Schema: Example

The following statement creates a schema named `oe` for the sample order entry user `oe`, creates the table `new_product`, creates the view `new_product_view`, and grants the `SELECT` object privilege on `new_product_view` to the sample human resources user `hr`.

```
CREATE SCHEMA AUTHORIZATION oe
   CREATE TABLE new_product
      (color VARCHAR2(10)  PRIMARY KEY, quantity NUMBER)
   CREATE VIEW new_product_view
      AS SELECT color, quantity FROM new_product WHERE color = 'RED'
   GRANT select ON new_product_view TO hr;
```