13

SQL Statements: COMMIT to CREATE JSON RELATIONAL DUALITY VIEW

This chapter contains the following SQL statements:

COMMIT

Purpose

Use the `COMMIT` statement to end your current transaction and make permanent all changes performed in the transaction. A **transaction** is a sequence of SQL statements that Oracle Database treats as a single unit. This statement also erases all savepoints in the transaction and releases transaction locks.

Until you commit a transaction:

- You can see any changes you have made during the transaction by querying the modified tables, but other users cannot see the changes. After you commit the transaction, the changes are visible to other users' statements that execute after the commit.
- You can roll back (undo) any changes made during the transaction with the `ROLLBACK` statement (see ROLLBACK).

Oracle Database issues an implicit `COMMIT` under the following circumstances:

- Before any syntactically valid data definition language (DDL) statement, even if the statement results in an error
- After any data definition language (DDL) statement that completes without an error

You can also use this statement to:

- Commit an in-doubt distributed transaction manually
- Terminate a read-only transaction begun by a `SET TRANSACTION` statement

Oracle recommends that you explicitly end every transaction in your application programs with a `COMMIT` or `ROLLBACK` statement, including the last transaction, before disconnecting from Oracle Database. If you do not explicitly commit the transaction and the program terminates abnormally, then the last uncommitted transaction is automatically rolled back.

A normal exit from most Oracle utilities and tools causes the current transaction to be committed. A normal exit from an Oracle precompiler program does not commit the transaction and relies on Oracle Database to roll back the current transaction.

> **See Also:**
>
> - *Oracle Database Concepts* for more information on transactions
> - SET TRANSACTION for more information on specifying characteristics of a transaction

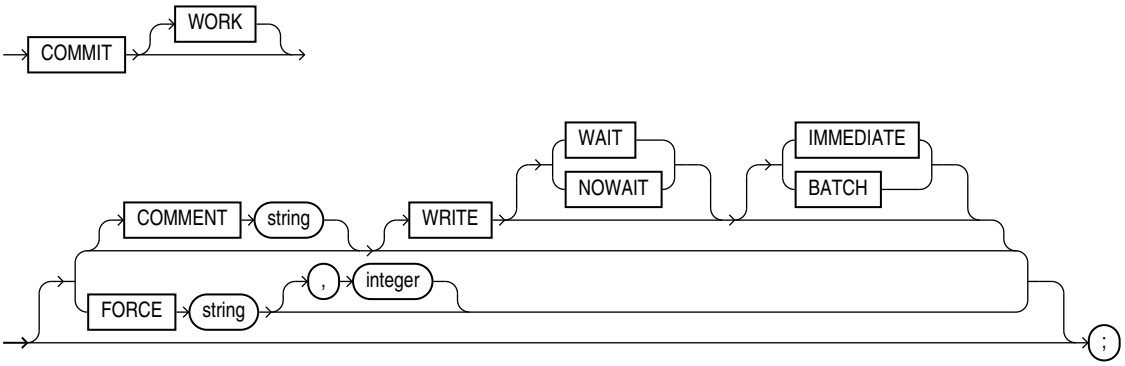
Prerequisites

You need no privileges to commit your current transaction.

To manually commit a distributed in-doubt transaction that you originally committed, you must have `FORCE TRANSACTION` system privilege. To manually commit a distributed in-doubt transaction that was originally committed by another user, you must have `FORCE ANY TRANSACTION` system privilege.

Syntax

***commit*::=**



Semantics

COMMIT

All clauses after the `COMMIT` keyword are optional. If you specify only `COMMIT`, then the default is `COMMIT WORK WRITE WAIT IMMEDIATE`.

WORK

The `WORK` keyword is supported for compliance with standard SQL. The statements `COMMIT` and `COMMIT WORK` are equivalent.

COMMENT Clause

This clause is supported for backward compatibility. Oracle recommends that you use named transactions instead of commit comments.

> **See Also:**
>
> SET TRANSACTION and *Oracle Database Concepts* for more information on named transactions

Specify a comment to be associated with the current transaction. The '*text*' is a quoted literal of up to 255 bytes that Oracle Database stores in the data dictionary view `DBA_2PC_PENDING` along with the transaction ID if a distributed transaction becomes in doubt. This comment can help you diagnose the failure of a distributed transaction.

> **See Also:**
>
> COMMENT for more information on adding comments to SQL statements

WRITE Clause

Use this clause to specify the priority with which the redo information generated by the commit operation is written to the redo log. This clause can improve performance by reducing latency, thus eliminating the wait for an I/O to the redo log. Use this clause to improve response time in environments with stringent response time requirements where the following conditions apply:

- The volume of update transactions is large, requiring that the redo log be written to disk frequently.
- The application can tolerate the loss of an asynchronously committed transaction.
- The latency contributed by waiting for the redo log write to occur contributes significantly to overall response time.

You can specify the `WAIT` | `NOWAIT` and `IMMEDIATE` | `BATCH` clauses in any order.

> **Note:**
>
> If you omit this clause, then the behavior of the commit operation is controlled by the `COMMIT_LOGGING` and `COMMIT_WAIT` initialization parameters, if they have been set.

WAIT | NOWAIT

Use these clauses to specify when control returns to the user.

- The `WAIT` parameter ensures that the commit will return only after the corresponding redo is persistent in the online redo log. Whether in `BATCH` or `IMMEDIATE` mode, when the client receives a successful return from this `COMMIT` statement, the transaction has been committed to durable media. A crash occurring after a successful write to the log can prevent the success message from returning to the client. In this case the client cannot tell whether or not the transaction committed.
- The `NOWAIT` parameter causes the commit to return to the client whether or not the write to the redo log has completed. This behavior can increase transaction throughput. With the `WAIT` parameter, if the commit message is received, then you can be sure that no data has been lost.

> **Note:**
>
> With `NOWAIT`, a crash occurring after the commit message is received, but before the redo log record(s) are written, can falsely indicate to a transaction that its changes are persistent.

If you omit this clause, then the transaction commits with the `WAIT` behavior.

IMMEDIATE | BATCH

Use these clauses to specify when the redo is written to the log.

- The `IMMEDIATE` parameter causes the log writer process (LGWR) to write the transaction's redo information to the log. This operation option forces a disk I/O, so it can reduce transaction throughput.

- The `BATCH` parameter causes the redo to be buffered to the redo log, along with other concurrently executing transactions. When sufficient redo information is collected, a disk write of the redo log is initiated. This behavior is called "group commit", as redo for multiple transactions is written to the log in a single I/O operation.

If you omit this clause, then the transaction commits with the `IMMEDIATE` behavior.

> **See Also:**
>
> *Oracle Database Concepts* for more information on asynchronous commit

FORCE Clause

In a distributed database system, the `FORCE` *`string`* [, *`integer`*] clause lets you manually commit an in-doubt distributed transaction. The transaction is identified by the '*`string`*' containing its local or global transaction ID. To find the IDs of such transactions, query the data dictionary view `DBA_2PC_PENDING`. You can use *`integer`* to specifically assign the transaction a system change number (SCN). If you omit *`integer`*, then the transaction is committed using the current SCN.

> **Note:**
>
> A `COMMIT` statement with a `FORCE` clause commits only the specified transactions. Such a statement does not affect your current transaction.

> **See Also:**
>
> *Oracle Database Administrator's Guide* for more information on these topics

Examples

Committing an Insert: Example

This statement inserts a row into the `hr.regions` table and commits this change:

```
INSERT INTO regions VALUES (5, 'Antarctica');

COMMIT WORK;
```

To commit the same insert operation and instruct the database to buffer the change to the redo log, without initiating disk I/O, use the following `COMMIT` statement:

```
COMMIT WRITE BATCH;
```

Commenting on COMMIT: Example

The following statement commits the current transaction and associates a comment with it:

```
COMMIT
    COMMENT 'In-doubt transaction Code 36, Call (415) 555-2637';
```

If a network or machine failure prevents this distributed transaction from committing properly, then Oracle Database stores the comment in the data dictionary along with the transaction ID. The comment indicates the part of the application in which the failure occurred and provides information for contacting the administrator of the database where the transaction was committed.

Forcing an In-Doubt Transaction: Example

The following statement manually commits a hypothetical in-doubt distributed transaction. Query the `V$CORRUPT_XID_LIST` data dictionary view to find the transaction IDs of corrupt transactions. You must have DBA privileges to view the `V$CORRUPT_XID_LIST` and to issue this statement.

```
COMMIT FORCE '22.57.53';
```

CREATE ANALYTIC VIEW

Purpose

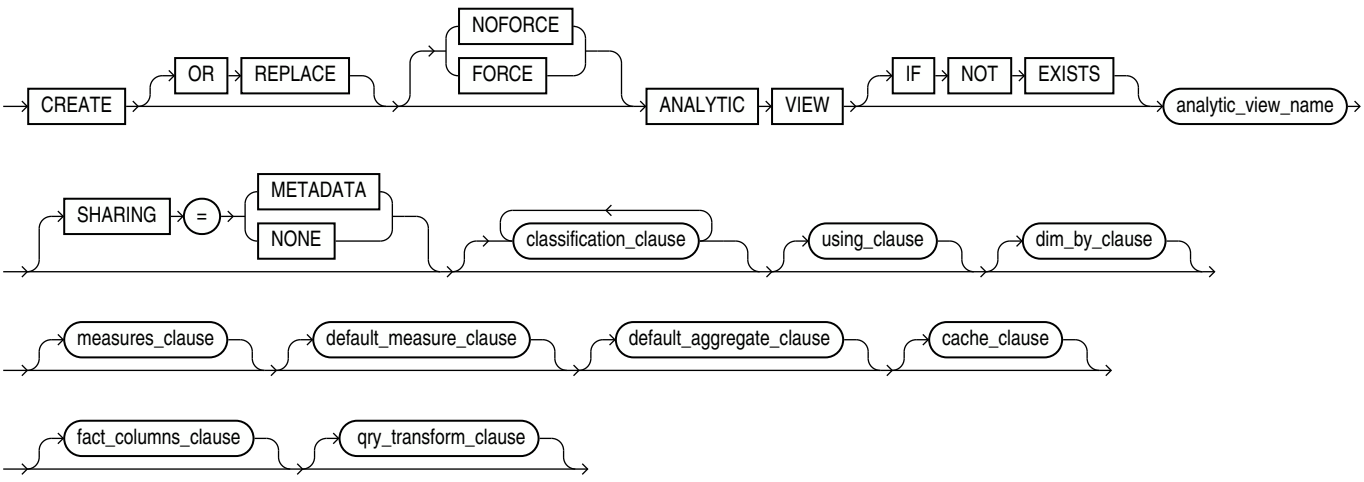
Use the `CREATE ANALYTIC VIEW` statement to create an analytic view. An analytic view specifies the source of its fact data and defines measures that describe calculations or other analytic operations to perform on the data. An analytic view also specifies the attribute dimensions and hierarchies that define the rows of the analytic view.

Prerequisites

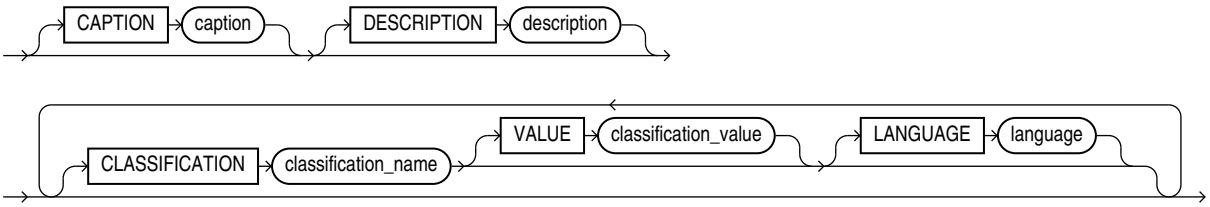
To create an analytic view in your own schema, you must have the `CREATE ANALYTIC VIEW` system privilege. To create an analytic view in another user's schema, you must have the `CREATE ANY ANALYTIC VIEW` system privilege.

Syntax

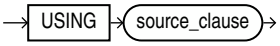
***create_analytic_view*::=**



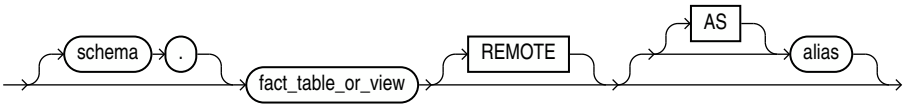
***classification_clause*::=**



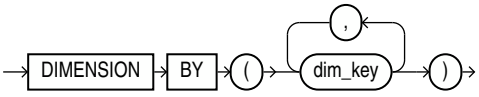
***using_clause*::=**



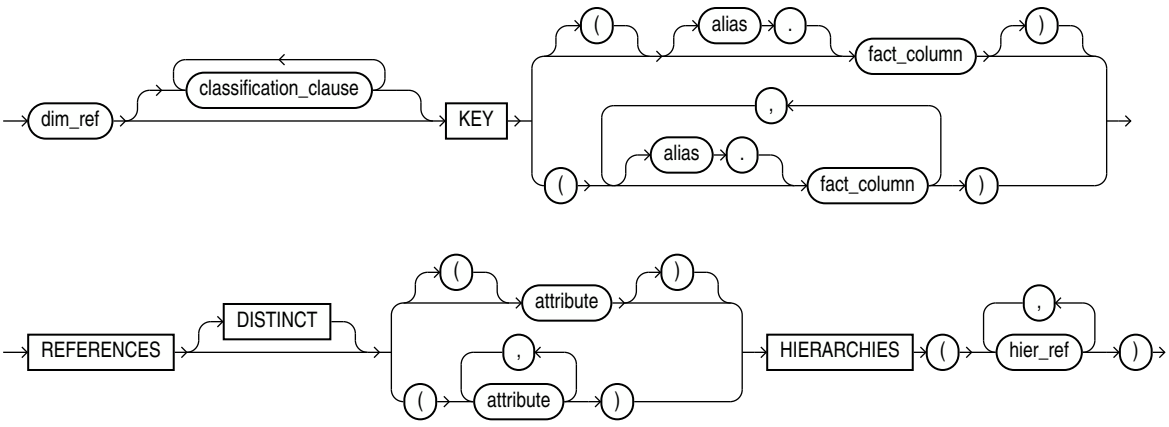
***source_clause*::=**



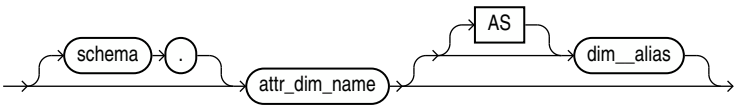
***dim_by_clause*::=**



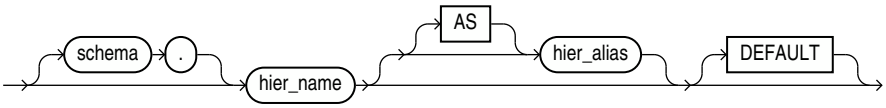
***dim_key*::=**



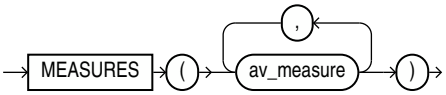
***dim_ref*::=**



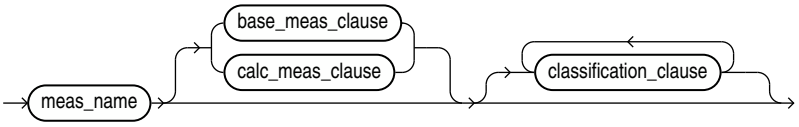
***hier_ref*::=**



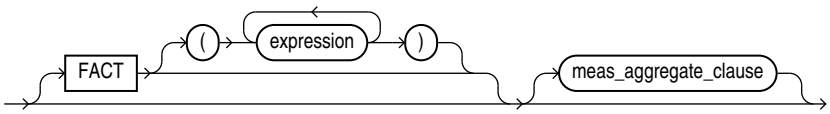
***measures_clause*::=**



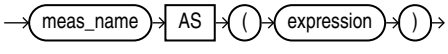
***av_measure*::=**



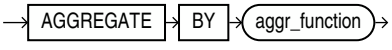
***base_meas_clause*::=**



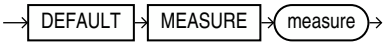
***calc_meas_clause*::=**



***meas_aggregate_clause*::=**

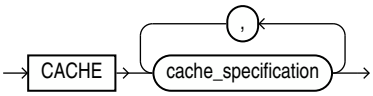


***default_measure_clause*::=**

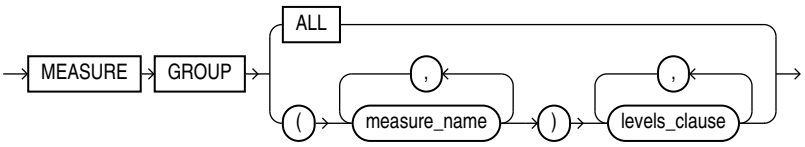


***default_aggregate_clause*::=**

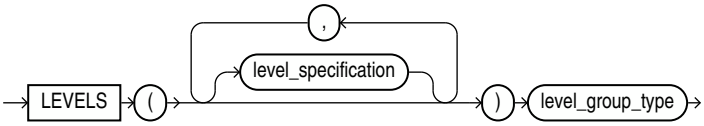
***cache_clause*::=**



***cache_specification*::=**

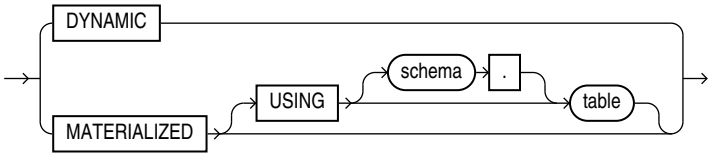


***levels_clause*::=**

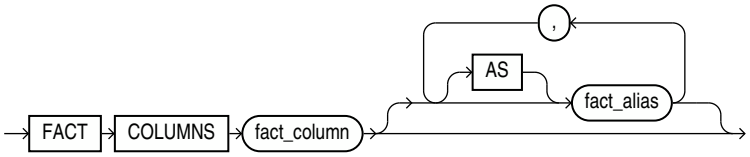


***level_specification*::=**

***level_group_type*::=**



***fact_columns_clause*::=**



***qry_transform_clause*::=**

Semantics

OR REPLACE

Specify `OR REPLACE` to replace an existing definition of an analytic view with a different definition.

FORCE and NOFORCE

Specify `FORCE` to force the creation of the analytic view even if it does not successfully compile. If you specify `NOFORCE`, then the analytic view must compile successfully, otherwise an error occurs. The default is `NOFORCE`.

IF NOT EXISTS

Specifying `IF NOT EXISTS` has the following effects:

- If the analytic view does not exist, a new analytic view is created at the end of the statement.
- If the analytic view exists, this is the analytic view 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 in which to create the analytic view. If you do not specify a schema, then Oracle Database creates the analytic view in your own schema.

analytic_view_name

Specify a name for the analytic view.

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.

classification_clause

Use the classification clause to specify values for the `CAPTION` or `DESCRIPTION` classifications and to specify user-defined classifications. Classifications provide descriptive metadata that applications may use to provide information about analytic views and their components.

You may specify any number of classifications for the same object. A classification can have a maximum length of 4000 bytes.

For the `CAPTION` and `DESCRIPTION` classifications, you may use the DDL shortcuts `CAPTION '`*`caption`*`'` and `DESCRIPTION '`*`description`*`'` or the full classification syntax.

You may vary the classification values by language. To specify a language for the `CAPTION` or `DESCRIPTION` classification, you must use the full syntax. If you do not specify a language, then the language value for the classification is `NULL`. The language value must either be `NULL` or a valid `NLS_LANGUAGE` value.

using_clause

Use this clause to declare the sources that you want to use to build the analytic view.

source_clause

You can specify any of the following sources to build an analytic view:

- A fact table or a view.
- External tables and remote tables.
- A table or a view in another schema. You can specify an alias for the table or the view.

REMOTE

Specify `REMOTE` on a given source to indicate to the analytic view that the given source is backed by remote data and should be optimized as remote data.

dim_by_clause

Specify the attribute dimensions of the analytic view.

dim_key

Specify an attribute dimension, columns of the fact table, columns of the attribute dimension, and hierarchies that are related in the analytic view.

With the `KEY` keyword, specify one or more columns in the fact table.

With the `REFERENCES` keyword, specify attributes of the attribute dimensions that the analytic view is dimensioned by. Each attribute must be a level key. The `DISTINCT` keyword supports the use of denormalized fact tables, in which the attribute dimension and fact data are in the same table. Use `REFERENCES DISTINCT` when an attribute dimension is defined using the fact table.

With the `HIERARCHIES` keyword, specify the hierarchies in the analytic view that use the attribute dimension.

dim_ref

Specify an attribute dimension. You can specify an alias for an attribute dimension, which is required if you use the same dimension more than once or if you use multiple dimensions with the same name from different schemas.

hier_ref

Specify a hierarchy. You can specify an alias for a hierarchy. You can specify one of the hierarchies in the list as the default. If you do not specify a default, the first hierarchy in the list is the default.

measures_clause

Specify the measures for the analytic view.

av_measure

Define a measure using either a single fact column or an expression over measures in this analytic view. A measure can be either a base measure or a calculated measure.

base_measure_clause

Define a base measure by optionally specifying a fact column or a *`meas_aggregate_clause`*, or both. If you do not specify a fact column, then the analytic view uses the column of the fact table that has the same name as the measure. If a column by the same name does not exist, an error is raised.

calc_measure_clause

Define a calculated measure by specifying an analytic view expression. The expression may reference other measures in the analytic view, but may not reference fact columns. Calculated measures do not have an aggregate clause because they're computed over the aggregated base measures.

For the syntax and descriptions of analytic view expressions, see Analytic View Expressions.

default_measure_clause

Specify a measure to use as the default measure for the analytic view. If you do not specify a measure, the first measure defined is the default.

meas_aggregate_clausè

Specify a default aggregation function for a base measure. If you do not specify an aggregation function, then the function specified by the *`default_aggregate_clause`* is used.

aggr_function

The functions that you can aggregate by in the *meas_aggregate_clause* and *default_aggregate_clause* are the following: APPROX_COUNT_DISTINCT, APPROX_COUNT_DISTINCT_AGG, AVG, COUNT, MAX, MIN, STDDEV, STDDEV_POP, STDDEV_SAMP, SUM, VAR_POP, VAR_SAMP, and VARIANCE.

default_aggregate_clause

Specify a default aggregation function for all of the base measures in the analytic view. If you do not specify a default aggregation function, then the default value is SUM.

cache_clause

Specify a cache clause to improve query response time when an appropriate materialized view is available. You can specify one or more cache specifications.

cache_specification

Specify one or more measure groups for a cache clause. To include all measure groups, specify ALL. Each measure group can contain one or more measures and one or more level groupings. A level grouping can contain one or more level specifications.

level_specification

Specify one or more levels for a level grouping of a measure group for a cache specification. Specify only one level per hierarchy. A materialized view must exist that contains the aggregated values for the hierarchy level.

level_group_type

If you specify the USING clause, then the given table will be directly used at query time, if the analytic view determines that this is the best cache to use for the query. The typical shape of the cache is a column for each measure in the MEASURE GROUP plus a column per level key of each level in the cache. There is one row per member combination, across all given levels, that has at least one contributing row from the fact table. The column names of the given table must match a specific format so that the analytic view can identify which columns line up with which measures and level keys. The names of the columns can be retrieved from the DBMS_HIERARCHY package using the method GET_MV_SQL_FOR_AV_CACHE.

This method takes in the cache to generate SQL for and returns the SQL text for the cache. This SQL text can be used to create a materialized view for the cache. It can also be used to create an aggregate table using CREATE TABLE AS.

At compile time of the analytic view, the following checks will be made in regard to the materialized table:

- The table must exist and be accessible by the owner of the analytic view
- The columns of the table must include the expected cache columns

fact_columns_clause

Specify this clause to explictly state the fact columns that can be accessed by the dervided analytic view. You can aggregate any columns of the fact table that appear in *fact_columns_clause* at query time with the aggregation operator specified in the derived analytic view

If an alias is provided for the fact column, then the alias name must be used in the dervided analytic view. The alias defaults to the fact column name if not specified.

It is a semantic analysis error, if two or more fact columns are specified with the same name.

If you do not specify this clause, then no fact columns can be accessed for aggregation by the derived analytic view. This is the default.

qry_transform_clause

Specify this clause on an analytic view, if you want the view to participate in detecting queries that match its semantic model and transform it into an equivalent analytic view query if appropriate.

Restrictions

You cannot use *qry_transform_clause* on an analytic view in the following cases:

- When the analytic view contains an attribute dimension with more than one dimension table (either a snowflake or starflake schema)
- When a dimension table joins to the fact table at a level that is above the leaf level of the dimension (i.e. a `REFERENCES DISTINCT` join)
- When `NORELY` is specified and one or more base tables are remote tables

The new clause allows for an optional `RELY` or `NORELY` keyword. The default is `NORELY`.

The analytic view metadata can be viewed as a set of constraints on the underlying data. These constraints are not enforced by the database, but can be checked using the `DBMS_HIERARCHY.VALIDATE_ANALYTIC_VIEW` procedure.

The `RELY` keyword indicates that the constraints implied on the data by the analytic view metadata can be relied upon without validation when being considered for base table transform. If `NORELY` is specified, then the data must be in a valid state in relation to the metadata in order for the base table transform to take place.

Examples

The following is a description of the `SALES_FACT` table:

```
desc SALES_FACT
Name              Null? Type
----------------- ----- -------------
MONTH_ID                VARCHAR2(10)
CATEGORY_ID             NUMBER(6)
STATE_PROVINCE_ID       VARCHAR2(120)
UNITS                   NUMBER(6)
SALES                   NUMBER(12,2)
```

The following example creates the `SALES_AV` analytic view using the `SALES_FACT` table:

```
CREATE OR REPLACE ANALYTIC VIEW sales_av
USING sales_fact
DIMENSION BY
  (time_attr_dim                        -- An attribute dimension of time
data
    KEY month_id REFERENCES month_id
    HIERARCHIES (
```

```
      time_hier DEFAULT),
   product_attr_dim                       -- An attribute dimension of product
data
    KEY category_id REFERENCES category_id
    HIERARCHIES (
      product_hier DEFAULT),
   geography_attr_dim                     -- An attribute dimension of store
data
    KEY state_province_id
    REFERENCES state_province_id HIERARCHIES (
      geography_hier DEFAULT)
   )
MEASURES
 (sales FACT sales,                       -- A base measure
  units FACT units,                       -- A base measure
  sales_prior_period AS                   -- Calculated measures
    (LAG(sales) OVER (HIERARCHY time_hier OFFSET 1)),
  sales_year_ago AS
    (LAG(sales) OVER (HIERARCHY time_hier OFFSET 1
     ACROSS ANCESTOR AT LEVEL year)),
  chg_sales_year_ago AS
    (LAG_DIFF(sales) OVER (HIERARCHY time_hier OFFSET 1
     ACROSS ANCESTOR AT LEVEL year)),
  pct_chg_sales_year_ago AS
    (LAG_DIFF_PERCENT(sales) OVER (HIERARCHY time_hier OFFSET 1
     ACROSS ANCESTOR AT LEVEL year)),
  sales_qtr_ago AS
    (LAG(sales) OVER (HIERARCHY time_hier OFFSET 1
     ACROSS ANCESTOR AT LEVEL quarter)),
  chg_sales_qtr_ago AS
    (LAG_DIFF(sales) OVER (HIERARCHY time_hier OFFSET 1
     ACROSS ANCESTOR AT LEVEL quarter)),
  pct_chg_sales_qtr_ago AS
    (LAG_DIFF_PERCENT(sales) OVER (HIERARCHY time_hier OFFSET 1
     ACROSS ANCESTOR AT LEVEL quarter))
  )
DEFAULT MEASURE SALES;
```

CREATE ATTRIBUTE DIMENSION

Purpose

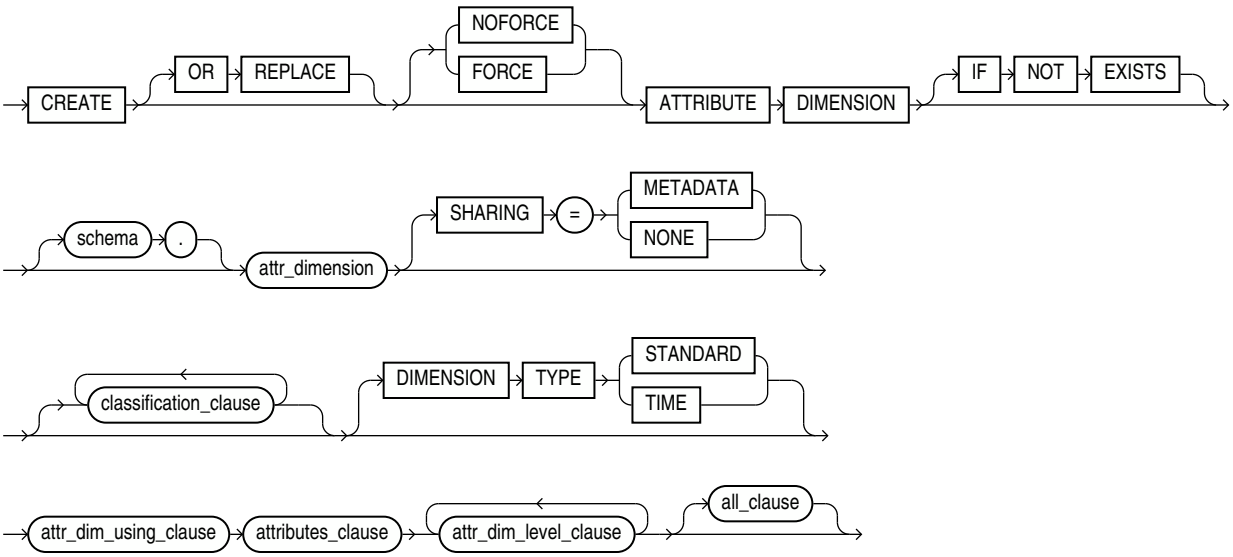
Use the `CREATE ATTRIBUTE DIMENSION` statement to create an attribute dimension. An attribute dimension specifies dimension members for one or more analytic view hierarchies. It specifies the data source it is using and the members it includes. It specifies levels for its members and determines attribute relationships between levels.

Prerequisites

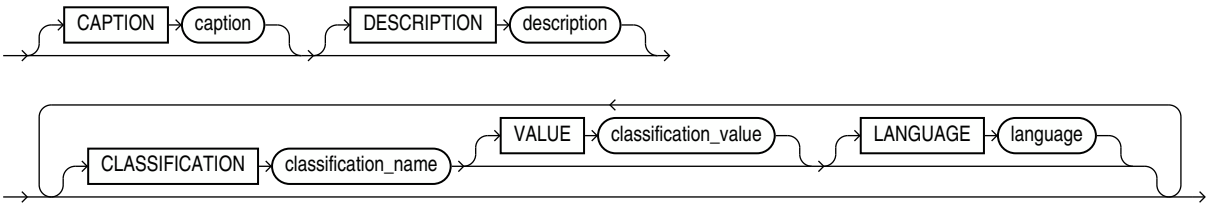
To create an attribute dimension in your own schema, you must have the `CREATE ATTRIBUTE DIMENSION` system privilege. To create an attribute dimension in another user's schema, you must have the `CREATE ANY ATTRIBUTE DIMENSION` system privilege.

Syntax

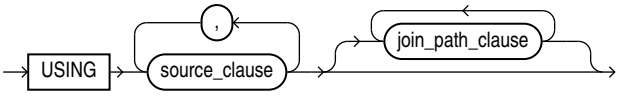
***create_attribute_dimension*::=**



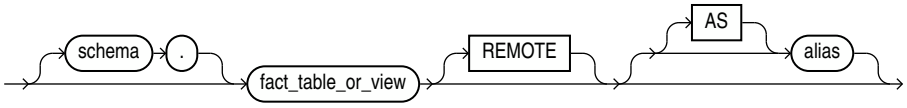
***classification_clause*::=**



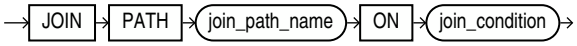
***attr_dim_using_clause*::=**



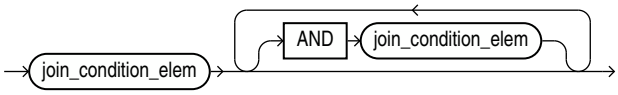
***source_clause*::=**



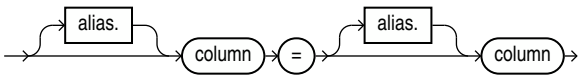
***join_path_clause*::=**



***join_condition*::=**

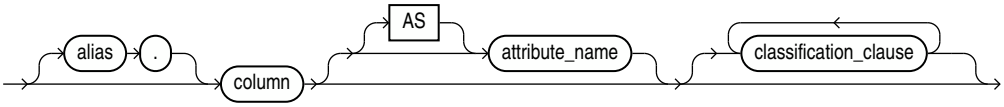


***join_condition_elem* ::=**

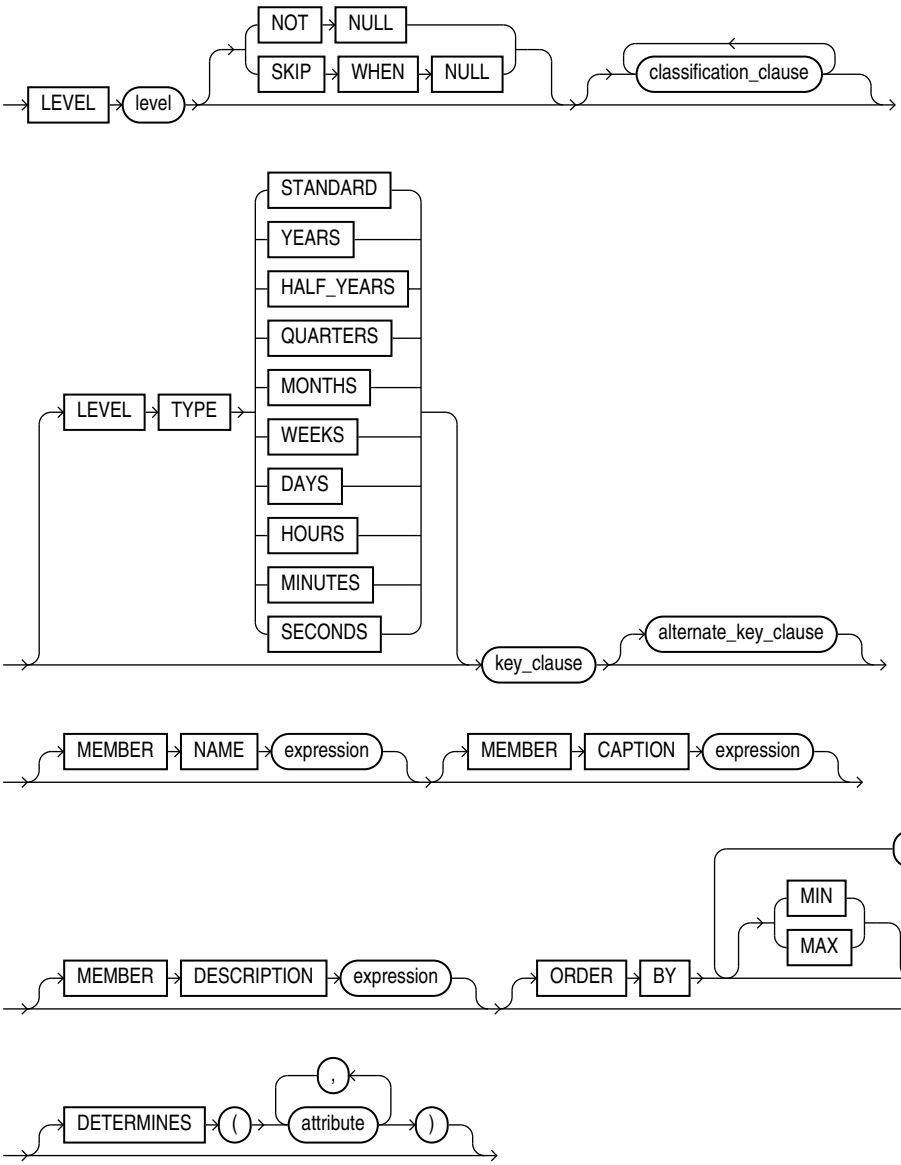


***attributes_clause*::=**

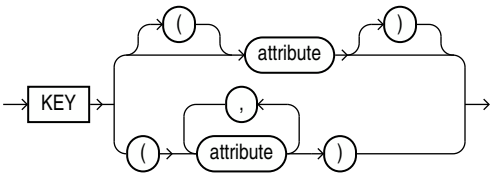
***attr_dim_attributes_clause*::=**



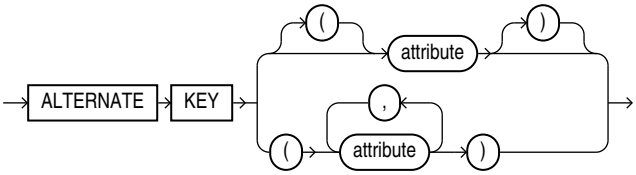
***attr_dim_level_clause*::=**



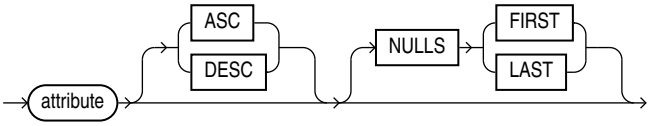
***key_clause*::=**



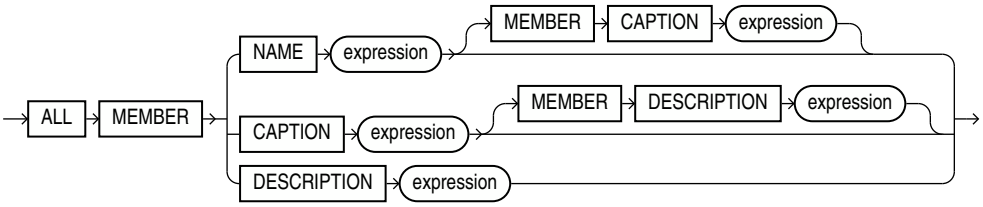
***alternate_key_clause*::=**



***dim_order_clause*::=**



***all_clause*::=**



Semantics

OR REPLACE

Specify `OR REPLACE` to replace an existing definition of an attribute dimension with a different definition.

FORCE and NOFORCE

Specify `FORCE` to force the creation of the attribute dimension even if it does not successfully compile. If you specify `NOFORCE`, then the attribute dimension must compile successfully, otherwise an error occurs. The default is `NOFORCE`.

IF NOT EXISTS

Specifying `IF NOT EXISTS` has the following effects:

- If the attribute dimension does not exist, a new attribute dimension is created at the end of the statement.
- If the attribute dimension exists, this is the attribute dimension 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 in which to create the attribute dimension. If you do not specify a schema, then Oracle Database creates the attribute dimension in your own schema.

attr_dimension

Specify a name for the attribute dimension.

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.

classification_clause

Use the classification clause to specify values for the `CAPTION` or `DESCRIPTION` classifications and to specify user-defined classifications. Classifications provide descriptive metadata that applications may use to provide information about analytic views and their components.

You may specify any number of classifications for the same object. A classification can have a maximum length of 4000 bytes.

For the `CAPTION` and `DESCRIPTION` classifications, you may use the DDL shortcuts `CAPTION 'caption'` and `DESCRIPTION 'description'` or the full classification syntax.

You may vary the classification values by language. To specify a language for the `CAPTION` or `DESCRIPTION` classification, you must use the full syntax. If you do not specify a language, then the language value for the classification is `NULL`. The language value must either be `NULL` or a valid `NLS_LANGUAGE` value.

DIMENSION TYPE

An attribute dimension may be either a `STANDARD` or a `TIME` type. A `STANDARD` type attribute dimension has `STANDARD` type levels. Each level of a `TIME` type attribute dimension is one of the time types. The default `DIMENSION TYPE` is `STANDARD`.

attr_dim_using_clause

Use this clause to declare the sources that you want to use to create the attribute dimension.

source_clause

You may specify the following sources:

- A table or a view.
- An alias for the table or the view by using the `AS` keyword.

- A join path. Use join paths to specify joins when the attribute dimension uses tables organized in a snowflake schema.

REMOTE

Specify `REMOTE` on a given source to indicate that the source is backed by remote data and should be optimized as remote data.

join_path_clause

The join path clause specifies a join condition between columns in different tables. The name for the join path specified by the *join_path_name* argument must be unique for each join path included in the `USING` clause.

join_condition

A join condition consists of one or more join condition elements; each additional join condition element is included by an `AND` operation.

join_condition_element

In a join condition element, the column references on the left-hand-side must come from a different table than the column references on the right-hand-side.

attributes_clause

Specify one or more *`attr_dim_attribute_clause`* clauses.

attr_dim_attribute_clause

Specify a column from the *`attr_dim_using_clause`* source. The attribute has the name of the column unless you specify an alias using the `AS` keyword. You may specify classifications for each attribute.

attr_dim_level_clause

Specify a level in the attribute dimension. A level specifies key and optional alternate key attributes that provide the members of the level.

If the key attribute has no `NULL` values, then you may specify `NOT NULL`, which is the default. If it does have one or more `NULL` values, then specify `SKIP WHEN NULL`.

LEVEL TYPE

A `STANDARD` type attribute dimension has `STANDARD` type levels. You do not need to specify a `LEVEL TYPE` for a `STANDARD` type attribute dimension.

In a `TIME` type attribute dimension, you must specify a level type. The type of the level may be one of the time types. You must specify a time type even if the values of the level members are not of that type. For example, you may have a SEASON level with values that are the names of seasons. In defining the level, you must specify any one of the time level types, such as `QUARTERS`. An application may use the level type designations for whatever purpose it chooses.

DETERMINES

With the `DETERMINES` keyword, you may specify other attributes of the attribute dimension that this level determines. If an attribute has only one value for each value of another attribute, then the value of the first attribute determines the value of the other attribute. For example, the QUARTER_ID attribute has only one value for each value of the MONTH_ID attribute, so you can include the the QUARTER_ID attribute in the `DETERMINES` phrase of the MONTHS level.

key_clause

Specify one or more attributes as the key for the level.

alternate_key_clause

Specify one or more attributes as the alternate key for the level.

dim_order_clause

Specify the ordering of the members of the level.

all_clause

Optionally specify `MEMBER NAME`, `MEMBER CAPTION`, and `MEMBER DESCRIPTION` values for the implicit ALL level. By default, the `MEMBER NAME` value is ALL.

Examples

The following example describes the TIME_DIM table:

```
desc TIME_DIM

Name               Null?   Type
------------------ -----   -------------
MONTH_ID                   VARCHAR2(10)
CATEGORY_ID                NUMBER(6)
STATE_PROVINCE_ID          VARCHAR2(120)
UNITS                      NUMBER(6)
SALES                      NUMBER(12,2)
YEAR_ID           NOT NULL VARCHAR2(30)
YEAR_NAME         NOT NULL VARCHAR2(40)
YEAR_END_DATE              DATE
QUARTER_ID        NOT NULL VARCHAR2(30)
QUARTER_NAME      NOT NULL VARCHAR2(40)
QUARTER_END_DATE           DATE
QUARTER_OF_YEAR            NUMBER
MONTH_ID          NOT NULL VARCHAR2(30)
MONTH_NAME        NOT NULL VARCHAR2(40)
MONTH_END_DATE             DATE
MONTH_OF_YEAR              NUMBER
MONTH_LONG_NAME            VARCHAR2(30)
SEASON                     VARCHAR2(10)
SEASON_ORDER               NUMBER(38)
MONTH_OF_QUARTER           NUMBER(38)
```

The following example creates a `TIME` type attribute dimension, using columns from the TIME_DIM table:

```
CREATE OR REPLACE ATTRIBUTE DIMENSION time_attr_dim
DIMENSION TYPE TIME
USING time_dim
ATTRIBUTES
 (year_id
   CLASSIFICATION caption VALUE 'YEAR_ID'
   CLASSIFICATION description VALUE 'YEAR ID',
```

```
  year_name
    CLASSIFICATION caption VALUE 'YEAR_NAME'
    CLASSIFICATION description VALUE 'Year',
  year_end_date
    CLASSIFICATION caption VALUE 'YEAR_END_DATE'
    CLASSIFICATION description VALUE 'Year End Date',
  quarter_id
    CLASSIFICATION caption VALUE 'QUARTER_ID'
    CLASSIFICATION description VALUE 'QUARTER ID',
  quarter_name
    CLASSIFICATION caption VALUE 'QUARTER_NAME'
    CLASSIFICATION description VALUE 'Quarter',
  quarter_end_date
    CLASSIFICATION caption VALUE 'QUARTER_END_DATE'
    CLASSIFICATION description VALUE 'Quarter End Date',
  quarter_of_year
    CLASSIFICATION caption VALUE 'QUARTER_OF_YEAR'
    CLASSIFICATION description VALUE 'Quarter of Year',
  month_id
    CLASSIFICATION caption VALUE 'MONTH_ID'
    CLASSIFICATION description VALUE 'MONTH ID',
  month_name
    CLASSIFICATION caption VALUE 'MONTH_NAME'
    CLASSIFICATION description VALUE 'Month',
  month_long_name
    CLASSIFICATION caption VALUE 'MONTH_LONG_NAME'
    CLASSIFICATION description VALUE 'Month Long Name',
  month_end_date
    CLASSIFICATION caption VALUE 'MONTH_END_DATE'
    CLASSIFICATION description VALUE 'Month End Date',
  month_of_quarter
    CLASSIFICATION caption VALUE 'MONTH_OF_QUARTER'
    CLASSIFICATION description VALUE 'Month of Quarter',
  month_of_year
    CLASSIFICATION caption VALUE 'MONTH_OF_YEAR'
    CLASSIFICATION description VALUE 'Month of Year',
  season
    CLASSIFICATION caption VALUE 'SEASON'
    CLASSIFICATION description VALUE 'Season',
  season_order
    CLASSIFICATION caption VALUE 'SEASON_ORDER'
    CLASSIFICATION description VALUE 'Season Order')
LEVEL month
  LEVEL TYPE MONTHS
  CLASSIFICATION caption VALUE 'MONTH'
  CLASSIFICATION description VALUE 'Month'
  KEY month_id
  MEMBER NAME month_name
  MEMBER CAPTION month_name
  MEMBER DESCRIPTION month_long_name
  ORDER BY month_end_date
  DETERMINES (month_end_date,
    quarter_id,
    season,
    season_order,
    month_of_year,
```

```
    month_of_quarter)
LEVEL quarter
  LEVEL TYPE QUARTERS
  CLASSIFICATION caption VALUE 'QUARTER'
  CLASSIFICATION description VALUE 'Quarter'
  KEY quarter_id
  MEMBER NAME quarter_name
  MEMBER CAPTION quarter_name
  MEMBER DESCRIPTION quarter_name
  ORDER BY quarter_end_date
  DETERMINES (quarter_end_date,
    quarter_of_year,
    year_id)
LEVEL year
  LEVEL TYPE YEARS
  CLASSIFICATION caption VALUE 'YEAR'
  CLASSIFICATION description VALUE 'Year'
  KEY year_id
  MEMBER NAME year_name
  MEMBER CAPTION year_name
  MEMBER DESCRIPTION year_name
  ORDER BY year_end_date
  DETERMINES (year_end_date)
LEVEL season
  LEVEL TYPE QUARTERS
  CLASSIFICATION caption VALUE 'SEASON'
  CLASSIFICATION description VALUE 'Season'
  KEY season
  MEMBER NAME season
  MEMBER CAPTION season
  MEMBER DESCRIPTION season
LEVEL month_of_quarter
  LEVEL TYPE MONTHS
  CLASSIFICATION caption VALUE 'MONTH_OF_QUARTER'
  CLASSIFICATION description VALUE 'Month of Quarter'
  KEY month_of_quarter;
```

The following example describes the PRODUCT_DIM table:

```
desc PRODUCT_DIM

Name            Null?    Type
--------------- -------- -------------
DEPARTMENT_ID   NOT NULL NUMBER
DEPARTMENT_NAME NOT NULL VARCHAR2(100)
CATEGORY_ID     NOT NULL NUMBER
CATEGORY_NAME   NOT NULL VARCHAR2(100)
```

The following example creates a `STANDARD` type attribute dimension, using columns from the PRODUCT_DIM table:

```
CREATE OR REPLACE ATTRIBUTE DIMENSION product_attr_dim
USING product_dim
ATTRIBUTES
 (department_id,
```

```
  department_name,
  category_id,
  category_name)
LEVEL DEPARTMENT
  KEY department_id
  ALTERNATE KEY department_name
  MEMBER NAME department_name
  MEMBER CAPTION department_name
  ORDER BY department_name
LEVEL CATEGORY
  KEY category_id
  ALTERNATE KEY category_name
  MEMBER NAME category_name
  MEMBER CAPTION category_name
  ORDER BY category_name
  DETERMINES(department_id)
ALL MEMBER NAME 'ALL PRODUCTS';
```

The following example describes the GEOGRAPHY_DIM table:

```
desc GEOGRAPHY_DIM

Name                Null?    Type
---------------     -------- -------------
DEPARTMENT_ID       NOT NULL NUMBER
DEPARTMENT_NAME     NOT NULL VARCHAR2(100)
CATEGORY_ID         NOT NULL NUMBER
CATEGORY_NAME       NOT NULL VARCHAR2(100)
REGION_ID           NOT NULL VARCHAR2(120)
REGION_NAME         NOT NULL VARCHAR2(100)
COUNTRY_ID          NOT NULL VARCHAR2(2)
COUNTRY_NAME        NOT NULL VARCHAR2(120)
STATE_PROVINCE_ID   NOT NULL VARCHAR2(120)
STATE_PROVINCE_NAME NOT NULL VARCHAR2(400)
```

The following example creates an `STANDARD` type attribute dimension, using columns from the GEOGRAPHY_DIM table:

```
CREATE OR REPLACE ATTRIBUTE DIMENSION geography_attr_dim
USING geography_dim
ATTRIBUTES
 (region_id,
  region_name,
  country_id,
  country_name,
  state_province_id,
  state_province_name)
LEVEL REGION
  KEY region_id
  ALTERNATE KEY region_name
  MEMBER NAME region_name
  MEMBER CAPTION region_name
  ORDER BY region_name
LEVEL COUNTRY
  KEY country_id
```

```
  ALTERNATE KEY country_name
  MEMBER NAME country_name
  MEMBER CAPTION country_name
  ORDER BY country_name
  DETERMINES(region_id)
LEVEL STATE_PROVINCE
  KEY state_province_id
  ALTERNATE KEY state_province_name
  MEMBER NAME state_province_name
  MEMBER CAPTION state_province_name
  ORDER BY state_province_name
  DETERMINES(country_id)
ALL MEMBER NAME 'ALL CUSTOMERS';
```

CREATE AUDIT POLICY (Unified Auditing)

This section describes the `CREATE AUDIT POLICY` statement for **unified auditing**. This type of auditing is new beginning with Oracle Database 12*c* and provides a full set of enhanced auditing features. Refer to *Oracle Database Security Guide* for more information on unified auditing.

Purpose

Use the `CREATE AUDIT POLICY` statement to create a unified audit policy.

> **See Also:**
>
> - ALTER AUDIT POLICY (Unified Auditing)
> - DROP AUDIT POLICY (Unified Auditing)
> - AUDIT (Unified Auditing)
> - NOAUDIT (Unified Auditing)

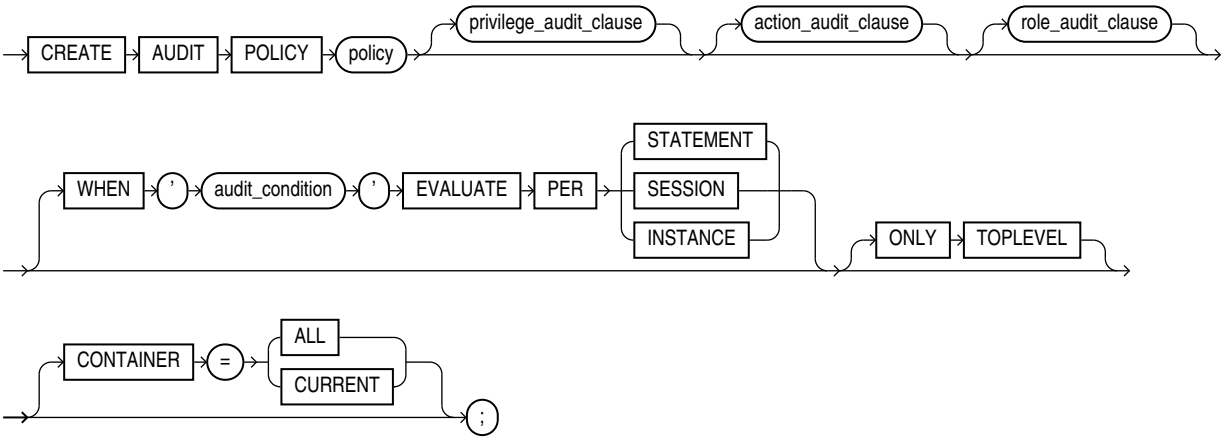
Prerequisites

You must have the `AUDIT SYSTEM` system privilege or the `AUDIT_ADMIN` role.

To specify the `CONTAINER` clause, you must be connected to a multitenant container database (CDB). To create a common unified audit policy, you must have the commonly granted `AUDIT SYSTEM` privilege or the `AUDIT_ADMIN` common role. To create a local unified audit policy, you must have the commonly granted `AUDIT SYSTEM` privilege or the `AUDIT_ADMIN` common role, or you must have the locally granted `AUDIT SYSTEM` privilege or the `AUDIT_ADMIN` local role in the container to which you are connected.

Syntax

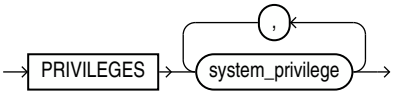
***create_audit_policy*::=**



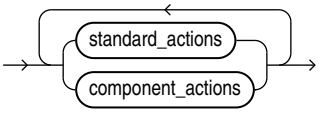
> **Note:**
>
> You must specify at least one of the clauses *privilege_audit_clause*, *action_audit_clause*, or *role_audit_clause*.

(*privilege_audit_clause*::=, *action_audit_clause*::=, *role_audit_clause*::=)

***privilege_audit_clause*::=**

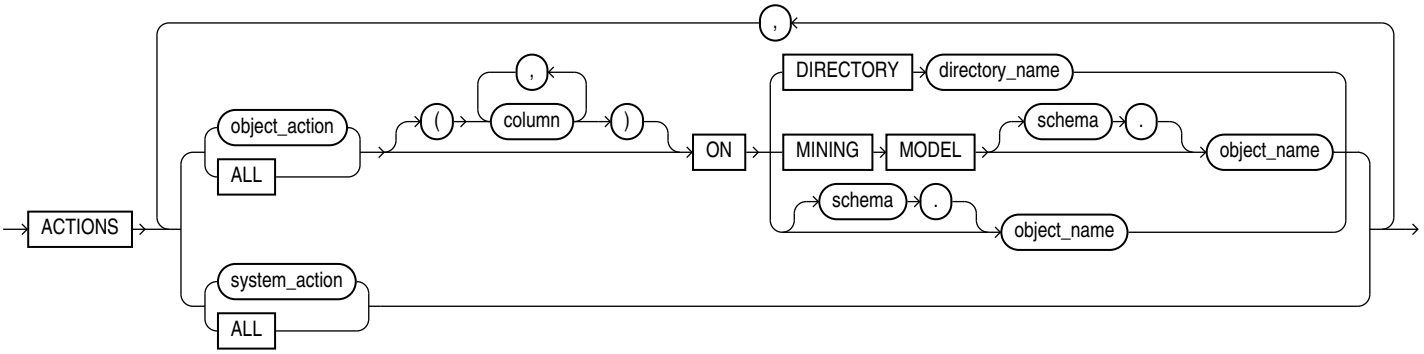


***action_audit_clause*::=**

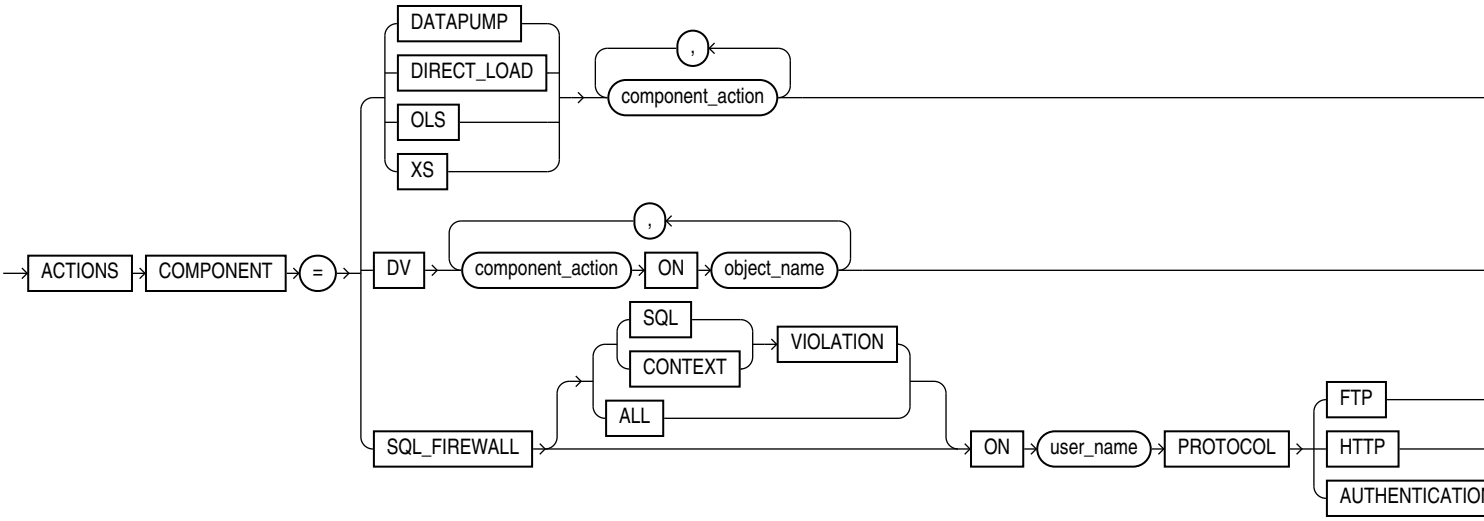


> **Note:**
>
> You can specify only the *standard_actions* clause, only the *component_actions* clause, or both clauses in either order, but you can specify each clause at most once.

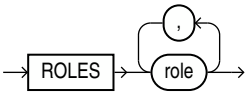
***standard_actions*::=**



***component_actions*::=**



***role_audit_clause*::=**



Semantics

policy

Specify the name of the unified audit policy to be created. The name of the policy must begin with the value of the `COMMON_USER_PREFIX` initialization parameter. The default value of the `COMMON_USER_PREFIX` parameter is `c##`.

The length of the audit policy name cannot exceed 128 bytes and must contain ASCII characters only.

These rules apply for application common audit policies as well. In this case, the value of the `COMMON_USER_PREFIX` is fetched from the application root. The default value in application root is an empty string.

The name must also satisfy the requirements listed in "Database Object Naming Rules ".

You can find the names of all unified audit policies by querying the `AUDIT_UNIFIED_POLICIES` view.

> **See Also:**
>
> *Oracle Database Reference* for more information on the `AUDIT_UNIFIED_POLICIES` view

privilege_audit_clause

Use this clause to audit one or more system privileges. For *`system_privilege`*, specify a valid system privilege. To view all valid system privileges, query the `NAME` column of the `SYSTEM_PRIVILEGE_MAP` view.

Only those SQL statements are audited, that successfully use system privileges. If a statement does not make use of a system privilege, it does not get audited with the *`privilege_audit_clause`*.

Restriction on Auditing System Privileges

You cannot audit the following system privileges: `INHERIT ANY PRIVILEGES`, `SYSASM`, `SYSBACKUP`, `SYSDBA`, `SYSDG`, `SYSKM`, `SYSRAC`, and `SYSOPER`.

action_audit_clause

Use this clause to specify one or more actions to be audited. Use the *`standard_actions`* clause to audit actions on standard RDBMS objects and to audit standard RDBMS system actions for the database. Use the *`component_actions`* clause to audit actions for components.

standard_actions

Use this clause to audit actions on standard RDBMS objects and to audit standard RDBMS system actions for the database.

You can also create unified audit policies to audit individual columns in tables and views. For examples on auditing columns see Examples

Note that column level audit policies generate audit records whenever the column is accessed.

***object_action* ON**

Use this clause to audit an action on the specified object. For *`object_action`*, specify the action to be audited. Table 13-1 lists the actions that can be audited on each type of object.

ALL ON

Use this clause to audit all actions on the specified object. All of the actions listed in Table 13-1 for the type of object that you specify in the `ON` clause will be audited.

ON Clause

Use the ON clause to specify the object to be audited. Directories and data mining models are identified separately because they reside in separate namespaces. To audit actions on a directory, specify ON DIRECTORY *directory_name*. To audit actions on a data mining model, specify ON MINING MODEL *object_name*. To audit actions on the other types of objects listed in Table 13-1, specify ON *object_name*. If you do not qualify *object_name* with *schema*, then the database assumes the object is in your own schema.

Table 13-1 Unified Auditing Objects and Actions

Type of Object	Actions
Directory	AUDIT, GRANT, READ
Function	AUDIT, EXECUTE (Notes 1 and 2), GRANT
Java Schema Objects (Source, Class, Resource)	AUDIT, EXECUTE, GRANT
Library	EXECUTE, GRANT
Materialized Views	ALTER, AUDIT, COMMENT, DELETE, INDEX, INSERT, LOCK, SELECT, UPDATE
Mining Model	AUDIT, COMMENT, GRANT, RENAME, SELECT
Object Type	ALTER, AUDIT, GRANT
Package	AUDIT, EXECUTE, GRANT
Procedure	AUDIT, EXECUTE (Notes 1 and 2), GRANT
Sequence	ALTER, AUDIT, GRANT, SELECT
Table	ALTER, AUDIT, COMMENT, DELETE, FLASHBACK, GRANT, INDEX, INSERT, LOCK, RENAME, SELECT, UPDATE, TRUNCATE
View	AUDIT, DELETE, FLASHBACK, GRANT, INSERT, LOCK, RENAME, SELECT, UPDATE

Note 1: When you audit the EXECUTE operation on a PL/SQL stored procedure or stored function, the database considers only its ability to find the procedure or function and authorize its execution when determining the success or failure of the operation for the purposes of auditing. Therefore, if you specify the WHENEVER NOT SUCCESSFUL clause, then only invalid object errors, non-existent object errors, and authorization failures are audited; errors encountered during the execution of the procedure or function are not audited. If you specify the WHENEVER SUCCESSFUL clause, then all executions that are not blocked by invalid object errors, non-existent object errors, or authorization failures are audited, regardless of whether errors are encountered during execution.

Note 2: To audit the failure of a recursive SQL operation inside a PL/SQL stored procedure or stored function, configure auditing for the SQL operation.

Note 3: The auditing of EXECUTE on a PL/SQL stored procedure, function, or package in the database happens during the instantiation phase of the procedure, function, or package.

Note 3: The auditing of the GRANT object audit option also audits the REVOKE audit option.

system_action

Use this clause to audit a system action for the database. To view the valid values for *system_action*, query the NAME column of the AUDITABLE_SYSTEM_ACTIONS view where COMPONENT is 'Standard'.

Example: Audit CHANGE PASSWORD in Unified Auditing

You can audit `CHANGE PASSWORD` system actions by configuring an audit policy to audit password changes. After you configure the audit policy, you must enable the audit policy.

Example: Create Audit Policy to Audit System Action Change Password

The following example creates an audit policy `mypolicy` to audit the action `CHANGE PASSWORD`:

```
CREATE AUDIT POLICY mypolicy ACTIONS CHANGE PASSWORD;
----------------------
Audit policy created.
```

Example: Enable Audit Policy Configured to Audit Password Changes

The following statement enables the audit policy `mypolicy`:

```
AUDIT POLICY mypolicy;
```

The audit policy `mypolicy` will now audit `CHANGE PASSWORD` actions for both successful and unsuccessful changes of password.

Example: Change Password

A user `hr_usr` with password `hr_pwd` can connect to PDB `hr_pdb` and change the password as follows:

```
CONNECT hr_usr/hr_pwd@hr_pdb;
PASSWORD
Changing password for hr_usr
Old password:
New password:
Retype new password:
Password changed.
```

In the SQL*Plus example above, the command `PASSWORD` run by user `hr_usr` initiates a `CHANGE PASSWORD` action that generates an audit record.

Example: Check Audit Trail for Password Changes

You can view the record by by querying the `UNIFIED_AUDIT_TRAIL` as follows:

```
SELECT ACTION_NAME, UNIFIED_AUDIT_POLICIES, OBJECT_NAME FROM UNIFIED_AUDIT_TRAIL;

ACTION_NAME
--------------------------------------------------------------
UNIFIED_AUDIT_POLICIES
--------------------------------------------------------------
OBJECT_NAME
--------------------------------------------------------------
CHANGE PASSWORD
MYPOLICY
HR_USR
```

Note that the audit policy `mypolicy` will not capture password changes via the `ALTER USER` statement.

ALL

Use this clause to audit all system actions for the database.

component_actions

Use this clause to audit actions for the following components: Oracle Data Pump, Oracle SQL*Loader Direct Path Load, Oracle Label Security, Oracle Database Real Application Security, Oracle Database Vault, and the transmission protocol.

DATAPUMP

Use this clause to audit actions for Oracle Data Pump. For *component_action*, specify the action to be audited. To view the valid actions for Oracle Data Pump, query the NAME column of the AUDITABLE_SYSTEM_ACTIONS view where COMPONENT is Datapump. For example:

```
SELECT name FROM auditable_system_actions WHERE component = 'Datapump';
```

Refer to *Oracle Database Security Guide* for complete information on auditing Oracle Data Pump.

DIRECT_LOAD

Use this clause to audit actions for Oracle SQL*Loader Direct Path Load. For *component_action*, specify the action to be audited. To view the valid actions for Oracle SQL*Loader Direct Path Load, query the NAME column of the AUDITABLE_SYSTEM_ACTIONS view where COMPONENT is Direct path API. For example:

```
SELECT name FROM auditable_system_actions WHERE component = 'Direct path API';
```

Refer to *Oracle Database Security Guide* for complete information on auditing Oracle SQL*Loader Direct Path Load.

OLS

Use this clause to audit actions for Oracle Label Security. For *component_action*, specify the action to be audited. To view the valid actions for Oracle Label Security, query the NAME column of the AUDITABLE_SYSTEM_ACTIONS view where COMPONENT is Label Security. For example:

```
SELECT name FROM auditable_system_actions WHERE component = 'Label Security';
```

Refer to *Oracle Database Security Guide* for complete information on auditing Oracle Label Security.

XS

Use this clause to audit actions for Oracle Database Real Application Security. For *component_action*, specify the action to be audited. To view the valid actions for Oracle Database Real Application Security, query the NAME column of the AUDITABLE_SYSTEM_ACTIONS view where COMPONENT is XS. For example:

```
SELECT name FROM auditable_system_actions WHERE component = 'XS';
```

Refer to *Oracle Database Security Guide* for complete information on auditing Oracle Database Real Application Security.

DV

Use this clause to audit actions for Oracle Database Vault. For *component_action*, specify the action to be audited. To view the valid actions for Oracle Database Vault, query the NAME column of the AUDITABLE_SYSTEM_ACTIONS view where COMPONENT is Database Vault. For example:

```
SELECT name FROM auditable_system_actions WHERE component = 'Database Vault';
```

For *object_name*, specify the name of the Database Vault object to be audited.

Refer to *Oracle Database Security Guide* for complete information on auditing Oracle Database Vault.

SQL_FIREWALL

Use this clause to set the unified audit policy to track SQL firewall violations.

PROTOCOL

Use the PROTOCOL component to audit FTP and HTTP messages.

Example 1: Audit all HTTP Messages

```
CREATE AUDIT POLICY mypolicy ACTIONS COMPONENT = PROTOCOL HTTP
    AUDIT POLICY mypolicy
```

Example 2: Audit Failed FTP Messages

```
CREATE AUDIT POLICY mypolicy ACTIONS COMPONENT = PROTOCOL FTP
    AUDIT POLICY mypolicy WHENEVER NOT SUCCESSFUL
```

Example 3: Audit HTTP Messages that had 401 AUTH Replies

```
CREATE AUDIT POLICY mypolicy ACTIONS COMPONENT = PROTOCOL AUTHENTICATION
    AUDIT POLICY mypolicy
```

role_audit_clause

Use this clause to specify one or more roles to be audited. When you audit a role, Oracle Database audits all system privileges that are granted directly to the role. SQL statements that require the system privileges in order to succeed are audited. For *role*, specify either a user-defined (local or external) or predefined role. For a list of predefined roles, refer to *Oracle Database Security Guide*.

WHEN Clause

Use this clause to control when the unified audit policy is enforced.

audit_condition

Specify a condition that determines if the unified audit policy is enforced. If *audit_condition* evaluates to TRUE, then the policy is enforced. If FALSE, then the policy is not enforced.

The *audit_condition* can have a maximum length of 4000 characters. It can contain expressions, as well as the following functions and conditions:

- Numeric functions: BITAND, CEIL, FLOOR, POWER
- Character functions returning character values: CONCAT, LOWER, UPPER
- Character functions returning number values: INSTR, LENGTH
- Environment and identifier functions: SYS_CONTEXT, UID
- Comparison conditions: =, !=, <>, <, >, <=, >=
- Logical conditions: AND, OR
- Null conditions: IS [NOT] NULL
- [NOT] BETWEEN condition
- [NOT] IN condition

The *audit_condition* must be enclosed in single quotation marks. If the *audit_condition* contains a single quotation mark, then specify two single quotation marks instead. For example, to specify the following condition:

```
SYS_CONTEXT('USERENV', 'CLIENT_IDENTIFIER') = 'myclient'
```

Specify the following for '*audit_condition*':

```
'SYS_CONTEXT(''USERENV'', ''CLIENT_IDENTIFIER'') = ''myclient'''
```

The EVALUATE PER clauses evaluate the audit condition per instance per container. For example, if a condition is evaluated in one container, it will be evaluated again in any other container even if the instance is same.

EVALUATE PER STATEMENT

Specify this clause to evaluate the *audit_condition* for each auditable statement for each instance in the container. If the *audit_condition* evaluates to TRUE, then the unified audit policy is enforced for the statement. If FALSE, then the unified audit policy is not enforced for the statement.

EVALUATE PER SESSION

Specify this clause to evaluate the *audit_condition* once during the session. The *audit_condition* is evaluated for the first auditable statement that is executed during the session. If the *audit_condition* evaluates to TRUE, then the unified audit policy is enforced for all applicable statements for the rest of the session. If FALSE, then the unified audit policy is not enforced for all applicable statements for the rest of the session.

EVALUATE PER INSTANCE

Specify this clause to evaluate the *audit_condition* once during the lifetime of the instance. The *audit_condition* is evaluated for the first auditable statement that is executed during the instance lifetime. If the *audit_condition* evaluates to TRUE, then the unified audit policy is enforced for all applicable statements for the rest of the lifetime of the instance. If FALSE, then the unified audit policy is not enforced for all applicable statements for the rest of the lifetime of the instance.

ONLY TOPLEVEL

Specify the ONLY TOPLEVEL clause when you want to audit the SQL statements issued directly by a user.

SQL statements that are run from within a PL/SQL procedure are not considered top-level statements. You can audit top-level statements from all users, including user SYS.

For more see *Database Security Guide*.

CONTAINER Clause

Use the CONTAINER clause to specify the scope of the unified audit policy.

- Specify CONTAINER = ALL to create a **common unified audit policy**. This type of policy is available to all pluggable databases (PDBs) in the CDB. The current container must be the root. If you specify the ACTIONS *object_action* ON or ACTIONS ALL ON clause, then you must specify a common object or an application common object.
- Specify CONTAINER = CURRENT to create a **local unified audit policy** in the current container. The current container can be the root or a PDB.

If you omit this clause, then CONTAINER = CURRENT is the default.

> **Note:**
>
> You cannot alter the scope of a unified audit policy after it has been created.

Examples

Auditing System Privileges: Example

The following statement creates unified audit policy `table_pol`, which audits the system privileges `CREATE ANY TABLE` and `DROP ANY TABLE`:

```
CREATE AUDIT POLICY table_pol
  PRIVILEGES CREATE ANY TABLE, DROP ANY TABLE;
```

The following statement verifies that `table_pol` now appears in the `AUDIT_UNIFIED_POLICIES` view:

```
SELECT *
  FROM audit_unified_policies
  WHERE policy_name = 'TABLE_POL';
```

Auditing Actions on Objects: Examples

The following statement creates unified audit policy `dml_pol`, which audits `DELETE`, `INSERT`, and `UPDATE` actions on table `hr.employees`, and all auditable actions on table `hr.departments`:

```
CREATE AUDIT POLICY dml_pol
  ACTIONS DELETE on hr.employees,
          INSERT on hr.employees,
          UPDATE on hr.employees,
          ALL on hr.departments;
```

The following statement creates unified audit policy `read_dir_pol`, which audits `READ` actions on directory `bfile_dir` (created in "Creating a Directory: Examples"):

```
CREATE AUDIT POLICY read_dir_pol
  ACTIONS READ ON DIRECTORY bfile_dir;
```

Auditing System Actions: Examples

The following query displays the standard RDBMS system actions that can be audited for the database:

```
SELECT name FROM auditable_system_actions
  WHERE component = 'Standard'
  ORDER BY name;

NAME
----
ADMINISTER KEY MANAGEMENT
ALL
ALTER ASSEMBLY
ALTER AUDIT POLICY
ALTER CLUSTER
...
```

The following statement creates unified audit policy `security_pol`, which audits the system action `ADMINISTER KEY MANAGEMENT`:

```
CREATE AUDIT POLICY security_pol
  ACTIONS ADMINISTER KEY MANAGEMENT;
```

The following statement creates unified audit policy `dir_pol`, which audits all read, write, and execute operations on any directory:

```
CREATE AUDIT POLICY dir_pol
  ACTIONS READ DIRECTORY, WRITE DIRECTORY, EXECUTE DIRECTORY;
```

See *31.4.4.11 Example: Auditing All Actions in the Database* of the *Database Security Guide* for guidelines to audit database actions without generating a large volume of audit records.

Auditing Component Actions: Example

The following query displays the actions that can be audited for Oracle Data Pump:

```
SELECT name FROM auditable_system_actions
  WHERE component = 'Datapump';

NAME
----
EXPORT
IMPORT
ALL
```

The following statement creates unified audit policy `dp_actions_pol`, which audits `IMPORT` actions for Oracle Data Pump:

```
CREATE AUDIT POLICY dp_actions_pol
  ACTIONS COMPONENT = datapump IMPORT;
```

Auditing Roles: Example

The following statement creates unified audit policy `java_pol`, which audits the predefined roles `java_admin` and `java_deploy`:

```
CREATE AUDIT POLICY java_pol
  ROLES java_admin, java_deploy;
```

Auditing System Privileges, Actions, and Roles: Example

The following statement creates unified audit policy `hr_admin_pol`, which audits multiple system privileges, actions, and roles:

```
CREATE AUDIT POLICY hr_admin_pol
  PRIVILEGES CREATE ANY TABLE, DROP ANY TABLE
  ACTIONS DELETE on hr.employees,
          INSERT on hr.employees,
          UPDATE on hr.employees,
          ALL on hr.departments,
          LOCK TABLE
  ROLES audit_admin, audit_viewer;
```

Controlling When a Unified Audit Policy is Enforced: Examples

The following statement creates unified audit policy `order_updates_pol`, which audits `UPDATE` actions on table `oe.orders`. This policy is enforced only when the auditable statement is issued by an external user. The audit condition is checked once per session.

```
CREATE AUDIT POLICY order_updates_pol
  ACTIONS UPDATE ON oe.orders
  WHEN 'SYS_CONTEXT(''USERENV'', ''IDENTIFICATION_TYPE'') = ''EXTERNAL'''
  EVALUATE PER SESSION;
```

The following statement creates unified audit policy `emp_updates_pol`, which audits `DELETE`, `INSERT`, and `UPDATE` actions on table `hr.employees`. This policy is enforced only when the auditable statement is issued by a user who does not have a UID of 100, 105, or 107. The audit condition is checked for each auditable statement.

```
CREATE AUDIT POLICY emp_updates_pol
        ACTIONS DELETE on hr.employees,
          INSERT on hr.employees,
          UPDATE on hr.employees
        WHEN 'UID NOT IN (100, 105, 107)'
        EVALUATE PER STATEMENT;
```

Creating a Local Unified Audit Policy: Example

The following statement creates local unified audit policy `local_table_pol`, which audits the system privileges `CREATE ANY TABLE` and `DROP ANY TABLE` in the current container:

```
CREATE AUDIT POLICY local_table_pol
       PRIVILEGES CREATE ANY TABLE, DROP ANY TABLE
       CONTAINER = CURRENT;
```

Creating a Common Unified Audit Policy: Example

The following statement creates common unified audit policy `common_role1_pol`, which audits the common role `c##role1` (created in `CREATE ROLE` "Examples") across the entire CDB:

```
CREATE AUDIT POLICY c##common_role1_pol
      ROLES c##role1
      CONTAINER = ALL;
```

Creating an Audit Policy on Columns: Example

The audit policy `pol` generates an audit record when granting privileges on the column `job` in the `emp` table.

```
CREATE AUDIT POLICY pol ACTIONS GRANT(job) on scott.emp;
```

The audit policy `pol` generates an audit record when a new department number is inserted into the `dept` table.

```
CREATE AUDIT POLICY pol ACTIONS INSERT(deptno) on scott.dept;
```

CREATE CLUSTER

Purpose

Use the `CREATE CLUSTER` statement to create a cluster. A **cluster** is a schema object that contains data from one or more tables.

- An **indexed cluster** must contain more than one table, and all of the tables in the cluster have one or more columns in common. Oracle Database stores together all the rows from all the tables that share the same cluster key.
- In a **hash cluster**, which can contain one or more tables, Oracle Database stores together rows that have the same hash key value.

For information on existing clusters, query the `USER_CLUSTERS`, `ALL_CLUSTERS`, and `DBA_CLUSTERS` data dictionary views.

> **See Also:**
>
> - *Oracle Database Concepts* for general information on clusters
> - *Oracle Database SQL Tuning Guide* for suggestions on when to use clusters
> - *Oracle Database Reference* for information on the data dictionary views

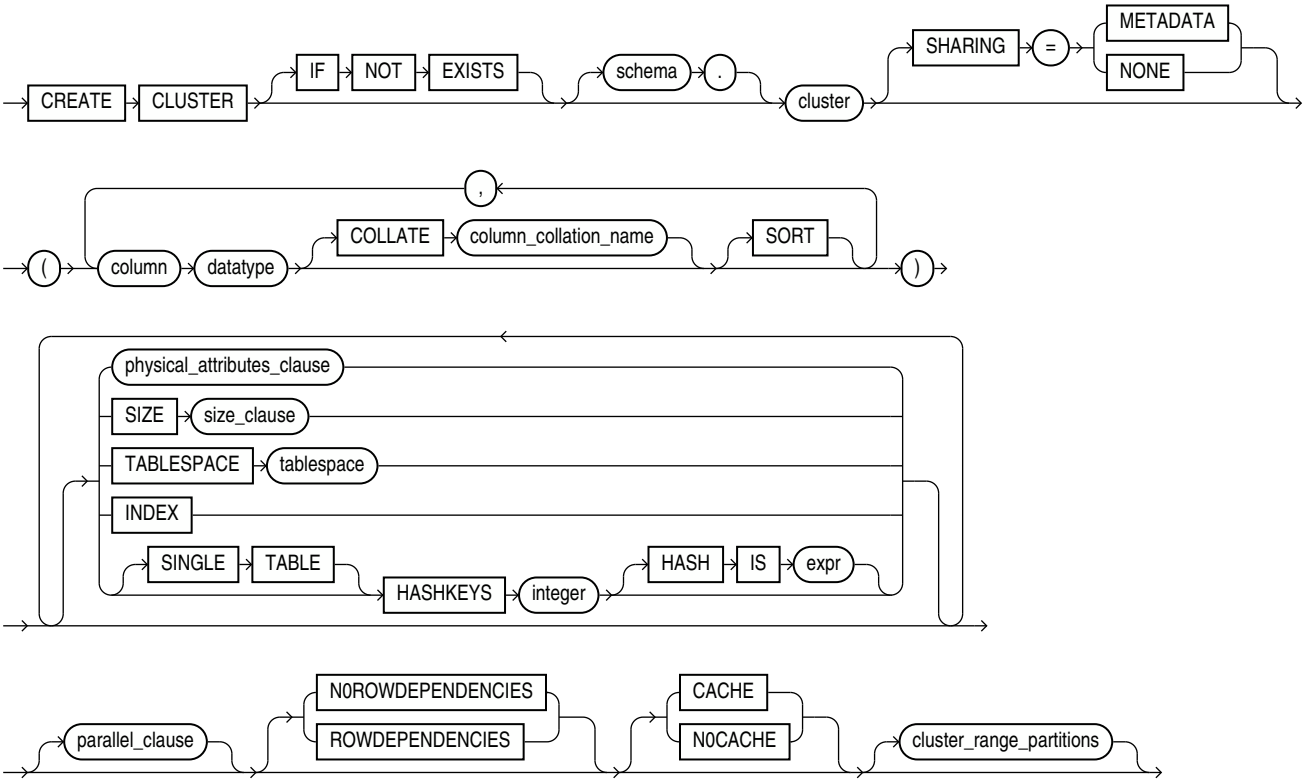
Prerequisites

To create a cluster in your own schema, you must have `CREATE CLUSTER` system privilege. To create a cluster in another user's schema, you must have `CREATE ANY CLUSTER` system privilege. Also, the owner of the schema to contain the cluster must have either space quota on the tablespace containing the cluster or the `UNLIMITED TABLESPACE` system privilege.

Oracle Database does not automatically create an index for a cluster when the cluster is initially created. Data manipulation language (DML) statements cannot be issued against cluster tables in an indexed cluster until you create a cluster index with a `CREATE INDEX` statement.

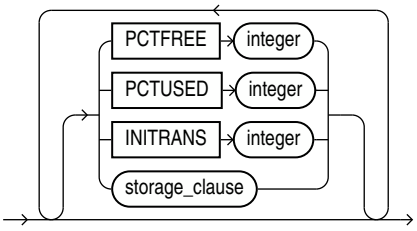
Syntax

***create_cluster*::=**



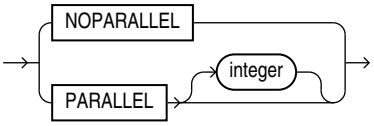
(*datatype*::=,*physical_attributes_clause*::=, *size_clause*::=, *cluster_range_partitions*::=)

***physical_attributes_clause*::=**

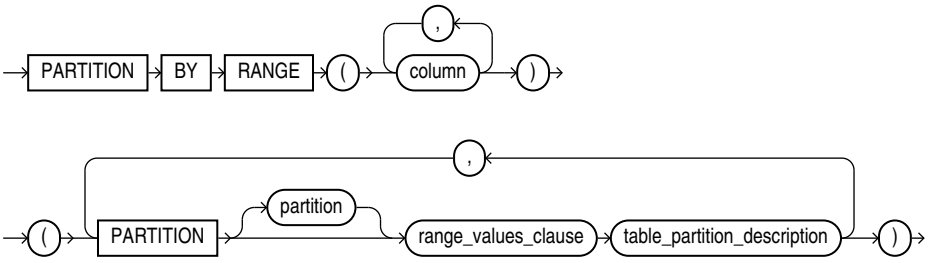


(*storage_clause*::=)

***parallel_clause*::=**



***cluster_range_partitions*::=**



(*range_values_clause*::=, *table_partition_description*::=)

Semantics

IF NOT EXISTS

Specifying `IF NOT EXISTS` has the following effects:

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

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

schema

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

cluster

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

After you create a cluster, you add tables to it. A cluster can contain a maximum of 32 tables. Object tables and tables containing LOB columns or columns of the `Any*` Oracle-supplied types cannot be part of a cluster. After you create a cluster and add tables to it, the cluster is transparent. You can access clustered tables with SQL statements just as you can access nonclustered tables.

> **See Also:**
>
> CREATE TABLE for information on adding tables to a cluster, "Creating a Cluster: Example", and "Adding Tables to a Cluster: Example"

SHARING

Use the sharing clause if you want to create the cluster 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.

column

Specify one or more names of columns in the cluster key. You can specify up to 16 cluster key columns. These columns must correspond in both data type and size to columns in each of the clustered tables, although they need not correspond in name.

You cannot specify integrity constraints as part of the definition of a cluster key column. Instead, you can associate integrity constraints with the tables that belong to the cluster.

> **See Also:**
>
> "Cluster Keys: Example"

datatype

Specify the data type of each cluster key column.

Restrictions on Cluster Data Types

Cluster data types are subject to the following restrictions:

- You cannot specify a cluster key column of data type `LONG`, `LONG RAW`, `REF`, nested table, varray, `BLOB`, `CLOB`, `BFILE`, the `Any*` Oracle-supplied types, or user-defined object type.
- You can specify a column of type `ROWID`, but Oracle Database does not guarantee that the values in such columns are valid rowids.

> **See Also:**
>
> "Data Types " for information on data types

COLLATE

Use this clause to specify the data-bound collation for character data type columns in the cluster key.

For *column_collation_name*, specify the collation as follows:

- When creating an indexed cluster or a sorted hash cluster, you can specify one of the following collations: `BINARY`, `USING_NLS_COMP`, `USING_NLS_SORT`, or `USING_NLS_SORT_CS`.
- When creating a hash cluster that is not sorted, you can specify any valid named collation or pseudo-collation.

If you omit this clause, then columns in the cluster key inherit the effective schema default collation of the schema containing the cluster. Refer to the DEFAULT_COLLATION clause of `ALTER SESSION` for more information on the effective schema default collation.

The collations of cluster key columns must match the collations of the corresponding columns in the tables created in the cluster.

You can specify the `COLLATE` 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 collation of a cluster key column, you must recreate the cluster.

SORT

The `SORT` keyword is valid only if you are creating a hash cluster. Table rows are hashed into buckets on cluster key columns without `SORT`, and then sorted in each bucket on the columns with this clause. This may improve response time during subsequent queries on the clustered data.

All columns without the `SORT` clause must come before all columns with the `SORT` clause in the `CREATE CLUSTER` statement.

Restriction on Sorted Hash Clusters

Row dependency is not supported for sorted hash clusters.

> **See Also:**
>
> - See "HASHKEYS Clause" for information on creating a hash cluster.
> - *Managing Hash Clusters* for more information.

physical_attributes_clause

The *physical_attributes_clause* lets you specify the storage characteristics of the cluster. Each table in the cluster uses these storage characteristics as well. If you do not specify values for these parameters, then Oracle Database uses the following defaults:

- PCTFREE: 10
- PCTUSED: 40
- INITRANS: 2 or the default value of the tablespace to contain the cluster, whichever is greater

> **See Also:**
>
> *physical_attributes_clause* and storage_clause for a complete description of these clauses

SIZE

Specify the amount of space in bytes reserved to store all rows with the same cluster key value or the same hash value. This space determines the maximum number of cluster or hash values stored in a data block. If SIZE is not a divisor of the data block size, then Oracle Database uses the next largest divisor. If SIZE is larger than the data block size, then the database uses the operating system block size, reserving at least one data block for each cluster or hash value.

The database also considers the length of the cluster key when determining how much space to reserve for the rows having a cluster key value. Larger cluster keys require larger sizes. To see the actual size, query the KEY_SIZE column of the USER_CLUSTERS data dictionary view. (This value does not apply to hash clusters, because hash values are not actually stored in the cluster.)

If you omit this parameter, then the database reserves one data block for each cluster key value or hash value.

TABLESPACE

Specify the tablespace in which the cluster is to be created.

INDEX Clause

Specify INDEX to create an **indexed cluster**. In an indexed cluster, Oracle Database stores together rows having the same cluster key value. Each distinct cluster key value is stored only once in each data block, regardless of the number of tables and rows in which it occurs. If you specify neither INDEX nor HASHKEYS, then Oracle Database creates an indexed cluster by default.

After you create an indexed cluster, you must create an index on the cluster key before you can issue any data manipulation language (DML) statements against a table in the cluster. This index is called the **cluster index**.

You cannot create a cluster index for a hash cluster, and you need not create an index on a hash cluster key.

> **See Also:**
>
> CREATE INDEX for information on creating a cluster index and *Oracle Database Concepts* for general information in indexed clusters

HASHKEYS Clause

Specify the `HASHKEYS` clause to create a **hash cluster** and specify the number of hash values for the hash cluster. In a hash cluster, Oracle Database stores together rows that have the same hash key value. The hash value for a row is the value returned by the hash function of the cluster.

Oracle Database rounds up the `HASHKEYS` value to the nearest prime number to obtain the actual number of hash values. The minimum value for this parameter is 2. If you omit both the `INDEX` clause and the `HASHKEYS` parameter, then the database creates an indexed cluster by default.

When you create a hash cluster, the database immediately allocates space for the cluster based on the values of the `SIZE` and `HASHKEYS` parameters.

> **See Also:**
>
> *Oracle Database Concepts* for more information on how Oracle Database allocates space for clusters and "Hash Clusters: Examples"

SINGLE TABLE

`SINGLE TABLE` indicates that the cluster is a type of hash cluster containing only one table. This clause can provide faster access to rows in the table.

Restriction on Single-table Clusters

Only one table can be present in the cluster at a time. However, you can drop the table and create a different table in the same cluster.

> **See Also:**
>
> "Single-Table Hash Clusters: Example"

HASH IS *expr*

Specify an expression to be used as the hash function for the hash cluster. The expression:

- Must evaluate to a positive value
- Must contain at least one column, with referenced columns of any data type as long as the entire expression evaluates to a number of scale 0. For example: *`number_column`* * `LENGTH`(*`varchar2_column`*)
- Cannot reference user-defined PL/SQL functions
- Cannot reference the pseudocolumns `LEVEL` or `ROWNUM`

- Cannot reference the user-related functions `USERENV`, `UID`, or `USER` or the datetime functions `CURRENT_DATE`, `CURRENT_TIMESTAMP`, `DBTIMEZONE`, `EXTRACT` (datetime), `FROM_TZ`, `LOCALTIMESTAMP`, `NUMTODSINTERVAL`, `NUMTOYMINTERVAL`, `SESSIONTIMEZONE`, `SYSDATE`, `SYSTIMESTAMP`, `TO_DSINTERVAL`, `TO_TIMESTAMP`, `TO_DATE`, `TO_TIMESTAMP_TZ`, `TO_YMINTERVAL`, and `TZ_OFFSET`.
- Cannot evaluate to a constant
- Cannot be a scalar subquery expression
- Cannot contain columns qualified with a schema or object name (other than the cluster name)

If you omit the `HASH IS` clause, then Oracle Database uses an internal hash function for the hash cluster.

For information on existing hash functions, query the `USER_`, `ALL_`, and `DBA_CLUSTER_HASH_EXPRESSIONS` data dictionary tables.

The cluster key of a hash column can have one or more columns of any data type. Hash clusters with composite cluster keys or cluster keys made up of noninteger columns must use the internal hash function.

> **See Also:**
>
> *Oracle Database Reference* for information on the data dictionary views

parallel_clause

The *parallel_clause* lets you parallelize the creation of the cluster.

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

NOROWDEPENDENCIES | ROWDEPENDENCIES

This clause has the same behavior for a cluster that it has for a table. Refer to "NOROWDEPENDENCIES | ROWDEPENDENCIES" in `CREATE TABLE` for information.

CACHE | NOCACHE

CACHE

Specify `CACHE` if you want the blocks retrieved for this cluster to be 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 clause is useful for small lookup tables.

NOCACHE

Specify `NOCACHE` if you want the blocks retrieved for this cluster to be placed at the least recently used end of the LRU list in the buffer cache when a full table scan is performed. This is the default behavior.

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

cluster_range_partitions

Specify the *cluster_range_partitions* clause to create a range-partitioned hash cluster. If you specify this clause, then you must also specify the HASHKEYS clause.

Use the *cluster_range_partitions* clause to partition the cluster on ranges of values from the column list. When you add a table to a range-partitioned hash cluster, it is automatically partitioned on the same columns, with the same number of partitions, and on the same partition bounds as the cluster. Oracle Database assigns system-generated names to the table partitions.

Each partitioning key column with a character data type must have one of the following declared collations: BINARY, USING_NLS_COMP, USING_NLS_SORT, or USING_NLS_SORT_CS.

The *cluster_range_partitions* clause has the same semantics as the *range_partitions* clause of CREATE TABLE, except that here you cannot specify the INTERVAL clause. For complete information, refer to *range_partitions* in the documentation on CREATE TABLE.

> **See Also:**
>
> "Range-Partitioned Hash Clusters: Example"

Examples

Creating a Cluster: Example

The following statement creates a cluster named personnel with the cluster key column department, a cluster size of 512 bytes, and storage parameter values:

```
CREATE CLUSTER personnel
   (department NUMBER(4))
SIZE 512
STORAGE (initial 100K next 50K);
```

Cluster Keys: Example

The following statement creates the cluster index on the cluster key of personnel:

```
CREATE INDEX idx_personnel ON CLUSTER personnel;
```

After creating the cluster index, you can add tables to the index and perform DML operations on those tables.

Adding Tables to a Cluster: Example

The following statements create some departmental tables from the sample hr.employees table and add them to the personnel cluster created in the earlier example:

```
CREATE TABLE dept_10
   CLUSTER personnel (department_id)
   AS SELECT * FROM employees WHERE department_id = 10;

CREATE TABLE dept_20
   CLUSTER personnel (department_id)
   AS SELECT * FROM employees WHERE department_id = 20;
```

Hash Clusters: Examples

The following statement creates a hash cluster named `language` with the cluster key column `cust_language`, a maximum of 10 hash key values, each of which is allocated 512 bytes, and storage parameter values:

```
CREATE CLUSTER language (cust_language VARCHAR2(3))
   SIZE 512 HASHKEYS 10
   STORAGE (INITIAL 100k next 50k);
```

Because the preceding statement omits the `HASH IS` clause, Oracle Database uses the internal hash function for the cluster.

The following statement creates a hash cluster named `address` with the cluster key made up of the columns `postal_code` and `country_id`, and uses a SQL expression containing these columns for the hash function:

```
CREATE CLUSTER address
   (postal_code NUMBER, country_id CHAR(2))
   HASHKEYS 20
   HASH IS MOD(postal_code + country_id, 101);
```

Single-Table Hash Clusters: Example

The following statement creates a single-table hash cluster named `cust_orders` with the cluster key `customer_id` and a maximum of 100 hash key values, each of which is allocated 512 bytes:

```
CREATE CLUSTER cust_orders (customer_id NUMBER(6))
   SIZE 512 SINGLE TABLE HASHKEYS 100;
```

Range-Partitioned Hash Clusters: Example

The following statement creates a range-partitioned hash cluster named `sales` with five range partitions based on the amount sold. The cluster key is made up of the columns `amount_sold` and `prod_id`. The cluster uses the hash function `(amount_sold * 10 + prod_id)` and has a maximum of 100000 hash key values, each of which is allocated 300 bytes.

```
CREATE CLUSTER sales (amount_sold NUMBER, prod_id NUMBER)
  HASHKEYS 100000
  HASH IS (amount_sold * 10 + prod_id)
  SIZE 300
  TABLESPACE example
  PARTITION BY RANGE (amount_sold)
    (PARTITION p1 VALUES LESS THAN (2001),
     PARTITION p2 VALUES LESS THAN (4001),
     PARTITION p3 VALUES LESS THAN (6001),
     PARTITION p4 VALUES LESS THAN (8001),
     PARTITION p5 VALUES LESS THAN (MAXVALUE));
```

Create Cluster Tables: Example

The following statement creates a cluster named `emp_dept` with the default key size (600):

```
CREATE CLUSTER emp_dept (deptno NUMBER(3))
   SIZE 600
   TABLESPACE USERS
   STORAGE (INITIAL 200K
      NEXT 300K
      MINEXTENTS 2
      PCTINCREASE 33);
```

The following statement creates a cluster table named `dept` under `emp_dept` cluster:

```
CREATE TABLE dept (
   deptno NUMBER(3) PRIMARY KEY)
   CLUSTER emp_dept (deptno);
```

The following statement creates another cluster table named `empl` under `emp_dept` cluster:

```
CREATE TABLE empl (
   emplno NUMBER(5) PRIMARY KEY,
   emplname VARCHAR2(15) NOT NULL,
   deptno NUMBER(3) REFERENCES dept)
   CLUSTER emp_dept (deptno);
```

The following statement creates an index for the `emp_dept` cluster:

```
CREATE INDEX emp_dept_index
   ON CLUSTER emp_dept
   TABLESPACE USERS
   STORAGE (INITIAL 50K
      NEXT 50K
      MINEXTENTS 2
      MAXEXTENTS 10
      PCTINCREASE 33);
```

The following statement queries `USER_CLUSTERS` to display the cluster metadata:

```
SELECT CLUSTER_NAME, TABLESPACE_NAME, CLUSTER_TYPE, PCT_INCREASE,
MIN_EXTENTS, MAX_EXTENTS FROM USER_CLUSTERS;

CLUSTER_NAME    TABLESPACE CLUST PCT_INCREASE MIN_EXTENTS MAX_EXTENTS
--------------- ---------- ----- ------------ ----------- -----------
EMP_DEPT    USERS       INDEX            1  2147483645
```

The following statement queries `USER_CLU_COLUMNS` to display the cluster metadata:

```
SELECT * FROM USER_CLU_COLUMNS;

CLUSTER_NAME    CLU_COLUMN_NAME      TABLE_NAME TAB_COLUMN_NAME
--------------- -------------------- ---------- --------------------
EMP_DEPT    DEPTNO             DEPT    DEPTNO
EMP_DEPT    DEPTNO             EMPL    DEPTNO
```

The following statement queries `USER_INDEXES` to display the index attributes of the `emp_dept` cluster:

```
SELECT INDEX_NAME, INDEX_TYPE, PCT_INCREASE, MIN_EXTENTS, MAX_EXTENTS FROM
USER_INDEXES WHERE TABLE_NAME='EMP_DEPT';

INDEX_NAME    INDEX_TYPE    PCT_INCREASE MIN_EXTENTS MAX_EXTENTS
--------------- --------------- ------------ ----------- -----------
EMP_DEPT_INDEX    CLUSTER                   1  2147483645
```

CREATE CONTEXT

Purpose

Use the `CREATE CONTEXT` statement to:

- Create a namespace for a **context** (a set of application-defined attributes that validates and secures an application)
- Associate the namespace with the externally created package that sets the context

You can use the `DBMS_SESSION.SET_CONTEXT` procedure in your designated package to set or reset the attributes of the context.

> **See Also:**
>
> - *Oracle Database Security Guide* for a discussion of contexts
> - *Oracle Database PL/SQL Packages and Types Reference* for information on the `DBMS_SESSION.SET_CONTEXT` procedure

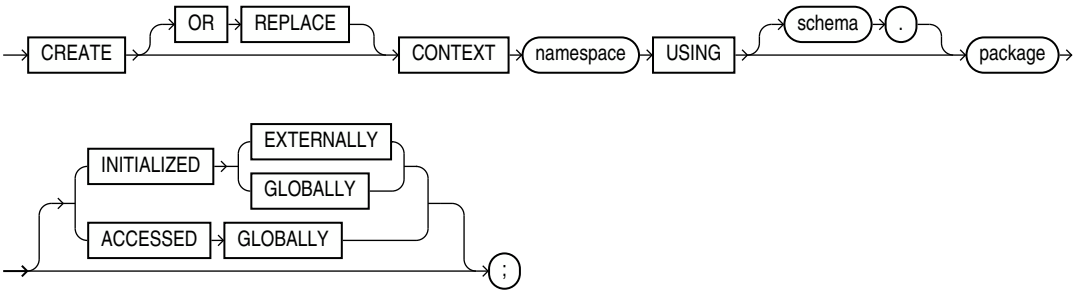
Prerequisites

To create a context namespace, you must have `CREATE ANY CONTEXT` system privilege.

Note that you cannot use a synonym for a package name in the `CREATE CONTEXT` command.

Syntax

***create_context*::=**



Semantics

OR REPLACE

Specify `OR REPLACE` to redefine an existing context namespace using a different package.

namespace

Specify the name of the context namespace to create or modify. The name must satisfy the requirements listed in "Database Object Naming Rules ". Context namespaces are always stored in the schema `SYS`.

> **See Also:**
>
> "Database Object Naming Rules " for guidelines on naming a context namespace

schema

Specify the schema owning *package*. If you omit *schema*, then Oracle Database uses the current schema.

package

Specify the PL/SQL package that sets or resets the context attributes under the namespace for a user session.

To provide some design flexibility, Oracle Database does not verify the existence of the schema or the validity of the package at the time you create the context.

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.

INITIALIZED Clause

The `INITIALIZED` clause lets you specify an entity other than Oracle Database that can initialize the context namespace.

EXTERNALLY

`EXTERNALLY` indicates that the namespace can be initialized using an OCI interface when establishing a session.

> **See Also:**
>
> *Oracle Call Interface Programmer's Guide* for information on using OCI to establish a session

GLOBALLY

`GLOBALLY` indicates that the namespace can be initialized by the LDAP directory when a global user connects to the database.

After the session is established, only the designated PL/SQL package can issue commands to write to any attributes inside the namespace.

> **See Also:**
>
> *Oracle Database Security Guide* for information on establishing globally initialized contexts

ACCESSED GLOBALLY

This clause indicates that any application context set in *namespace* is accessible throughout the entire instance. This setting lets multiple sessions share application attributes.

Examples

Creating an Application Context: Example

This example uses a PL/SQL package `emp_mgmt`, which validates and secures a human resources application. See *Oracle Database PL/SQL Language Reference* for the example that creates that package. The following statement creates the context namespace `hr_context` and associates it with the package `emp_mgmt`:

```
CREATE CONTEXT hr_context USING emp_mgmt;
```

You can control data access based on this context using the `SYS_CONTEXT` function. For example, the `emp_mgmt` package has defined an attribute `department_id` as a particular department identifier. You can secure the base table `employees` by creating a view that restricts access based on the value of `department_id`, as follows:

```
CREATE VIEW hr_org_secure_view AS
   SELECT * FROM employees
   WHERE department_id = SYS_CONTEXT('hr_context', 'department_id');
```

> **See Also:**
>
> SYS_CONTEXT and *Oracle Database Security Guide* for more information on using application contexts to retrieve user information

CREATE CONTROLFILE

> **Note:**
>
> Oracle recommends that you perform a full backup of all files in the database before using this statement. For more information, see *Oracle Database Backup and Recovery User's Guide*.

Purpose

The `CREATE CONTROLFILE` statement should be used in only a few cases. Use this statement to re-create a control file if all control files being used by the database are lost **and** no backup control file exists. You can also use this statement to change the maximum number of redo log

file groups, redo log file members, archived redo log files, data files, or instances that can concurrently have the database mounted and open.

To change the name of the database, Oracle recommends that you use the `DBNEWID` utility rather than the `CREATE CONTROLFILE` statement. `DBNEWID` is preferable because no `OPEN RESETLOGS` operation is required after changing the database name.

> **See Also:**
>
> - *Oracle Database Utilities* for more information about the `DBNEWID` utility
> - `ALTER DATABASE` "BACKUP CONTROLFILE Clause" for information creating a script based on an existing database control file

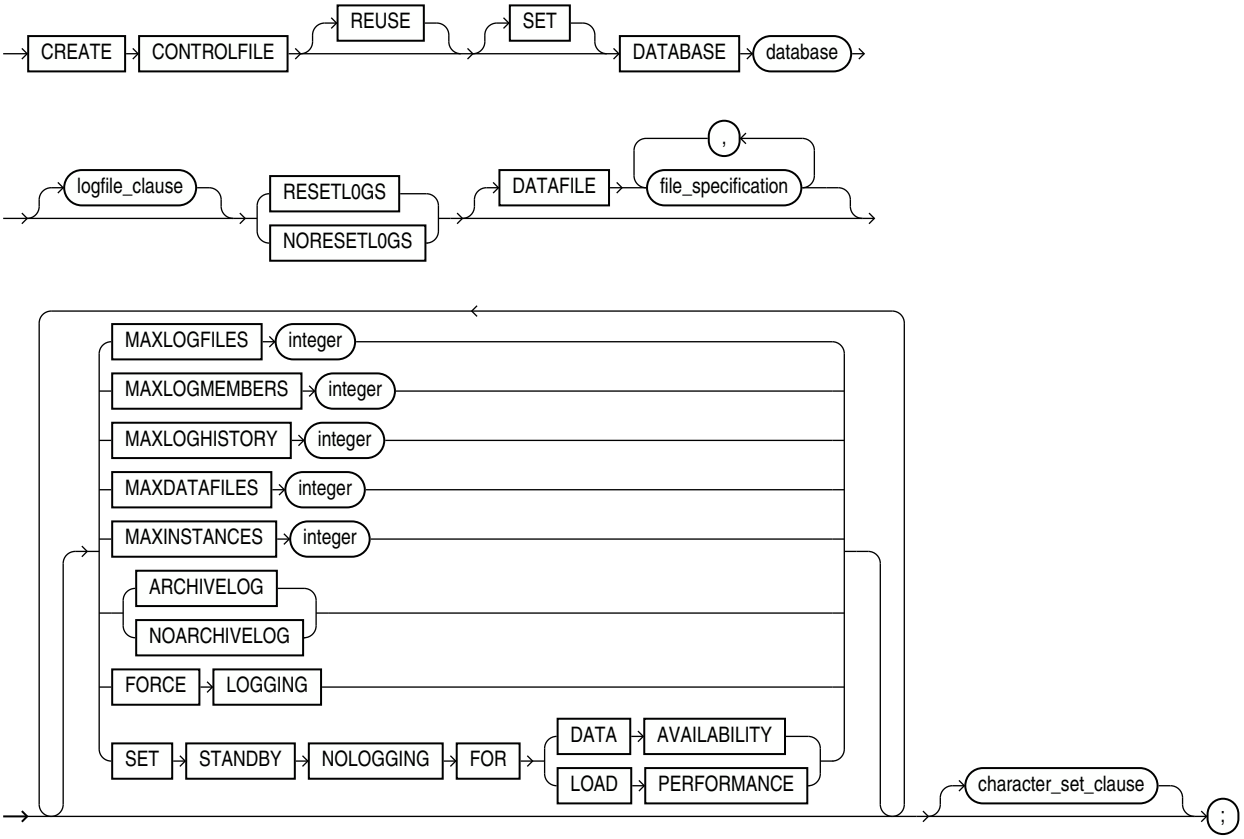
Prerequisites

To create a control file, you must have the `SYSDBA` or `SYSBACKUP` system privilege.

The database must not be mounted by any instance. After successfully creating the control file, Oracle mounts the database in the mode specified by the `CLUSTER_DATABASE` parameter. The DBA must then perform media recovery before opening the database. If you are using the database with Oracle Real Application Clusters (Oracle RAC), then you must then shut down and remount the database in `SHARED` mode (by setting the value of the `CLUSTER_DATABASE` initialization parameter to `TRUE`) before other instances can start up.

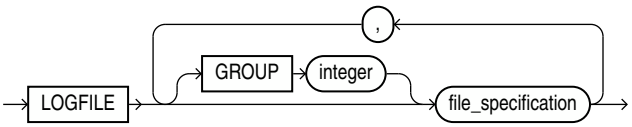
Syntax

***create_controlfile*::=**



(*storage_clause*::=)

***logfile_clause*::=**



(*file_specification*::=)

***character_set_clause*::=**

Semantics

When you issue a `CREATE CONTROLFILE` statement, Oracle Database creates a new control file based on the information you specify in the statement. The control file resides in the location

specified in the CONTROL_FILES initialization parameter. If that parameter does not have a value, then the database creates an Oracle-managed control file in the default control file destination, which is one of the following (in order of precedence):

1. One or more control files as specified in the DB_CREATE_ONLINE_LOG_DEST_*n* initialization parameter. The file in the first directory is the primary control file. When DB_CREATE_ONLINE_LOG_DEST_*n* is specified, the database does not create a control file in DB_CREATE_FILE_DEST or in DB_RECOVERY_FILE_DEST (the fast recovery area).
2. If no value is specified for DB_CREATE_ONLINE_LOG_DEST_*n*, but values are set for both the DB_CREATE_FILE_DEST and DB_RECOVERY_FILE_DEST, then the database creates one control file in each location. The location specified in DB_CREATE_FILE_DEST is the primary control file.
3. If a value is specified only for DB_CREATE_FILE_DEST, then the database creates one control file in that location.
4. If a value is specified only for DB_RECOVERY_FILE_DEST, then the database creates one control file in that location.

If no values are set for any of these parameters, then the database creates a control file in the default location for the operating system on which the database is running. This control file is not an Oracle Managed File.

If you omit any clauses, then Oracle Database uses the default values rather than the values for the previous control file. After successfully creating the control file, Oracle Database mounts the database in the mode specified by the initialization parameter CLUSTER_DATABASE. If that parameter is not set, then the default value is FALSE, and the database is mounted in EXCLUSIVE mode. Oracle recommends that you then shut down the instance and take a full backup of all files in the database.

> **See Also:**
>
> *Oracle Database Backup and Recovery User's Guide*

REUSE

Specify REUSE to indicate that existing control files identified by the initialization parameter CONTROL_FILES can be reused, overwriting any information they may currently contain. If you omit this clause and any of these control files already exists, then Oracle Database returns an error.

DATABASE Clause

Specify the name of the database. The value of this parameter must be the existing database name established by the previous CREATE DATABASE statement or CREATE CONTROLFILE statement.

SET DATABASE Clause

Use SET DATABASE to change the name of the database. The name of a database can be as long as eight bytes.

When you specify this clause, you must also specify RESETLOGS. If you want to rename the database and retain your existing log files, then after issuing this CREATE CONTROLFILE

statement you must complete a full database recovery using an `ALTER DATABASE RECOVER USING BACKUP CONTROLFILE` statement.

logfile_clause

Use the *logfile_clause* to specify the redo log files for your database. You must list all members of all redo log file groups.

Use the *redo_log_file_spec* form of *file_specification* (see *file_specification*) to list regular redo log files in an operating system file system or to list Oracle ASM disk group redo log files. When using a form of *ASM_filename*, you cannot specify the *autoextend_clause* of the *redo_log_file_spec*.

If you specify `RESETLOGS` in this clause, then you must use one of the file creation forms of *ASM_filename*. If you specify `NORESETLOGS`, then you must specify one of the reference forms of *ASM_filename*.

> **See Also:**
>
> *ASM_filename* for information on the different forms of syntax and *Oracle Automatic Storage Management Administrator's Guide* for general information about using Oracle ASM

GROUP *integer*

Specify the logfile group number. If you specify `GROUP` values, then Oracle Database verifies these values with the `GROUP` values when the database was last open.

If you omit this clause, then the database creates logfiles using system default values. In addition, if either the `DB_CREATE_ONLINE_LOG_DEST_n` or `DB_CREATE_FILE_DEST` initialization parameter has been set, and if you have specified `RESETLOGS`, then the database creates two logs in the default logfile destination specified in the `DB_CREATE_ONLINE_LOG_DEST_n` parameter, and if it is not set, then in the `DB_CREATE_FILE_DEST` parameter.

> **See Also:**
>
> *file_specification* for a full description of this clause

RESETLOGS

Specify `RESETLOGS` if you want Oracle Database to ignore the contents of the files listed in the `LOGFILE` clause. These files do not have to exist. You must specify this clause if you have specified the `SET DATABASE` clause.

Each *redo_log_file_spec* in the `LOGFILE` clause must specify the `SIZE` parameter. The database assigns all online redo log file groups to thread 1 and enables this thread for public use by any instance. After using this clause, you must open the database using the `RESETLOGS` clause of the `ALTER DATABASE` statement.

NORESETLOGS

Specify `NORESETLOGS` if you want Oracle Database to use all files in the `LOGFILE` clause as they were when the database was last open. These files must exist and must be the current online redo log files rather than restored backups. The database reassigns the redo log file groups to the threads to which they were previously assigned and reenables the threads as they were previously enabled.

You cannot specify `NORESETLOGS` if you have specified the `SET DATABASE` clause to change the name of the database. Refer to "SET DATABASE Clause" for more information.

DATAFILE Clause

Specify the data files of the database. You must list all data files. These files must all exist, although they may be restored backups that require media recovery.

Do not include in the `DATAFILE` clause any data files in read-only tablespaces. You can add these types of files to the database later. Also, do not include in this clause any temporary data files (temp files).

Use the *`datafile_tempfile_spec`* form of *`file_specification`* (see *file_specification*) to list regular data files and temp files in an operating system file system or to list Oracle ASM disk group files. When using a form of *`ASM_filename`*, you must use one of the reference forms of *`ASM_filename`*. Refer to *ASM_filename* for information on the different forms of syntax.

> **See Also:**
>
> *Oracle Automatic Storage Management Administrator's Guide* for general information about using Oracle ASM

Restriction on DATAFILE

You cannot specify the *`autoextend_clause`* of *`file_specification`* in this `DATAFILE` clause.

MAXLOGFILES Clause

Specify the maximum number of online redo log file groups that can ever be created for the database. Oracle Database uses this value to determine how much space to allocate in the control file for the names of redo log files. The default and maximum values depend on your operating system. The value that you specify should not be less than the greatest `GROUP` value for any redo log file group.

MAXLOGMEMBERS Clause

Specify the maximum number of members, or identical copies, for a redo log file group. Oracle Database uses this value to determine how much space to allocate in the control file for the names of redo log files. The minimum value is 1. The maximum and default values depend on your operating system.

MAXLOGHISTORY Clause

This parameter is useful only if you are using Oracle Database in `ARCHIVELOG` mode. Specify your current estimate of the maximum number of archived redo log file groups needed for automatic media recovery of the database. The database uses this value to determine how much space to allocate in the control file for the names of archived redo log files.

The minimum value is 0. The default value is a multiple of the MAXINSTANCES value and depends on your operating system. The maximum value is limited only by the maximum size of the control file. The database will continue to add additional space to the appropriate section of the control file as needed, so that you do not need to re-create the control file if your your original configuration is no longer adequate. As a result, the actual value of this parameter can eventually exceed the value you specify.

MAXDATAFILES Clause

Specify the initial sizing of the data files section of the control file at CREATE DATABASE or CREATE CONTROLFILE time. An attempt to add a file whose number is greater than MAXDATAFILES, but less than or equal to DB_FILES, causes the control file to expand automatically so that the data files section can accommodate more files.

The number of data files accessible to your instance is also limited by the initialization parameter DB_FILES.

MAXINSTANCES Clause

Specify the maximum number of instances that can simultaneously have the database mounted and open. This value takes precedence over the value of the initialization parameter INSTANCES. The minimum value is 1. The maximum and default values depend on your operating system.

ARCHIVELOG | NOARCHIVELOG

Specify ARCHIVELOG to archive the contents of redo log files before reusing them. This clause prepares for the possibility of media recovery as well as instance or system failure recovery.

If you omit both the ARCHIVELOG clause and NOARCHIVELOG clause, then Oracle Database chooses NOARCHIVELOG mode by default. After creating the control file, you can change between ARCHIVELOG mode and NOARCHIVELOG mode with the ALTER DATABASE statement.

FORCE LOGGING

Use this clause to put the database into FORCE LOGGING mode after control file creation. When the database is in this mode, Oracle Database logs all changes in the database except changes to temporary tablespaces and temporary segments. This setting takes precedence over and is independent of any NOLOGGING or FORCE LOGGING settings you specify for individual tablespaces and any NOLOGGING settings you specify for individual database objects. If you omit this clause, then the database will not be in FORCE LOGGING mode after the control file is created.

> **Note:**
>
> FORCE LOGGING mode can have performance effects. Refer to *Oracle Database Administrator's Guide* for information on when to use this setting.

SET STANDBY NOLOGGING FOR DATA AVAILABILITY | LOAD PERFORMANCE

SET STANDBY NOLOGGING

The SET STANDBY NOLOGGING disables logging on the standby. You can specify it in two modes:

- **SET STANDBY NOLOGGING FOR DATA AVAILABILITY** guarantees full data replication to the standby database. The primary and standby databases are synchronized during the load. In cases of network congestion the primary database will throttle its load.
- **SET STANDBY NOLOGGING FOR LOAD PERFORMANCE** to maintain speed of primary database load and synchronize with the standby later.

Restrictions On SET STANDBY NOLOGGING

The `SET STANDBY NOLOGGING` clause cannot be used at the same time as `FORCE LOGGING`.

character_set_clause

If you specify a character set, then Oracle Database reconstructs character set information in the control file. If media recovery of the database is subsequently required, then this information will be available before the database is open, so that tablespace names can be correctly interpreted during recovery. This clause is required only if you are using a character set other than the default, which depends on your operating system. Oracle Database prints the current database character set to the alert log in `$ORACLE_HOME/log` during startup.

If you are re-creating your control file and you are using Recovery Manager for tablespace recovery, and if you specify a different character set from the one stored in the data dictionary, then tablespace recovery will not succeed. However, at database open, the control file character set will be updated with the correct character set from the data dictionary.

You cannot modify the character set of the database with this clause.

> **See Also:**
>
> *Oracle Database Backup and Recovery User's Guide* for more information on tablespace recovery

Examples

Creating a Controlfile: Example

This statement re-creates a control file. In this statement, database `demo` was created with the WE8DEC character set. The example uses the word *`path`* where you would normally insert the path on your system to the appropriate Oracle Database directories.

```
STARTUP NOMOUNT

CREATE CONTROLFILE REUSE DATABASE "demo" NORESETLOGS NOARCHIVELOG
    MAXLOGFILES 32
    MAXLOGMEMBERS 2
    MAXDATAFILES 32
    MAXINSTANCES 1
    MAXLOGHISTORY 449
LOGFILE
  GROUP 1 '/path/oracle/dbs/t_log1.f'  SIZE 500K,
  GROUP 2 '/path/oracle/dbs/t_log2.f'  SIZE 500K
# STANDBY LOGFILE
DATAFILE
  '/path/oracle/dbs/t_db1.f',
  '/path/oracle/dbs/dbu19i.dbf',
  '/path/oracle/dbs/tbs_11.f',
  '/path/oracle/dbs/smundo.dbf',
  '/path/oracle/dbs/demo.dbf'
```

```
CHARACTER SET WE8DEC
;
```

CREATE DATABASE

> **Note:**
>
> This statement prepares a database for initial use and erases any data currently in the specified files. Use this statement only when you understand its ramifications.

> **Note:**
>
> In this release of Oracle Database and in subsequent releases, several enhancements are being made to ensure the security of default database user accounts. You can find a security checklist for this release in *Oracle Database Security Guide*. Oracle recommends that you read this checklist and configure your database accordingly.

Purpose

Use the `CREATE DATABASE` statement to create a database, making it available for general use.

This statement erases all data in any specified data files that already exist in order to prepare them for initial database use. If you use the statement on an existing database, then all data in the data files is lost.

After creating the database, this statement mounts it in either exclusive or parallel mode, depending on the value of the `CLUSTER_DATABASE` initialization parameter and opens it, making it available for normal use. You can then create tablespaces for the database.

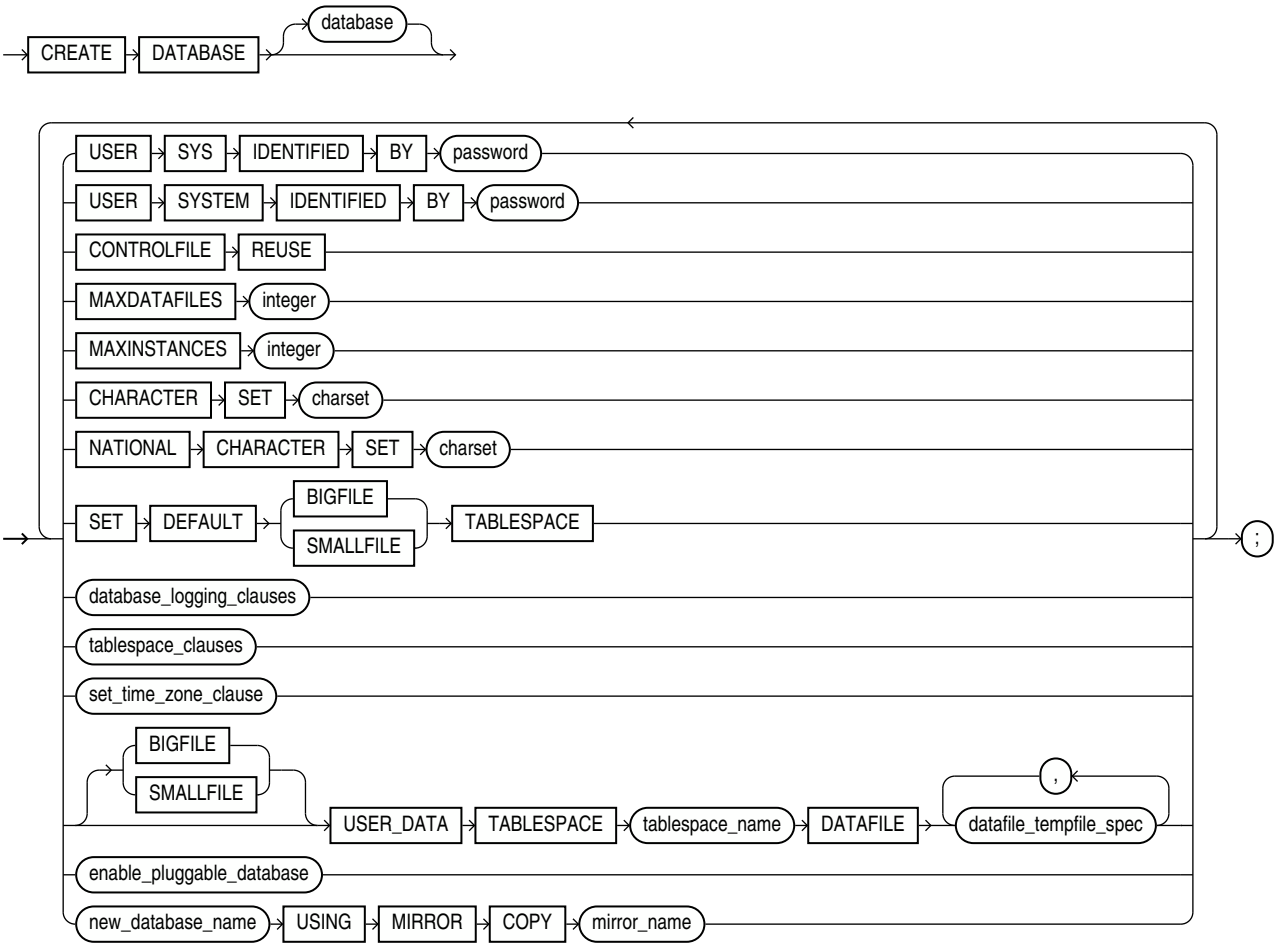
> **See Also:**
>
> - ALTER DATABASE for information on modifying a database
> - *Oracle Database Java Developer's Guide* for information on creating an Oracle Java virtual machine
> - CREATE TABLESPACE for information on creating tablespaces

Prerequisites

To create a database, you must have the `SYSDBA` system privilege. An initialization parameter file with the name of the database to be created must be available, and you must be in `STARTUP NOMOUNT` mode.

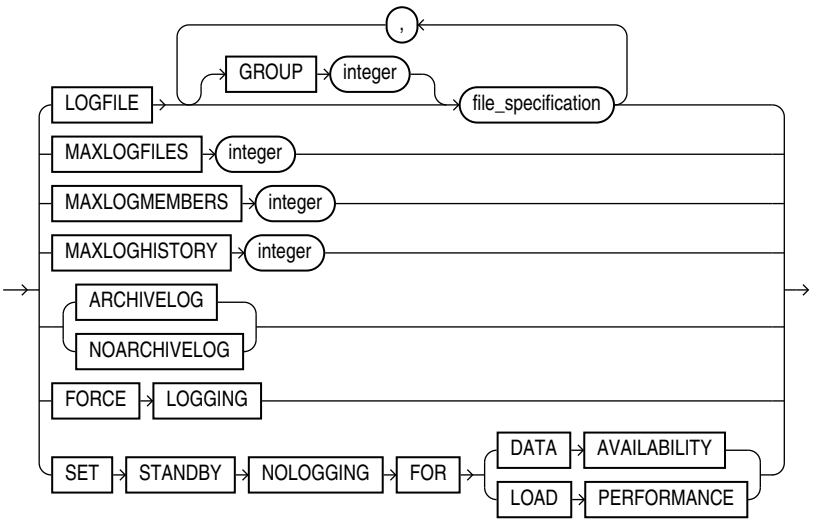
Syntax

***create_database*::=**



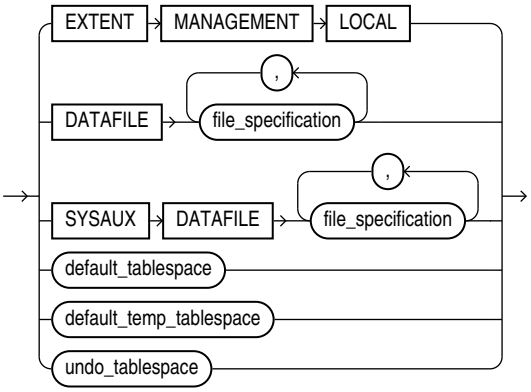
(*database_logging_clauses*::=, *tablespace_clauses*::=, *set_time_zone_clause*::=, *datafile_tempfile_spec*::=, *enable_pluggable_database*::=)

***database_logging_clauses*::=**



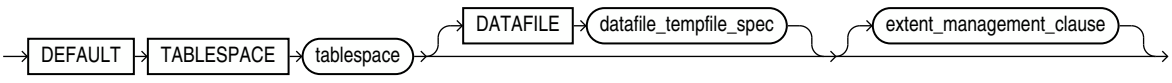
(*file_specification*::=)

***tablespace_clauses*::=**

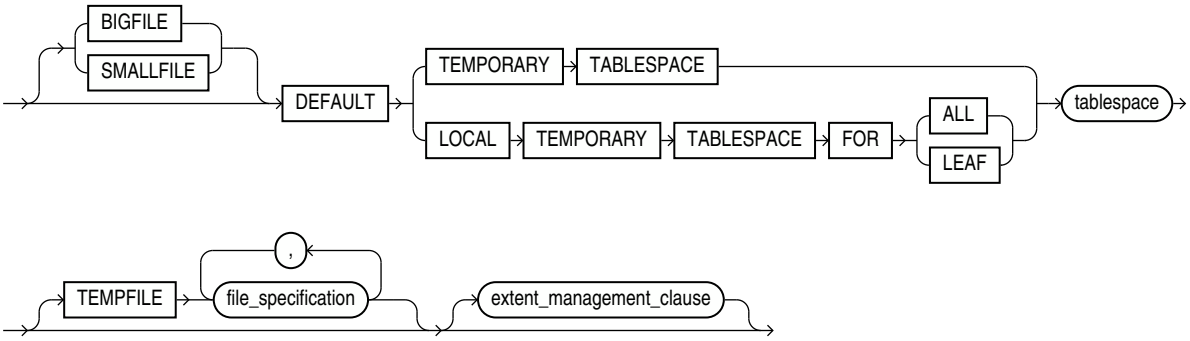


(*file_specification*::=, *default_tablespace*::=, *default_temp_tablespace*::=, *undo_tablespace*::=, *undo_tablespace*::=)

***default_tablespace*::=**

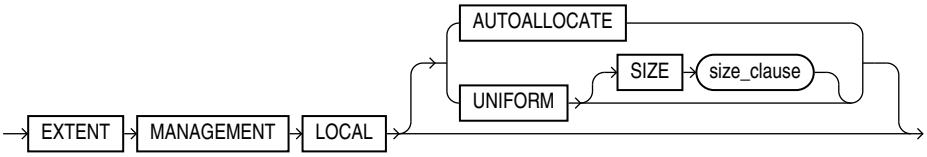


***default_temp_tablespace*::=**



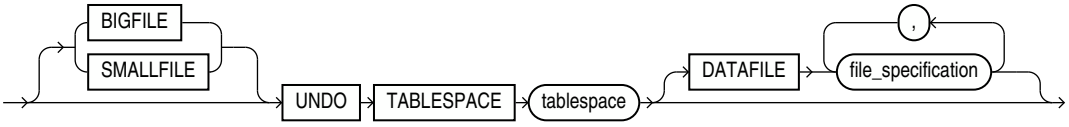
(*file_specification*::=)

***extent_management_clause*::=**



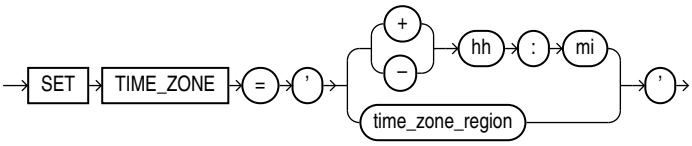
(*size_clause*::=)

***undo_tablespace*::=**

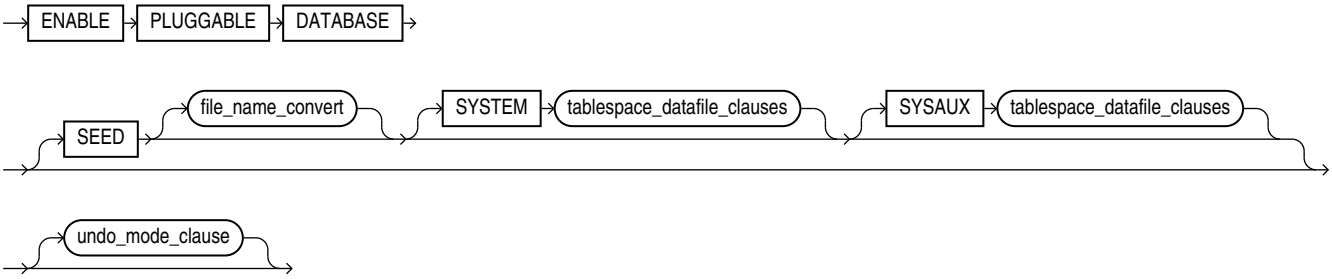


(*file_specification*::=)

***set_time_zone_clause*::=**

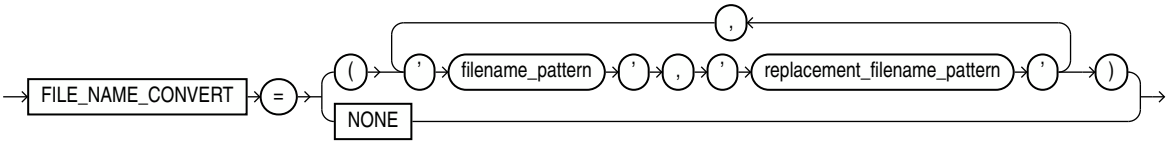


***enable_pluggable_database*::=**

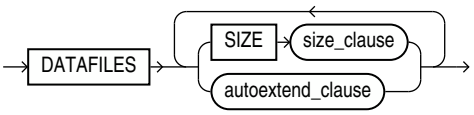


(*tablespace_datafile_clauses*::=, *undo_mode_clause*::=)

***file_name_convert*::=**

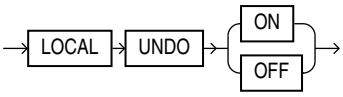


***tablespace_datafile_clauses*::=**



(*size_clause*::=, *autoextend_clause*::=)

***undo_mode_clause*::=**



Semantics

database

Specify the name of the database to be created. The name must match the value of the `DB_NAME` initialization parameter. The name can be up to 8 bytes long and can contain only ASCII characters. The following characters are valid in a database name: alphanumeric characters, underscore (_), number sign (#), and dollar sign ($). No other characters are valid. The database name must start with an alphabetic character. Oracle Database writes this name into the control file. If you subsequently issue an `ALTER DATABASE` statement that explicitly specifies a database name, then Oracle Database verifies that name with the name in the control file.

The database name is case insensitive and is stored in uppercase ASCII characters. If you specify the database name as a quoted identifier, then the quotation marks are silently ignored.

> **Note:**
>
> You cannot use special characters from European or Asian character sets in a database name. For example, characters with umlauts are not allowed.

If you omit the database name from a `CREATE DATABASE` statement, then Oracle Database uses the name specified by the initialization parameter `DB_NAME`. The `DB_NAME` initialization parameter must be set in the database initialization parameter file, and if you specify a different name from the value of that parameter, then the database returns an error. Refer to "Database Object Naming Rules " for additional rules to which database names should adhere.

USER SYS ..., USER SYSTEM ...

Use these clauses to establish passwords for the `SYS` and `SYSTEM` users. These clauses are not mandatory in this release. However, if you specify either clause, then you must specify both clauses.

If you do not specify these clauses, then Oracle Database creates the default password `change_on_install` for user `SYS` . You can change this password later with the `ALTER USER` statement.

> **See Also:**
>
> ALTER USER

CONTROLFILE REUSE Clause

Specify `CONTROLFILE REUSE` to reuse existing control files identified by the initialization parameter `CONTROL_FILES`, overwriting any information they currently contain. Normally you use this clause only when you are re-creating a database, rather than creating one for the first time. When you create a database for the first time, Oracle Database creates a control file in the default destination, which is dependent on the value or several initialization parameters. See `CREATE CONTROLFILE`, "Semantics".

You cannot use this clause if you also specify a parameter value that requires that the control file be larger than the existing files. These parameters are `MAXLOGFILES`, `MAXLOGMEMBERS`, `MAXLOGHISTORY`, `MAXDATAFILES`, and `MAXINSTANCES`.

If you omit this clause and any of the files specified by `CONTROL_FILES` already exist, then the database returns an error.

MAXDATAFILES Clause

Specify the initial sizing of the data files section of the control file at `CREATE DATABASE` or `CREATE CONTROLFILE` time. An attempt to add a file whose number is greater than `MAXDATAFILES`, but less than or equal to `DB_FILES`, causes the Oracle Database control file to expand automatically so that the data files section can accommodate more files.

The number of data files accessible to your instance is also limited by the initialization parameter DB_FILES.

MAXINSTANCES Clause

Specify the maximum number of instances that can simultaneously have this database mounted and open. This value takes precedence over the value of initialization parameter INSTANCES. The minimum value is 1. The maximum value is 1055. The default depends on your operating system.

CHARACTER SET Clause

Specify the character set the database uses to store data. The supported character sets and default value of this parameter depend on your operating system.

Restriction on CHARACTER SET

You cannot specify the AL16UTF16 character set as the database character set.

> **See Also:**
>
> *Oracle Database Globalization Support Guide* for more information about choosing a character set

NATIONAL CHARACTER SET Clause

Specify the national character set used to store data in columns specifically defined as NCHAR, NCLOB, or NVARCHAR2. Valid values are AL16UTF16 and UTF8. The default is AL16UTF16.

> **See Also:**
>
> *Oracle Database Globalization Support Guide* for information on Unicode data type support

SET DEFAULT TABLESPACE Clause

Use this clause to determine the default type of subsequently created tablespaces and of the SYSTEM and SYSAUX tablespaces. Specify either BIGFILE or SMALLFILE to set the default type of subsequently created tablespaces as a bigfile or smallfile tablespace, respectively.

- A **bigfile tablespace** contains only one data file or temp file, which can contain up to approximately 4 billion (2^{32}) blocks. The maximum size of the single data file or temp file is 128 terabytes (TB) for a tablespace with 32K blocks and 32TB for a tablespace with 8K blocks.
- A **smallfile tablespace** is a traditional Oracle tablespace, which can contain 1022 data files or temp files, each of which can contain up to approximately 4 million (2^{22}) blocks.

If you omit this clause, then Oracle Database creates bigfile tablespaces by default for SYSAUX, SYSTEM, and USER tablespaces .

> **See Also:**
>
> - *Oracle Database Administrator's Guide* for more information about bigfile tablespaces
> - "Setting the Default Type of Tablespaces: Example" for an example using this syntax

database_logging_clauses

Use the *database_logging_clauses* to determine how Oracle Database will handle redo log files for this database.

LOGFILE Clause

Specify one or more files to be used as redo log files. Use the *redo_log_file_spec* form of *file_specification* to create regular redo log files in an operating system file system or to create Oracle ASM disk group redo log files. When using a form of *ASM_filename*, you cannot specify the *autoextend_clause* of *redo_log_file_spec*.

The *redo_log_file_spec* clause specifies a redo log file group containing one or more redo log file members (copies). All redo log files specified in a CREATE DATABASE statement are added to redo log thread number 1.

> **See Also:**
>
> *file_specification* for a full description of this clause

If you omit the LOGFILE clause, then Oracle Database creates an Oracle-managed log file member in the default destination, which is one of the following locations (in order of precedence):

- If DB_CREATE_ONLINE_LOG_DEST_*n* is set, then the database creates a log file member in each directory specified, up to the value of the MAXLOGMEMBERS initialization parameter.
- If the DB_CREATE_ONLINE_LOG_DEST_*n* parameter is not set, but both the DB_CREATE_FILE_DEST and DB_RECOVERY_FILE_DEST initialization parameters are set, then the database creates one Oracle-managed log file member in each of those locations. The log file in the DB_CREATE_FILE_DEST destination is the first member.
- If only the DB_CREATE_FILE_DEST initialization parameter is specified, then Oracle Database creates a log file member in that location.
- If only the DB_RECOVERY_FILE_DEST initialization parameter is specified, then Oracle Database creates a log file member in that location.

In all these cases, the parameter settings must correctly specify operating system filenames or creation form Oracle ASM filenames, as appropriate.

If no values are set for any of these parameters, then the database creates a log file in the default location for the operating system on which the database is running. This log file is not an Oracle Managed File.

GROUP *integer*

Specify the number that identifies the redo log file group. The value of *integer* can range from 1 to the value of the MAXLOGFILES parameter. A database must have at least two redo log file groups. You cannot specify multiple redo log file groups having the same GROUP value. If you omit this parameter, then Oracle Database generates its value automatically. You can examine the GROUP value for a redo log file group through the dynamic performance view V$LOG.

MAXLOGFILES Clause

Specify the maximum number of redo log file groups that can ever be created for the database. Oracle Database uses this value to determine how much space to allocate in the control file for the names of redo log files. The default, minimum, and maximum values depend on your operating system.

MAXLOGMEMBERS Clause

Specify the maximum number of members, or copies, for a redo log file group. Oracle Database uses this value to determine how much space to allocate in the control file for the names of redo log files. The minimum value is 1. The maximum and default values depend on your operating system.

MAXLOGHISTORY Clause

This parameter is useful only if you are using Oracle Database in ARCHIVELOG mode with Oracle Real Application Clusters (Oracle RAC). Specify the maximum number of archived redo log files for automatic media recovery of Oracle RAC. The database uses this value to determine how much space to allocate in the control file for the names of archived redo log files. The minimum value is 0. The default value is a multiple of the MAXINSTANCES value and depends on your operating system. The maximum value is limited only by the maximum size of the control file.

ARCHIVELOG

Specify ARCHIVELOG if you want the contents of a redo log file group to be archived before the group can be reused. This clause prepares for the possibility of media recovery.

NOARCHIVELOG

Specify NOARCHIVELOG if the contents of a redo log file group need not be archived before the group can be reused. This clause does not allow for the possibility of media recovery.

The default is NOARCHIVELOG mode. After creating the database, you can change between ARCHIVELOG mode and NOARCHIVELOG mode with the ALTER DATABASE statement.

FORCE LOGGING

Use this clause to put the database into FORCE LOGGING mode. Oracle Database will log all changes in the database except for changes in temporary tablespaces and temporary segments. This setting takes precedence over and is independent of any NOLOGGING or FORCE LOGGING settings you specify for individual tablespaces and any NOLOGGING settings you specify for individual database objects.

FORCE LOGGING mode is persistent across instances of the database. If you shut down and restart the database, then the database is still in FORCE LOGGING mode. However, if you re-create the control file, then Oracle Database will take the database out of FORCE LOGGING mode unless you specify FORCE LOGGING in the CREATE CONTROLFILE statement.

> **Note:**
>
> `FORCE LOGGING` mode can have performance effects. Refer to *Oracle Database Administrator's Guide* for information on when to use this setting.

> **See Also:**
>
> CREATE CONTROLFILE

SET STANDBY NOLOGGING FOR DATA AVAILABILITY | LOAD PERFORMANCE

The `SET STANDBY NOLOGGING` disables logging on the standby. You can specify it in two modes:

- **SET STANDBY NOLOGGING FOR DATA AVAILABILITY** guarantees full data replication to the standby database. The primary and standby databases are synchronized during the load. In cases of network congestion the primary database will throttle its load.
- **SET STANDBY NOLOGGING FOR LOAD PERFORMANCE** to maintain speed of primary database load and synchronize with the standby later.

Restrictions SET STANDBY NOLOGGING

The`SET STANDBY NOLOGGING` clause cannot be used at the same time as `FORCE LOGGING`.

tablespace_clauses

Use the tablespace clauses to configure the `SYSTEM` and `SYSAUX` tablespaces and to specify a default temporary tablespace and an undo tablespace.

extent_management_clause

Use this clause to create a locally managed `SYSTEM` tablespace. If you omit this clause, then the `SYSTEM` tablespace will be dictionary managed.

> **Note:**
>
> When you create a locally managed `SYSTEM` tablespace, you cannot change it to be dictionary managed, nor can you create any other dictionary-managed tablespaces in this database.

If you specify this clause, then the database must have a default temporary tablespace, because a locally managed `SYSTEM` tablespace cannot store temporary segments.

- If you specify `EXTENT MANAGEMENT LOCAL` but you do not specify the `DATAFILE` clause, then you can omit the *`default_temp_tablespace`* clause. Oracle Database will create a default temporary tablespace called `TEMP` with one data file of size 10M with autoextend disabled.
- If you specify both `EXTENT MANAGEMENT LOCAL` and the `DATAFILE` clause, then you must also specify the *`default_temp_tablespace`* clause and explicitly specify a temp file for that temporary tablespace.

If you have opened the instance in automatic undo mode, similar requirements exist for the database undo tablespace:

- If you specify `EXTENT MANAGEMENT LOCAL` but you do not specify the `DATAFILE` clause, then you can omit the *undo_tablespace* clause. Oracle Database will create an undo tablespace named `SYS_UNDOTBS`.
- If you specify both `EXTENT MANAGEMENT LOCAL` and the `DATAFILE` clause, then you must also specify the *undo_tablespace* clause and explicitly specify a data file for that tablespace.

> **See Also:**
>
> *Oracle Database Administrator's Guide* for more information on locally managed and dictionary-managed tablespaces

DATAFILE Clause

Specify one or more files to be used as data files. All these files become part of the `SYSTEM` tablespace. Use the data *file_tempfile_spec* form of *file_specification* to create regular data files and temp files in an operating system file system or to create Oracle ASM disk group files.

> **Note:**
>
> This clause is optional, as is the `DATAFILE` clause of the *undo_tablespace* clause. Therefore, to avoid ambiguity, if your intention is to specify a data file for the `SYSTEM` tablespace with this clause, then do *not* specify it immediately after an *undo_tablespace* clause that does not include the optional `DATAFILE` clause. If you do so, then Oracle Database will interpret the `DATAFILE` clause to be part of the *undo_tablespace* clause.

The syntax for specifying data files for the `SYSTEM` tablespace is the same as that for specifying data files during tablespace creation using the `CREATE TABLESPACE` statement, whether you are storing files using Oracle ASM or in a file system.

> **See Also:**
>
> CREATE TABLESPACE for information on specifying data files

If you are running the database in automatic undo mode and you specify a data file name for the `SYSTEM` tablespace, then Oracle Database expects to generate data files for all tablespaces. Oracle Database does this automatically if you are using Oracle Managed Files—you have set a value for the `DB_CREATE_FILE_DEST` initialization parameter. However, if you are not using Oracle Managed Files and you specify this clause, then you must also specify the *undo_tablespace* clause and the *default_temp_tablespace* clause.

If you omit this clause, then:

- If the DB_CREATE_FILE_DEST initialization parameter is set, then Oracle Database creates a 100 MB Oracle-managed data file with a system-generated name in the default file destination specified in the parameter.
- If the DB_CREATE_FILE_DEST initialization parameter is not set, then Oracle Database creates one data file whose name and size depend on your operating system.

> **See Also:**
>
> *file_specification* for syntax

SYSAUX Clause

Oracle Database creates both the SYSTEM and SYSAUX tablespaces as part of every database. Use this clause if you are not using Oracle Managed Files and you want to specify one or more data files for the SYSAUX tablespace.

You must specify this clause if you have specified one or more data files for the SYSTEM tablespace using the DATAFILE clause. If you are using Oracle Managed Files and you omit this clause, then the database creates the SYSAUX data files in the default location set up for Oracle Managed Files.

If you have enabled Oracle Managed Files and you omit the SYSAUX clause, then the database creates the SYSAUX tablespace as an online, permanent, locally managed tablespace with one data file of 100 MB, with logging enabled and automatic segment-space management.

The syntax for specifying data files for the SYSAUX tablespace is the same as that for specifying data files during tablespace creation using the CREATE TABLESPACE statement, whether you are storing files using Oracle ASM or in a file system.

> **See Also:**
>
> - CREATE TABLESPACE for information on creating the SYSAUX tablespace during database upgrade and for information on specifying data files in a tablespace
> - *Oracle Database Administrator's Guide* for more information on creating the SYSAUX tablespace

default_tablespace

Specify this clause to create a default permanent tablespace for the database. Oracle Database creates a smallfile tablespace and subsequently will assign to this tablespace any non-SYSTEM users for whom you do not specify a different permanent tablespace. If you do not specify this clause, then the SYSTEM tablespace is the default permanent tablespace for non-SYSTEM users.

The DATAFILE clause and *extent_management_clause* have the same semantics they have in a CREATE TABLESPACE statement. Refer to "DATAFILE | TEMPFILE Clause" and *extent_management_clause* for information on these clauses.

default_temp_tablespace

Use this clause to create a default shared temporary tablespace or a default local temporary tablespace. Oracle Database will assign to these temporary tablespaces any users for whom you do not specify different temporary tablespaces.

- Specify `DEFAULT TEMPORARY TABLESPACE` to create a default shared temporary tablespace for the database. Shared temporary tablespaces were available in prior releases of Oracle Database and were called "temporary tablespaces." Elsewhere in this guide, the term "temporary tablespace" refers to a shared temporary tablespace unless specified otherwise. If you do not specify this clause, and if the database does not create a default shared temporary tablespace automatically in the process of creating a locally managed `SYSTEM` tablespace, then the `SYSTEM` tablespace is the default shared temporary tablespace.
- Starting with Oracle Database 12*c* Release 2 (12.2), you can specify `DEFAULT LOCAL TEMPORARY TABLESPACE` to create a default local temporary tablespace. Local temporary tablespaces are useful for Oracle Real Application Clusters and Oracle Flex Clusters. They store a separate, nonshared temp file for each database instance, which can improve I/O performance. A local temporary tablespace must be a `BIGFILE` tablespace.
  - Specify `FOR ALL` to instruct the database to create separate, nonshared temp files for all HUB and LEAF nodes.
  - Specify `FOR LEAF` to instruct the database to create separate nonshared temp files for only LEAF nodes. If you specify this clause, then HUB nodes will use the default shared temporary tablespace. For SQL operations that span both HUB and LEAF nodes, HUB nodes will use the default shared temporary tablespace and LEAF nodes will use the default local temporary tablespace.

  If you do not create a local temporary tablespace, then HUB and LEAF nodes will use the default shared temporary tablespace.

Specify `BIGFILE` or `SMALLFILE` to determine whether the default temporary tablespace is a bigfile or smallfile tablespace. These clauses have the same semantics as in the "SET DEFAULT TABLESPACE Clause ".

The `TEMPFILE` clause part of this clause is optional if you have enabled Oracle Managed Files by setting the `DB_CREATE_FILE_DEST` initialization parameter. If you have not specified a value for this parameter, then the `TEMPFILE` clause is required. If you have specified `BIGFILE`, then you can specify only one temp file in this clause.

The syntax for specifying temp files for the default temporary tablespace is the same as that for specifying temp files during temporary tablespace creation using the `CREATE TABLESPACE` statement, whether you are storing files using Oracle ASM or in a file system.

The *`extent_management_clause`* clause has the same semantics in `CREATE DATABASE` and `CREATE TABLESPACE` statements. For complete information, refer to the `CREATE TABLESPACE` ... *extent_management_clause* .

> **See Also:**
>
> CREATE TABLESPACE for information on specifying temp files

> **Note:**
>
> On some operating systems, Oracle does not allocate space for a temp file until the temp file blocks are actually accessed. This delay in space allocation results in faster creation and resizing of temp files, but it requires that sufficient disk space is available when the temp files are later used. To avoid potential problems, before you create or resize a temp file, ensure that the available disk space exceeds the size of the new temp file or the increased size of a resized temp file. The excess space should allow for anticipated increases in disk space use by unrelated operations as well. Then proceed with the creation or resizing operation.

Restrictions on Default Temporary Tablespaces

Default temporary tablespaces are subject to the following restrictions:

- You cannot specify the `SYSTEM` tablespace in this clause.
- The default temporary tablespace must have a standard block size.

undo_tablespace

If you have opened the instance in automatic undo mode (the `UNDO_MANAGEMENT` initialization parameter is set to `AUTO`, which is the default), then you can specify the *`undo_tablespace`* to create a tablespace to be used for undo data. Oracle strongly recommends that you use automatic undo mode. However, if you want undo space management to be handled by way of rollback segments, then you must omit this clause. You can also omit this clause if you have set a value for the `UNDO_TABLESPACE` initialization parameter. If that parameter has been set, and if you specify this clause, then *`tablespace`* must be the same as that parameter value.

- Specify `BIGFILE` if you want the undo tablespace to be a bigfile tablespace. A **bigfile tablespace** contains only one data file, which can be up to 8 exabytes (8 million terabytes) in size.

  Tablespaces `SYSAUX`, `SYSTEM`, and `USER` are `BIGFILE` by default.
- Specify `SMALLFILE` if you want the undo tablespace to be a smallfile tablespace. A **smallfile tablespace** is a traditional Oracle Database tablespace, which can contain 1022 data files or temp files, each of which can contain up to approximately 4 million (2^{22}) blocks.
- The `DATAFILE` clause part of this clause is optional if you have enabled Oracle Managed Files by setting the `DB_CREATE_FILE_DEST` initialization parameter. If you have not specified a value for this parameter, then the `DATAFILE` clause is required. If you have specified `BIGFILE`, then you can specify only one data file in this clause.

The syntax for specifying data files for the undo tablespace is the same as that for specifying data files during tablespace creation using the `CREATE TABLESPACE` statement, whether you are storing files using Oracle ASM or in a file system.

> **See Also:**
>
> CREATE TABLESPACE for information on specifying data files

If you specify this clause, then Oracle Database creates an undo tablespace named *tablespace*, creates the specified data file(s) as part of the undo tablespace, and assigns this tablespace as the undo tablespace of the instance. Oracle Database will manage undo data using this undo tablespace. The DATAFILE clause of this clause has the same behavior as described in "DATAFILE Clause".

If you have specified a value for the UNDO_TABLESPACE initialization parameter in your initialization parameter file before mounting the database, then you must specify the same name in this clause. If these names differ, then Oracle Database will return an error when you open the database.

If you omit this clause, then Oracle Database creates a default database with a default smallfile undo tablespace named SYS_UNDOTBS and assigns this default tablespace as the undo tablespace of the instance. This undo tablespace allocates disk space from the default files used by the CREATE DATABASE statement, and it has an initial extent of 10M. Oracle Database handles the system-generated data file as described in "DATAFILE Clause". If Oracle Database is unable to create the undo tablespace, then the entire CREATE DATABASE operation fails.

See Also:

- *Oracle Database Administrator's Guide* for information on automatic undo management and undo tablespaces
- CREATE TABLESPACE for information on creating an undo tablespace after database creation

set_time_zone_clause

Use the SET TIME_ZONE clause to set the time zone of the database. You can specify the time zone in two ways:

- By specifying a displacement from UTC (Coordinated Universal Time—formerly Greenwich Mean Time). The valid range of *hh:mi* is -12:00 to +14:00.
- By specifying a time zone region. To see a listing of valid time zone region names, query the TZNAME column of the V$TIMEZONE_NAMES dynamic performance view.

Note:

Oracle recommends that you set the database time zone to UTC (0:00). Doing so can improve performance, especially across databases, as no conversion of time zones will be required.

See Also:

Oracle Database Reference for information on the dynamic performance views

Oracle Database normalizes all TIMESTAMP WITH LOCAL TIME ZONE data to the time zone of the database when the data is stored on disk. If you do not specify the SET TIME_ZONE clause, then

the database uses the operating system time zone of the server. If the operating system time zone is not a valid Oracle Database time zone, then the database time zone defaults to UTC.

USER_DATA TABLESPACE Clause

This clause lets you create a tablespace that is used for storing user data and database options such as Oracle XML DB.

If you specify this clause when creating a multitenant container database (CDB), then the tablespace is created as part of the seed. Pluggable databases (PDBs) subsequently created using the seed will include this tablespace and its data file. The tablespace and data file specified in this clause are not used by the root.

Specify `BIGFILE` or `SMALLFILE` to determine whether the tablespace is a bigfile or smallfile tablespace. If you omit these clauses, then Oracle Database creates a tablespace of the type that you specify with the `SET DEFAULT TABLESPACE` clause. If you do not specify the `SET DEFAULT TABLESPACE` clause, then Oracle Database creates a smallfile tablespace. These clauses have the same semantics as in the "SET DEFAULT TABLESPACE Clause ".

Use the *`datafile_tempfile_spec`* clause to specify one or more data files for the tablespace. Refer to datafile_tempfile_spec for the full semantics of this clause.

enable_pluggable_database

Starting with Oracle Database 21c, the `ENABLE_PLUGGABLE_DATABASE` initialization parameter is set to `TRUE` by default. If you set the `ENABLE_PLUGGABLE_DATABASE` initialization parameter to `FALSE`, the command will fail.

The `CREATE DATABASE` *`enable_pluggable_database`* statement creates a CDB that contains a root and a seed container. You then create PDBs in the CDB by using the `CREATE PLUGGABLE DATABASE` statement.

> **See Also:**
>
> - Creating and configuring a cdb.
> - CREATE PLUGGABLE DATABASE
> - "Creating a CDB: Example"

file_name_convert

Use the *`file_name_convert`* clause to determine how the database generates the names of files (such as data files and wallet files) associated with the seed by using the names of files associated with the root.

- For *`filename_pattern`*, specify a string found in file names associated with the root.
- 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 seed.

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 seed 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.

tablespace_datafile_clauses

Use these clauses to specify attributes for all data files comprising the SYSTEM and SYSAUX tablespaces in the seed PDB. If you do not specify SIZE *size_clause*, then the data file size for a given tablespace will be set to a predetermined fraction of the size of the corresponding root data file. If you do not specify the *autoextend_clause*, then those values are inherited from the root.

Refer to *size_clause* and *autoextend_clause* for the full semantics of these clauses.

undo_mode_clause

This clause lets you specify local undo mode or shared undo mode for the CDB.

- Use LOCAL UNDO ON to specify local undo mode for the CDB. In this mode, every container in the CDB uses local undo.
- Use LOCAL UNDO OFF to specify shared undo mode for the CDB. In this mode, there is one active undo tablespace for a single-instance CDB, or for an Oracle RAC CDB, there is one active undo tablespace for each instance.

If you omit this clause, then the default is LOCAL UNDO OFF.

USING MIRROR COPY

Use this clause to create a database with *new_database_name* using the prepared files of the mirror copy, identified by *mirror_name*.

Examples

Creating a Database: Example

The following statement creates a database and fully specifies each argument:

```
CREATE DATABASE sample
   CONTROLFILE REUSE
   LOGFILE
      GROUP 1 ('diskx:log1.log', 'disky:log1.log') SIZE 50K,
      GROUP 2 ('diskx:log2.log', 'disky:log2.log') SIZE 50K
   MAXLOGFILES 5
   MAXLOGHISTORY 100
   MAXDATAFILES 10
   MAXINSTANCES 2
   ARCHIVELOG
   CHARACTER SET AL32UTF8
   NATIONAL CHARACTER SET AL16UTF16
   DATAFILE
      'disk1:df1.dbf' AUTOEXTEND ON,
      'disk2:df2.dbf' AUTOEXTEND ON NEXT 10M MAXSIZE UNLIMITED
   DEFAULT TEMPORARY TABLESPACE temp_ts
   UNDO TABLESPACE undo_ts
   SET TIME_ZONE = '+02:00';
```

This example assumes that you have enabled Oracle Managed Files by specifying a value for the DB_CREATE_FILE_DEST parameter in your initialization parameter file. Therefore no file specification is needed for the DEFAULT TEMPORARY TABLESPACE and UNDO TABLESPACE clauses.

Creating a CDB: Example

The following statement creates a CDB `newcdb`. The `ENABLE PLUGGABLE DATABASE` clause indicates that a CDB is being created. The CDB will contain a root (`CDB$ROOT`) and a seed (`PDB$SEED`). The `FILE_NAME_CONVERT` clause specifies that names of files for the seed will be generated by replacing `/u01/app/oracle/oradata/newcdb` in the names of files associated with the root with `/u01/app/oracle/oradata/pdbseed`.

```
CREATE DATABASE newcdb
  USER SYS IDENTIFIED BY sys_password
  USER SYSTEM IDENTIFIED BY system_password
  LOGFILE GROUP 1 ('/u01/logs/my/redo01a.log','/u02/logs/my/redo01b.log')
             SIZE 100M BLOCKSIZE 512,
          GROUP 2 ('/u01/logs/my/redo02a.log','/u02/logs/my/redo02b.log')
             SIZE 100M BLOCKSIZE 512,
          GROUP 3 ('/u01/logs/my/redo03a.log','/u02/logs/my/redo03b.log')
             SIZE 100M BLOCKSIZE 512
  MAXLOGHISTORY 1
  MAXLOGFILES 16
  MAXLOGMEMBERS 3
  MAXDATAFILES 1024
  CHARACTER SET AL32UTF8
  NATIONAL CHARACTER SET AL16UTF16
  EXTENT MANAGEMENT LOCAL
  DATAFILE '/u01/app/oracle/oradata/newcdb/system01.dbf'
    SIZE 700M REUSE AUTOEXTEND ON NEXT 10240K MAXSIZE UNLIMITED
  SYSAUX DATAFILE '/u01/app/oracle/oradata/newcdb/sysaux01.dbf'
    SIZE 550M REUSE AUTOEXTEND ON NEXT 10240K MAXSIZE UNLIMITED
  DEFAULT TABLESPACE deftbs
    DATAFILE '/u01/app/oracle/oradata/newcdb/deftbs01.dbf'
    SIZE 500M REUSE AUTOEXTEND ON MAXSIZE UNLIMITED
  DEFAULT TEMPORARY TABLESPACE tempts1
    TEMPFILE '/u01/app/oracle/oradata/newcdb/temp01.dbf'
    SIZE 20M REUSE AUTOEXTEND ON NEXT 640K MAXSIZE UNLIMITED
  UNDO TABLESPACE undotbs1
    DATAFILE '/u01/app/oracle/oradata/newcdb/undotbs01.dbf'
    SIZE 200M REUSE AUTOEXTEND ON NEXT 5120K MAXSIZE UNLIMITED
  ENABLE PLUGGABLE DATABASE
    SEED
    FILE_NAME_CONVERT = ('/u01/app/oracle/oradata/newcdb/',
                         '/u01/app/oracle/oradata/pdbseed/')
    SYSTEM DATAFILES SIZE 125M AUTOEXTEND ON NEXT 10M MAXSIZE UNLIMITED
    SYSAUX DATAFILES SIZE 100M
  USER_DATA TABLESPACE usertbs
    DATAFILE '/u01/app/oracle/oradata/pdbseed/usertbs01.dbf'
    SIZE 200M REUSE AUTOEXTEND ON MAXSIZE UNLIMITED;
```

CREATE DATABASE LINK

Purpose

Use the `CREATE DATABASE LINK` statement to create a database link. A **database link** is a schema object in one database that enables you to access objects on another database. The other database need not be an Oracle Database system. However, to access non-Oracle systems you must use Oracle Heterogeneous Services.

After you have created a database link, you can use it in SQL statements to refer to tables, views, and PL/SQL objects in the other database by appending *`@dblink`* to the table, view, or PL/SQL object name. You can query a table or view in the other database with the `SELECT` statement. You can also access remote tables and views using any `INSERT`, `UPDATE`, `DELETE`, or `LOCK TABLE` statement.

See Also:

- *Oracle Database Development Guide* for information about accessing remote tables or views with PL/SQL functions, procedures, packages, and data types
- *Oracle Database Administrator's Guide* for information on distributed database systems
- *Oracle Database Reference* for descriptions of existing database links in the `ALL_DB_LINKS`, `DBA_DB_LINKS`, and `USER_DB_LINKS` data dictionary views and for information on monitoring the performance of existing links through the `V$DBLINK` dynamic performance view
- ALTER DATABASE LINK for information on altering a database link when the password of a connection or authentication user changes.
- DROP DATABASE LINK for information on dropping existing database links
- INSERT , UPDATE , DELETE , and LOCK TABLE for using links in DML operations

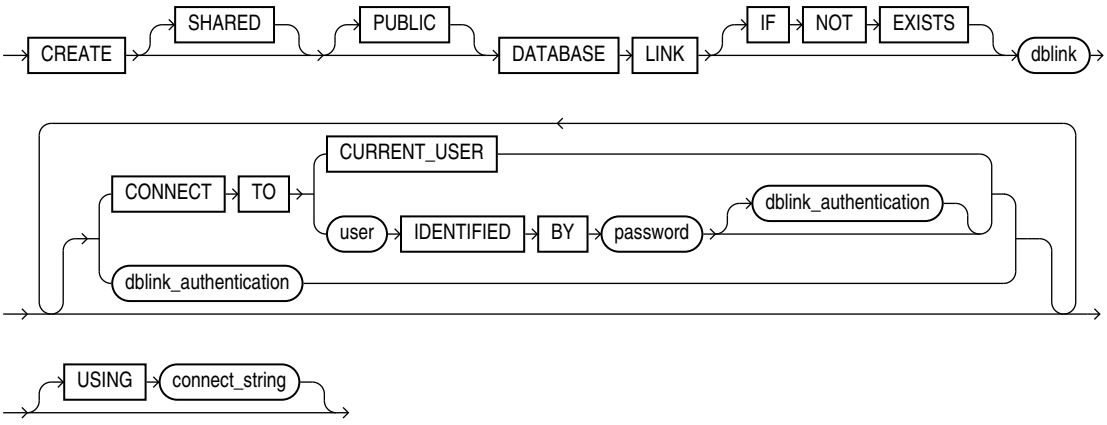
Prerequisites

To create a private database link, you must have the `CREATE DATABASE LINK` system privilege. To create a public database link, you must have the `CREATE PUBLIC DATABASE LINK` system privilege. Also, you must have the `CREATE SESSION` system privilege on the remote Oracle Database.

Oracle Net must be installed on both the local and remote Oracle Databases.

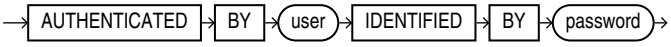
Syntax

***create_database_link*::=**



(*dblink*::=)

***dblink_authentication*::=**



Semantics

SHARED

Specify `SHARED` to create a database link that can be shared by multiple sessions using a single network connection from the source database to the target database. In a shared server configuration, shared database links can keep the number of connections into the remote database from becoming too large. Shared links are typically also public database links. However, a shared private database link can be useful when many clients access the same local schema, and therefore use the same private database link.

In a shared database link, multiple sessions in the source database share the same connection to the target database. Once a session is established on the target database, that session is disassociated from the connection, to make the connection available to another session on the source database. To prevent an unauthorized session from attempting to connect through the database link, when you specify `SHARED` you must also specify the *`dblink_authentication`* clause for the users authorized to use the database link.

> **See Also:**
>
> *Oracle Database Administrator's Guide* for more information about shared database links

PUBLIC

Specify `PUBLIC` to create a public database link visible to all users. If you omit this clause, then the database link is private and is available only to you.

The data accessible on the remote database depends on the identity the database link uses when connecting to the remote database:

- If you specify `CONNECT TO` *`user`* `IDENTIFIED BY` *`password`*, then the database link connects with the specified user and password.
- If you specify `CONNECT TO CURRENT_USER`, then the database link connects with the user in effect based on the scope in which the link is used.
- If you omit both of those clauses, then the database link connects to the remote database as the locally connected user.

> **See Also:**
>
> "Defining a Public Database Link: Example"

dblink

Specify the complete or partial name of the database link. If you specify only the database name, then Oracle Database implicitly appends the database domain of the local database.

Use only ASCII characters for *`dblink`*. Multibyte characters are not supported. The database link name is case insensitive and is stored in uppercase ASCII characters. If you specify the database name as a quoted identifier, then the quotation marks are silently ignored.

If the value of the GLOBAL_NAMES initialization parameter is TRUE, then the database link must have the same name as the database to which it connects. If the value of GLOBAL_NAMES is FALSE, and if you have changed the global name of the database, then you can specify the global name.

The maximum number of database links that can be open in one session or one instance of an Oracle RAC configuration depends on the value of the OPEN_LINKS and OPEN_LINKS_PER_INSTANCE initialization parameters.

Restriction on Creating Database Links

You cannot create a database link in another user's schema, and you cannot qualify *dblink* with the name of a schema. Periods are permitted in names of database links, so Oracle Database interprets the entire name, such as ralph.linktosales, as the name of a database link in your schema rather than as a database link named linktosales in the schema ralph.

> **See Also:**
>
> - "References to Objects in Remote Databases " for guidelines for naming database links
> - *Oracle Database Reference* for information on the GLOBAL_NAMES, OPEN_LINKS, and OPEN_LINKS_PER_INSTANCE initialization parameters
> - "RENAME GLOBAL_NAME Clause" (an ALTER DATABASE clause) for information on changing the database global name

IF NOT EXISTS

Specifying IF NOT EXISTS has the following effects:

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

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

CONNECT TO Clause

The CONNECT TO clause lets you specify the user and credentials, if any, to be used to connect to the remote database.

CURRENT_USER Clause

Specify CURRENT_USER to create a **current user database link**. The current user must be a global user with a valid account on the remote database.

If the database link is used directly rather than from within a stored object, then the current user is the same as the connected user.

When executing a stored object (such as a procedure, view, or trigger) that initiates a database link, CURRENT_USER is the name of the user that owns the stored object, and not the name of the user that called the object. For example, if the database link appears inside procedure scott.p (created by scott), and user jane calls procedure scott.p, then the current user is scott.

However, if the stored object is an invoker-rights function, procedure, or package, then the invoker's authorization ID is used to connect as a remote user. For example, if the privileged database link appears inside procedure `scott.p` (an invoker-rights procedure created by `scott`), and user Jane calls procedure `scott.p`, then `CURRENT_USER` is `jane` and the procedure executes with Jane's privileges.

> **See Also:**
>
> - CREATE FUNCTION for more information on invoker-rights functions
> - "Defining a CURRENT_USER Database Link: Example"

user* IDENTIFIED BY *password

Specify the user name and password used to connect to the remote database using a **fixed user database link**. If you omit this clause, then the database link uses the user name and password of each user who is connected to the database. This is called a **connected user database link**.

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

> **See Also:**
>
> "Defining a Fixed-User Database Link: Example"

dblink_authentication

You can specify this clause only if you are creating a shared database link—that is, you have specified the `SHARED` clause. Specify the username and password on the target instance. This clause authenticates the user to the remote server and is required for security. The specified username and password must be a valid username and password on the remote instance. The username and password are used only for authentication. No other operations are performed on behalf of this user.

CONNECT WITH Clause

Use `CONNECT WITH` to specify the credential object that stores the username and password to connect to the remote database.

You can create, update, or drop the credential using the `DBMS_CREDENTIAL` package.

You can use a credential object belonging to another user, if that user has granted you execute privileges on the credential object.

USING '*connect string*'

Specify the service name of a remote database. If you specify only the database name, then Oracle Database implicitly appends the database domain to the connect string to create a complete service name. Therefore, if the database domain of the remote database is different from that of the current database, then you must specify the complete service name.

> **See Also:**
>
> *Oracle Database Administrator's Guide* for information on specifying remote databases

Examples

The examples that follow assume two databases, one with the database name `local` and the other with the database name `remote`. The examples use the Oracle Database domain. Your database domain will be different.

Defining a Public Database Link: Example

The following statement defines a shared public database link named `remote` that refers to the database specified by the service name `remote`:

```
CREATE PUBLIC DATABASE LINK remote
   USING 'remote';
```

This database link allows user `hr` on the `local` database to update a table on the `remote` database (assuming `hr` has appropriate privileges):

```
UPDATE employees@remote
   SET salary=salary*1.1
   WHERE last_name = 'Baer';
```

Defining a Fixed-User Database Link: Example

In the following statement, user `hr` on the `remote` database defines a fixed-user database link named `local` to the `hr` schema on the `local` database:

```
CREATE DATABASE LINK local
   CONNECT TO hr IDENTIFIED BY password
   USING 'local';
```

After this database link is created, `hr` can query tables in the schema `hr` on the `local` database in this manner:

```
SELECT * FROM employees@local;
```

User `hr` can also use DML statements to modify data on the `local` database:

```
INSERT INTO employees@local
   (employee_id, last_name, email, hire_date, job_id)
   VALUES (999, 'Claus', 'sclaus@example.com', SYSDATE, 'SH_CLERK');

UPDATE jobs@local SET min_salary = 3000
   WHERE job_id = 'SH_CLERK';

DELETE FROM employees@local
   WHERE employee_id = 999;
```

Using this fixed database link, user hr on the `remote` database can also access tables owned by other users on the same database. This statement assumes that user `hr` has the `READ` or `SELECT` privilege on the `oe.customers` table. The statement connects to the user `hr` on the `local` database and then queries the `oe.customers` table:

```
SELECT * FROM oe.customers@local;
```

Defining a CURRENT_USER Database Link: Example

The following statement defines a current-user database link to the `remote` database, using the entire service name as the link name:

```
CREATE DATABASE LINK remote.us.example.com
   CONNECT TO CURRENT_USER
   USING 'remote';
```

The user who issues this statement must be a global user registered with the LDAP directory service.

You can create a synonym to hide the fact that a particular table is on the `remote` database. The following statement causes all future references to `emp_table` to access the `employees` table owned by `hr` on the `remote` database:

```
CREATE SYNONYM emp_table
   FOR oe.employees@remote.us.example.com;
```

CREATE DIMENSION

Purpose

Use the `CREATE DIMENSION` statement to create a **dimension**. A dimension defines a parent-child relationship between pairs of column sets, where all the columns of a column set must come from the same table. However, columns in one column set (called a **level**) can come from a different table than columns in another set. The optimizer uses these relationships with materialized views to perform **query rewrite**. The SQL Access Advisor uses these relationships to recommend creation of specific materialized views.

> **Note:**
>
> Oracle Database does not automatically validate the relationships you declare when creating a dimension. To validate the relationships specified in the *`hierarchy_clause`* and the *`dimension_join_clause`* of `CREATE DIMENSION`, you must run the `DBMS_OLAP.VALIDATE_DIMENSION` procedure.

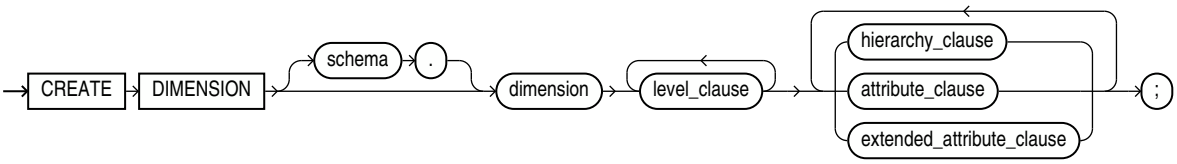
> **See Also:**
>
> - CREATE MATERIALIZED VIEW for more information on materialized views
> - *Oracle Database SQL Tuning Guide* for more information on query rewrite, the optimizer and the SQL Access Advisor

Prerequisites

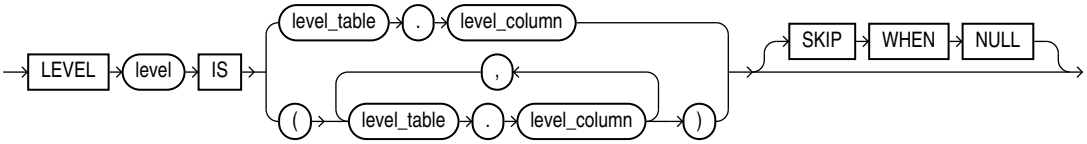
To create a dimension in your own schema, you must have the `CREATE DIMENSION` system privilege. To create a dimension in another user's schema, you must have the `CREATE ANY DIMENSION` system privilege. In either case, you must have the `READ` or `SELECT` object privilege on any objects referenced in the dimension.

Syntax

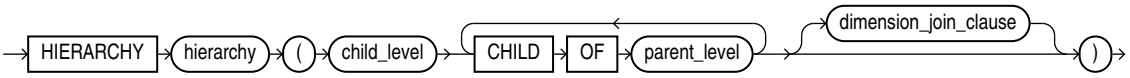
***create_dimension*::=**



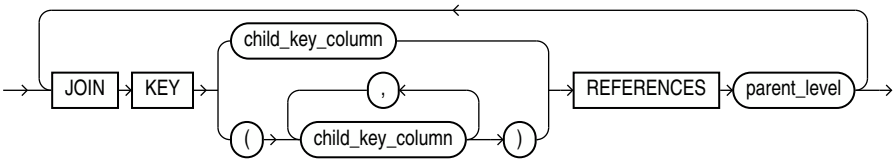
***level_clause*::=**



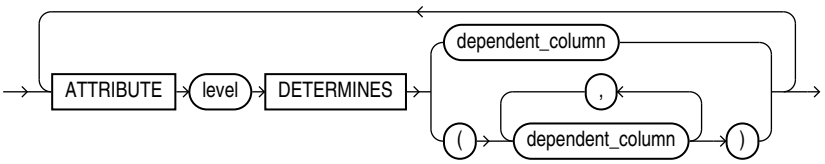
***hierarchy_clause*::=**



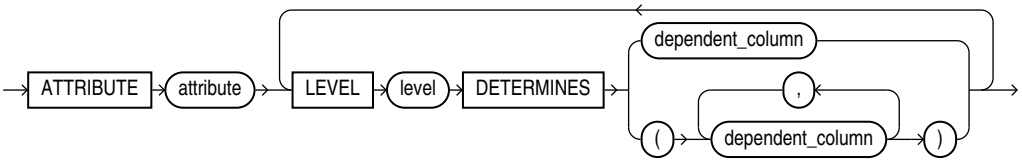
***dimension_join_clause*::=**



***attribute_clause*::=**



***extended_attribute_clause*::=**



Semantics

schema

Specify the schema in which the dimension will be created. If you do not specify *schema*, then Oracle Database creates the dimension in your own schema.

dimension

Specify the name of the dimension. The name must satisfy the requirements listed in "Database Object Naming Rules ".

level_clause

The *level_clause* defines a level in the dimension. A level defines dimension hierarchies and attributes.

level

Specify the name of the level.

level_table . level_column

Specify the columns in the level. You can specify up to 32 columns. The tables you specify in this clause must already exist.

SKIP WHEN NULL

Specify this clause to indicate that if the specified level is `NULL`, then the level is to be skipped. This clause lets you preserve the hierarchical chain of parent-child relationship by an alternative path that skips over the specified level. See *hierarchy_clause*.

Restrictions on Dimension Level Columns

Dimension level columns are subject to the following restrictions:

- All of the columns in a level must come from the same table.
- If columns in different levels come from different tables, then you must specify the *dimension_join_clause*.
- The set of columns you specify must be unique to this level.
- The columns you specify cannot be specified in any other dimension.
- Each `level_column` must be non-null unless the level is specified with `SKIP WHEN NULL`. The non-null columns need not have `NOT NULL` constraints. The column for which you specify `SKIP WHEN NULL` cannot have a `NOT NULL` constraint).

hierarchy_clause

The *hierarchy_clause* defines a linear hierarchy of levels in the dimension. Each hierarchy forms a chain of parent-child relationships among the levels in the dimension. Hierarchies in a dimension are independent of each other. They may, but need not, have columns in common.

Each level in the dimension should be specified at most once in this clause, and each level must already have been named in the *level_clause*.

hierarchy

Specify the name of the hierarchy. This name must be unique in the dimension.

child_level

Specify the name of a level that has an n:1 relationship with a parent level. The *level_columns* of *child_level* cannot be null, and each *child_level* value uniquely determines the value of the next named *parent_level*.

If the child *level_table* is different from the parent *level_table*, then you must specify a join relationship between them in the *dimension_join_clause*.

parent_level

Specify the name of a level.

dimension_join_clause

The *dimension_join_clause* lets you specify an inner equijoin relationship for a dimension whose columns are contained in multiple tables. This clause is required and permitted only when the columns specified in the hierarchy are not all in the same table.

child_key_column

Specify one or more columns that are join-compatible with columns in the parent level.

If you do not specify the schema and table of each *child_column*, then the schema and table are inferred from the `CHILD OF` relationship in the *hierarchy_clause*. If you do specify the schema and column of a *child_key_column*, then the schema and table must match the schema and table of columns in the child of *parent_level* in the *hierarchy_clause*.

parent_level

Specify the name of a level.

Restrictions on Join Dimensions

Join dimensions are subject to the following restrictions:

- You can specify only one *dimension_join_clause* for a given pair of levels in the same hierarchy.
- The *child_key_columns* must be non-null, and the parent key must be unique and non-null. You need not define constraints to enforce these conditions, but queries may return incorrect results if these conditions are not true.
- Each child key must join with a key in the *parent_level* table.
- Self-joins are not permitted. The *child_key_columns* cannot be in the same table as *parent_level*.

- All of the child key columns must come from the same table.
- The number of child key columns must match the number of columns in *parent_level*, and the columns must be joinable.
- You cannot specify multiple child key columns unless the parent level consists of multiple columns.

attribute_clause

The *attribute_clause* lets you specify the columns that are uniquely determined by a hierarchy level. The columns in *level* must all come from the same table as the *dependent_columns*. The *dependent_columns* need not have been specified in the *level_clause*.

For example, if the hierarchy levels are `city`, `state`, and `country`, then `city` might determine `mayor`, `state` might determine `governor`, and `country` might determine `president`.

extended_attribute_clause

This clause lets you specify an attribute name for one or more level-to-column relations. The type of attribute you create with this clause is not different from the type of attribute created using the *attribute_clause*. The only difference is that this clause lets you assign a name to the attribute that is different from the level name.

Examples

Creating a Dimension: Examples

This statement was used to create the `customers_dim` dimension in the sample schema `sh`:

```
CREATE DIMENSION customers_dim
   LEVEL customer   IS (customers.cust_id)
   LEVEL city       IS (customers.cust_city)
   LEVEL state      IS (customers.cust_state_province)
   LEVEL country    IS (countries.country_id)
   LEVEL subregion  IS (countries.country_subregion)
   LEVEL region     IS (countries.country_region)
   HIERARCHY geog_rollup (
      customer      CHILD OF
      city          CHILD OF
      state         CHILD OF
      country       CHILD OF
      subregion     CHILD OF
      region
   JOIN KEY (customers.country_id) REFERENCES country
   )
   ATTRIBUTE customer DETERMINES
   (cust_first_name, cust_last_name, cust_gender,
    cust_marital_status, cust_year_of_birth,
    cust_income_level, cust_credit_limit)
   ATTRIBUTE country DETERMINES (countries.country_name)
;
```

Creating a Dimension with Extended Attributes: Example

Alternatively, the *extended_attribute_clause* could have been used instead of the *attribute_clause*, as shown in the following example:

```
CREATE DIMENSION customers_dim
   LEVEL customer   IS (customers.cust_id)
   LEVEL city       IS (customers.cust_city)
```

```
   LEVEL state      IS (customers.cust_state_province)
   LEVEL country    IS (countries.country_id)
   LEVEL subregion  IS (countries.country_subregion)
   LEVEL region     IS (countries.country_region)
   HIERARCHY geog_rollup (
      customer      CHILD OF
      city          CHILD OF
      state         CHILD OF
      country       CHILD OF
      subregion     CHILD OF
      region
   JOIN KEY (customers.country_id) REFERENCES country
   )
   ATTRIBUTE customer_info LEVEL customer DETERMINES
   (cust_first_name, cust_last_name, cust_gender,
    cust_marital_status, cust_year_of_birth,
    cust_income_level, cust_credit_limit)
   ATTRIBUTE country DETERMINES (countries.country_name);
```

Creating a Dimension with NULL Column Values: Example

The following example shows how to create the dimension if one of the level columns is null and you want to preserve the hierarchical chain. The example uses the `cust_marital_status` column for simplicity because it is not a `NOT NULL` column. If it had such a constraint, then you would have to disable the constraint before using the `SKIP WHEN NULL` clause.

```
CREATE DIMENSION customers_dim
   LEVEL customer IS (customers.cust_id)
   LEVEL status IS (customers.cust_marital_status) SKIP WHEN NULL
   LEVEL city IS (customers.cust_city)
   LEVEL state IS (customers.cust_state_province)
   LEVEL country IS (countries.country_id)
   LEVEL subregion IS (countries.country_subregion) SKIP WHEN NULL
   LEVEL region IS (countries.country_region)
   HIERARCHY geog_rollup (
      customer CHILD OF
      city CHILD OF
      state CHILD OF
      country CHILD OF
      subregion CHILD OF
      region
   JOIN KEY (customers.country_id) REFERENCES country
   )
   ATTRIBUTE customer DETERMINES
   (cust_first_name, cust_last_name, cust_gender,
    cust_marital_status, cust_year_of_birth,
    cust_income_level, cust_credit_limit)
   ATTRIBUTE country DETERMINES (countries.country_name)
;
```

CREATE DIRECTORY

Purpose

Use the `CREATE DIRECTORY` statement to create a directory object. A directory object specifies an alias for a directory on the server file system where external binary file LOBs (`BFILE`s) and external table data are located. You can use directory names when referring to `BFILE`s in your PL/SQL code and OCI calls, rather than hard coding the operating system path name, for management flexibility.

All directories are created in a single namespace and are not owned by an individual schema. You can secure access to the BFILEs stored within the directory structure by granting object privileges on the directories to specific users.

> **See Also:**
>
> - "Large Object (LOB) Data Types " for more information on BFILE objects
> - GRANT for more information on granting object privileges
> - *external_table_clause*::= of CREATE TABLE

Prerequisites

You must have the CREATE ANY DIRECTORY system privilege to create directories.

When you create a directory, you are automatically granted the READ, WRITE, and EXECUTE object privileges on the directory, and you can grant these privileges to other users and roles. The DBA can also grant these privileges to other users and roles.

WRITE privileges on a directory are useful in connection with external tables. They let the grantee determine whether the external table agent can write a log file or a bad file to the directory.

For file storage, you must also create a corresponding operating system directory, an Oracle Automatic Storage Management (Oracle ASM) disk group, or a directory within an Oracle ASM disk group. Your system or database administrator must ensure that the operating system directory has the correct read and write permissions for Oracle Database processes.

Privileges granted for the directory are created independently of the permissions defined for the operating system directory, and the two may or may not correspond exactly. For example, an error occurs if sample user hr is granted READ privilege on the directory object but the corresponding operating system directory does not have READ permission defined for Oracle Database processes.

Restrictions

Symbolic links are not allowed in the directory object paths or filenames when opening BFILE objects. The entire directory path and filename is checked and the following error is returned if any symbolic link is found:

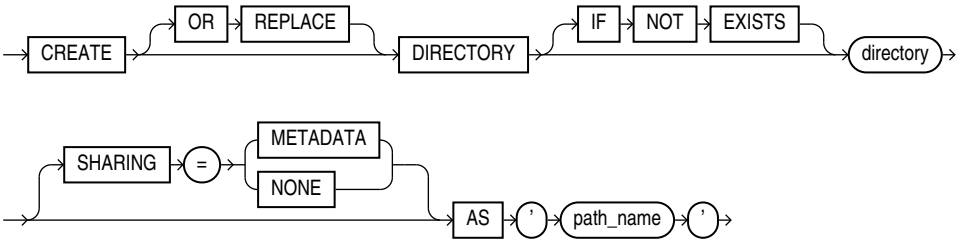
```
ORA-22288: file or LOB operation FILEOPEN failed soft link in path
```

Workaround

If the database directory object or filename you are trying to open contains symbolic links, change it to provide the real path and filename.

Syntax

***create_directory*::=**



Semantics

OR REPLACE

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

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

> **See Also:**
>
> DROP DIRECTORY for information on removing a directory from the database

IF NOT EXISTS

Specifying `IF NOT EXISTS` has the following effects:

- If the directory does not exist, a new directory is created at the end of the statement.
- If the directory exists, this is the directory 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.`

SHARING

This clause applies only when creating a directory in an application root. This type of directory is called an application common object and it can be shared with the application PDBs that belong to the application root. To determine how the directory is shared, specify one of the following sharing attributes:

- `METADATA` - A metadata link shares the directory's metadata, but its data is unique to each container. This type of directory is referred to as a **metadata-linked application common object**.

- `NONE` - The directory is not shared.

If you omit this clause, then the database uses the value of the `DEFAULT_SHARING` initialization parameter to determine the sharing attribute of the directory. If the `DEFAULT_SHARING` initialization parameter does not have a value, then the default is `METADATA`.

You cannot change the sharing attribute of a directory after it is created.

> **See Also:**
>
> - *Oracle Database Reference* for more information on the `DEFAULT_SHARING` initialization parameter
> - *Oracle Database Administrator's Guide* for complete information on creating application common objects

directory

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

Oracle Database does not verify that the directory you specify actually exists. Therefore, take care that you specify a valid directory in your operating system. In addition, if your operating system uses case-sensitive path names, then be sure you specify the directory in the correct format. You need not include a trailing slash at the end of the path name.

Do not refer to a parent directory in the directory name. For example, the following syntax is valid:

```
CREATE DIRECTORY mydir AS '/scratch/data/file_data';
```

However, the following syntax is not valid:

```
CREATE DIRECTORY mydir AS '/scratch/../file_data';
```

path_name

Specify the full path name of the operating system directory of the server where the files are located. The single quotation marks are required, with the result that the path name is case sensitive.

Examples

Creating a Directory: Examples

The following statement creates a directory database object that points to a directory on the server:

```
CREATE DIRECTORY admin AS '/disk1/oracle/admin';
```

The following statement redefines directory database object `bfile_dir` to enable access to `BFILE`s stored in the operating system directory /`usr/bin/bfile_dir`:

```
CREATE OR REPLACE DIRECTORY bfile_dir AS '/usr/bin/bfile_dir';
```

CREATE DISKGROUP

> **Note:**
>
> This SQL statement is valid only if you are using Oracle ASM and you have started an Oracle ASM instance. You must issue this statement from within the Oracle ASM instance, not from a normal database instance. For information on starting an Oracle ASM instance, refer to *Oracle Automatic Storage Management Administrator's Guide*.

Purpose

Use the `CREATE DISKGROUP` clause to name a group of disks and specify that Oracle Database should manage the group for you. Oracle Database manages a disk group as a logical unit and evenly spreads each file across the disks to balance I/O. Oracle Database also automatically distributes database files across all available disks in disk groups and rebalances storage automatically whenever the storage configuration changes.

This statement creates a disk group, assigns one or more disks to the disk group, and mounts the disk group for the first time. Note that `CREATE DISKGROUP` only mounts a disk group on the local node. If you want Oracle ASM to mount the disk group automatically in subsequent instances, then you must add the disk group name to the value of the `ASM_DISKGROUPS` initialization parameter in the initialization parameter file. If you use an `SPFILE`, then the disk group is added to the initialization parameter automatically.

> **See Also:**
>
> - ALTER DISKGROUP for information on modifying disk groups
> - *Oracle Automatic Storage Management Administrator's Guide* for information on Oracle ASM and using disk groups to simplify database administration
> - `ASM_DISKGROUPS` for more information about adding disk group names to the initialization parameter file
> - `V$ASM_OPERATION` for information on monitoring Oracle ASM operations
> - DROP DISKGROUP for information on dropping a disk group

Prerequisites

You must have the `SYSASM` system privilege to issue this statement.

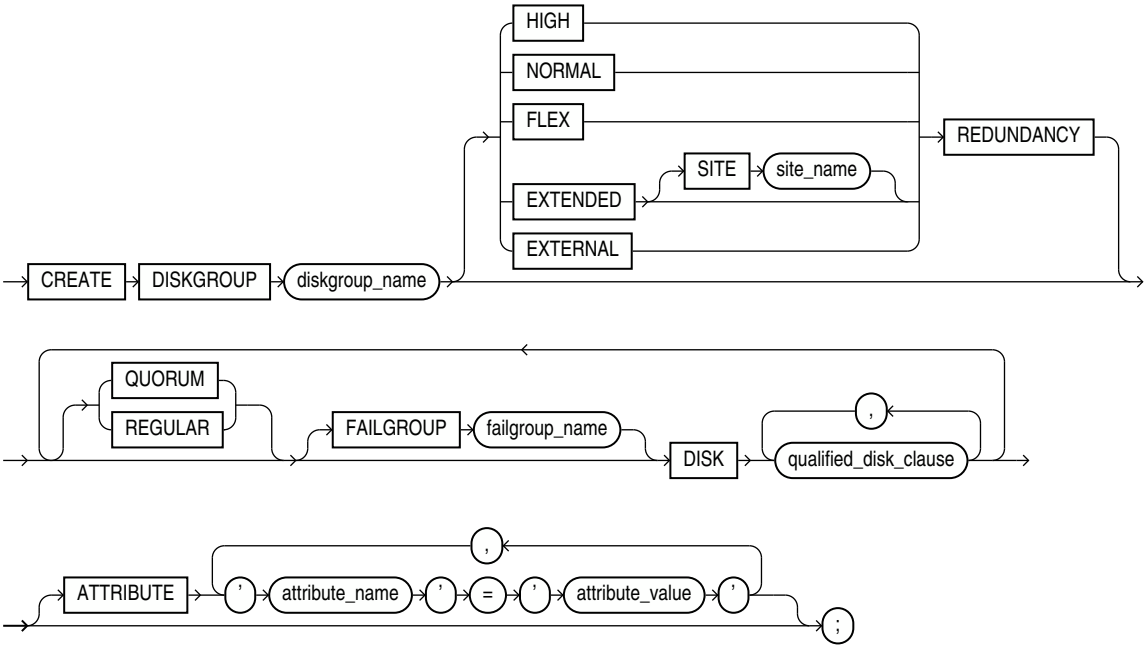
Before issuing this statement, you must format the disks using an operating system format utility. Also ensure that the Oracle Database user has read/write permission and the disks can be discovered using the `ASM_DISKSTRING`.

When you store your database files in Oracle ASM disk groups, rather than in a file system, before the database instance can access your files in the disk groups, you must configure and start up an Oracle ASM instance to manage the disk groups.

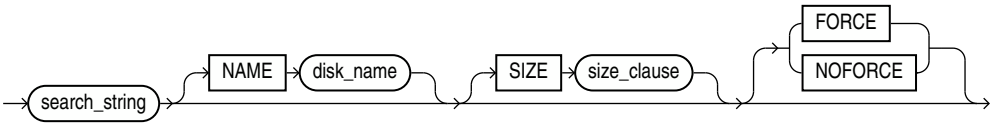
Each database instance communicates with a single Oracle ASM instance on the same node as the database. Multiple database instances on the same node can communicate with a single Oracle ASM instance.

Syntax

***create_diskgroup*::=**



***qualified_disk_clause*::=**



(*size_clause*::=)

diskgroup_name

Specify the name of the disk group. The name must satisfy the requirements listed in "Database Object Naming Rules ". However, disk groups are not schema objects.

> **Note:**
>
> Oracle does not recommend using quoted identifiers for disk group names. These quoted identifiers are accepted when issuing the `CREATE DISKGROUP` statement in SQL*Plus, but they may not be valid when using other tools that manage disk groups.

REDUNDANCY Clause

The REDUNDANCY clause lets you specify the redundancy level of the disk group.

- NORMAL REDUNDANCY requires the existence of at least two failure groups (see the FAILGROUP clause that follows). Oracle ASM provides redundancy for all files in the disk group according to the attributes specified in the disk group templates. NORMAL REDUNDANCY disk groups can tolerate the loss of one group. Refer to ALTER DISKGROUP ... *diskgroup_template_clauses* for more information on disk group templates.

  NORMAL REDUNDANCY is the default if you omit the REDUNDANCY clause. Therefore, if you omit this clause, you must create at least two failure groups, or the create operation will fail.
- HIGH REDUNDANCY requires the existence of at least three failure groups. Oracle ASM fixes mirroring at 3-way mirroring, with each extent getting two mirrored copies. HIGH REDUNDANCY disk groups can tolerate the loss of two failure groups.
- FLEX REDUNDANCY is a type of disk group that allows a database to specify its own redundancy after the disk group is created. A file's redundancy can also be changed after its creation. This type of disk group supports Oracle ASM file groups and quota groups. A flex disk group requires the existence of at least three failure groups. If a flex disk group has fewer than five failure groups, then it can tolerate the loss of one; otherwise, it can tolerate the loss of two failure groups. To create a flex disk group, the COMPATIBLE.ASM and COMPATIBLE.RDBMS disk group attributes must be set to 12.2 or greater.
- EXTENDED REDUNDANCY is a disk group that has all the features of a flex disk group in addition to being highly available in an extended cluster environment. The cluster contains nodes that span multiple physically separated sites. For more see About Oracle ASM Extended Disk Groups

  You can use the SITE keyword to specify the redundancy of files and file groups in an extended disk group for each site, rather than for each disk group.
- EXTERNAL REDUNDANCY indicates that Oracle ASM does not provide any redundancy for the disk group. The disks within the disk group must provide redundancy (for example, using a storage array), or you must be willing to tolerate loss of the disk group if a disk fails (for example, in a test environment). You cannot specify the FAILGROUP clause if you specify EXTERNAL REDUNDANCY.

You cannot change the redundancy level after the disk group has been created, with the following exception: You can convert a normal or high redundancy disk group to a flex disk group. For more information, see the *convert_redundancy_clause* of ALTER DISKGROUP.

QUORUM | REGULAR

Use these keywords to qualify either failure group or disk specifications.

- REGULAR disks, or disks in non-quorum failure groups, can contain any files.
- QUORUM disks, or disks in quorum failure groups, cannot contain any database files, the Oracle Cluster Registry (OCR), or dynamic volumes. However, QUORUM disks can contain the voting file for Cluster Synchronization Services (CSS). Oracle ASM uses quorum disks or disks in quorum failure groups for voting files whenever possible.

  A quorum failure group is not considered when determining redundancy requirements with respect to storing user data.

If you specify neither keyword, then REGULAR is the default.

Specify either `QUORUM` or `REGULAR` before the keyword `FAILGROUP` if you are explicitly specifying the failure group. If you are creating a disk group with implicitly created failure groups, then specify these keywords before the keyword `DISK`.

> **See Also:**
>
> *Oracle Automatic Storage Management Administrator's Guide* for more information about quorum and regular disks and failure groups

FAILGROUP Clause

Use this clause to specify a name for one or more failure groups. If you omit this clause, and you have specified `NORMAL` or `HIGH REDUNDANCY`, then Oracle Database automatically adds each disk in the disk group to its own failure group. The implicit name of the failure group is the same as the operating system independent disk name (see "NAME Clause").

You cannot specify this clause if you are creating an `EXTERNAL REDUNDANCY` disk group.

qualified_disk_clause

Specify `DISK` *`qualified_disk_clause`* to add a disk to a disk group.

search_string

For each disk you are adding to the disk group, specify the operating system dependent search string that Oracle ASM will use to find the disk. The *`search_string`* must point to a subset of the disks returned by discovery using the strings in the `ASM_DISKSTRING` initialization parameter. If *`search_string`* does not point to any disks the Oracle Database user has read/write access to, then Oracle ASM returns an error. If it points to one or more disks that have already been assigned to a different disk group, then Oracle Database returns an error unless you also specify `FORCE`.

For each valid candidate disk, Oracle ASM formats the disk header to indicate that it is a member of the new disk group.

> **See Also:**
>
> The `ASM_DISKSTRING` initialization parameter for more information on specifying the search string

NAME Clause

The `NAME` clause is valid only if the *`search_string`* points to a single disk. This clause lets you specify an operating system independent name for the disk. The name can be up to 30 characters long and can contain only alphanumeric characters. The first character must be alphabetic. If you omit this clause and you assigned a label to a disk through ASMLIB, then that label is used as the disk name. If you omit this clause and you did not assign a label through ASMLIB, then Oracle ASM creates a default name of the form *`diskgroupname_####`*, where *`####`* is the disk number. You use this name to refer to the disk in subsequent Oracle ASM operations.

SIZE Clause

Use this clause to specify in bytes the size of the disk. If you specify a size greater than the capacity of the disk, then Oracle ASM returns an error. If you specify a size less than the capacity of the disk, then you limit the disk space Oracle ASM will use. The size value must be identical for all disks in a disk group. If you omit this clause, then Oracle ASM attempts programmatically to determine the size of the disk.

FORCE

Specify `FORCE` if you want Oracle ASM to add the disk to the disk group even if the disk is already a member of a different disk group.

> **Note:**
>
> Using `FORCE` in this way may destroy existing disk groups.

For this clause to be valid, the disk must already be a member of a disk group and the disk cannot be part of a mounted disk group.

NOFORCE

Specify `NOFORCE` if you want Oracle ASM to return an error if the disk is already a member of a different disk group. `NOFORCE` is the default.

ATTRIBUTE Clause

Use this clause to set attribute values for the disk group. You can view the current attribute values by querying the `V$ASM_ATTRIBUTE` view. Table 13-2 lists the attributes you can set with this clause. All attribute values are strings.

Table 13-2 Disk Group Attributes

Attribute	Valid Values	Description
`ACCESS_CONTROL.ENABLED`	`true` or `false`	Specifies whether Oracle ASM File Access Control is enabled for a disk group. If set to `true`, accessing Oracle ASM files is subject to access control. If `false`, any user can access every file in the disk group. All other operations behave independently of this attribute. The default value is `false`. If both the `compatible.rdbms` and `compatible.asm` attributes are set to at least 11.2, you can set this attribute in an `ALTER DISKGROUP ... SET ATTRIBUTE` statement. You cannot set this attribute when creating a disk group. When you set up file access control on an existing disk group, the files previously created remain accessible by everyone, unless you run the `ALTER DISKGROUP SET PERMISSION` statement to restrict the permissions. **Note**: This attribute is used in conjunction with `ACCESS_CONTROL.UMASK` to manage Oracle ASM File Access Control. After setting the `ACCESS_CONTROL.ENABLED` disk attribute, you must set permissions with the `ACCESS_CONTROL.UMASK` attribute.

Table 13-2 (Cont.) Disk Group Attributes

Attribute	Valid Values	Description
ACCESS_CONTROL.UMASK	A three-digit number where each digit is 0, 2, or 6.	Determines which permissions are masked out on the creation of an Oracle ASM file for the user that owns the file (first digit), users in the same user group (second digit), and others not in the user group (third digit). This attribute applies to all files on a disk group. Setting to 0 masks out nothing. Setting to 2 masks out write permission. Setting to 6 masks out both read and write permissions. The default value is 066. If both the compatible.rdbms and compatible.asm attributes are set to at least 11.2, you can set this attribute in an ALTER DISKGROUP ... SET ATTRIBUTE statement. You cannot set this attribute when creating a disk group. When you set up file access control on an existing disk group, the files previously created remain accessible by everyone, unless you run the ALTER DISKGROUP SET PERMISSION statement to restrict the permissions. **Note**: This attribute is used in conjunction with ACCESS_CONTROL.ENABLED to manage Oracle ASM File Access Control. Before setting ACCESS_CONTROL.UMASK, you must set ACCESS_CONTROL.ENABLED to true.
AU_SIZE	Size in bytes. Valid values are powers of 2 from 1M to 64M. Examples '4M', '4194304'.	Specifies the allocation unit size. This attribute can be set only during disk group creation; it cannot be modified with an ALTER DISKGROUP statement.
COMPATIBLE.ADVM	Valid Oracle Database version number[1]	Determines whether the disk group can contain Oracle ADVM volumes. The value must be set to 11.2 or higher. Before setting this attribute, the COMPATIBLE.ASM value must be 11.2 or higher. Also, the Oracle ADVM volume drivers must be loaded. By default, the value of the COMPATIBLE.ADVM attribute is empty until set.
COMPATIBLE.ASM	Valid Oracle Database version number[1]	Determines the minimum software version for an Oracle ASM instance that can use the disk group. This setting also affects the format of the data structures for the Oracle ASM metadata on the disk. For Oracle ASM in Oracle Database 11*g*, 10.1 is the default setting for the COMPATIBLE.ASM attribute when using the SQL CREATE DISKGROUP statement, the ASMCMD mkdg command, and Oracle Enterprise Manager **Create Disk Group** page. When creating a disk group with ASMCA, the default setting is 11.2.

Table 13-2 (Cont.) Disk Group Attributes

Attribute	Valid Values	Description
COMPATIBLE.RDBMS	Valid Oracle Database version number[1]	Determines the minimum COMPATIBLE database initialization parameter setting for any database instance that is allowed to use the disk group. Before advancing the COMPATIBLE.RDBMS attribute, ensure that the values for the COMPATIBLE initialization parameter for all of the databases that access the disk group are set to at least the value of the new setting for COMPATIBLE.RDBMS. For example, if the COMPATIBLE initialization parameters of the databases are set to either 11.1 or 11.2, then COMPATIBLE.RDBMS can be set to any value between 10.1 and 11.1 inclusively. For Oracle ASM in Oracle Database 11*g*, 10.1 is the default setting for the COMPATIBLE.RDBMS attribute when using the SQL CREATE DISKGROUP statement, the ASMCMD mkdg command, ASMCA **Create Disk Group** page, and Oracle Enterprise Manager **Create Disk Group** page.
CONTENT.CHECK	true or false	Enables (true) or disables (false) content checking when performing data copy operations for rebalancing a disk group. You cannot set this attribute when creating a disk group. The default value is dependent on the COMPATIBLE.ASM attribute and follows this rule: • If COMPATIBLE.ASM > = 19.0.0.0.0, then CONTENT.CHECK defaults to true. • If COMPATIBLE.ASM < 19.0.0.0.0, then CONTENT.CHECK defaults to false. **Note**: This rule is ONLY true for the creation of new diskgroups. If the COMPATIBLE.ASM attribute of an existing diskgroup is updated to 19.0.0.0.0 or above, the CONTENT.CHECK attribute remains at its current value.
DISK_REPAIR_TIME	0 to 136 years	When disks are taken offline, Oracle ASM drops them after a default period of time. If both the compatible.rdbms and compatible.asm attributes are set to at least 11.1, you can set the disk_repair_time attribute in an ALTER DISKGROUP ... SET ATTRIBUTE statement to change that default period of time so that the disk can be repaired and brought back online. You cannot set this attribute when creating a disk group. The time can be specified in units of minute (M) or hour (H). The specified time elapses only when the disk group is mounted. If you omit the unit, then the default is H. If you omit this attribute, and both compatible.rdbms and compatible.asm are set to at least 11.1, then the default is 12 H. Otherwise the disk is dropped immediately. You can override this attribute with an ALTER DISKGROUP ... OFFLINE DISK statement and the DROP AFTER clause. **Note**: If a disk is taken offline using the current value of disk_repair_time, and the value of this attribute is subsequently changed, then the changed value is used by Oracle ASM in the disk offline logic. **See Also**: The ALTER DISKGROUP ... *disk_offline_clause* and *Oracle Automatic Storage Management Administrator's Guide* for more information

Table 13-2 (Cont.) Disk Group Attributes

Attribute	Valid Values	Description
FAILGROUP_REPAIR_TIME	<number>m (number of minutes) or <number>h (number of hours)	Specifies a default repair time for the failure groups in the disk group. The failure group repair time is used if Oracle ASM determines that an entire failure group has failed. The default value is 24 hours (24h). If there is a repair time specified for a disk, such as with the DROP AFTER clause of the ALTER DISKGROUP OFFLINE DISK statement, then that disk repair time overrides the failure group repair time. This attribute can only be set when altering a disk group and is only applicable to normal and high redundancy disk groups.
LOGICAL_SECTOR_SIZE	512, 4096, or 4K	Sets the logical sector size of a disk group. This value specifies the smallest possible I/O that the disk group can accept. The default value is estimated from the disks that join the disk group. To set this disk group attribute during the creation of a disk group or to alter it after a disk group has been created, the COMPATIBLE.ASM disk group attribute must be set to 12.2 or higher.
PHYS_META_REPLICATED	true or false	Tracks the replication status of a disk group. When the Oracle ASM compatibility of a disk group is advanced to 12.0 or higher, the physical metadata of each disk, including its disk header, free space table blocks and allocation table blocks, is replicated. The replication is performed online asynchronously. PHYS_META_REPLICATED is set to true by Oracle ASM if the physical metadata of every disk in the disk group has been replicated. This disk group attribute is only defined in a disk group with the Oracle ASM disk group compatibility (COMPATIBLE.ASM) set to 12.0 and higher. This attribute is read-only and is intended for information only. You cannot set or change its value.
PREFERRED_READ.ENABLED	true or false	In an Oracle extended cluster, which contains nodes that span multiple physically separated sites, the PREFERRED_READ.ENABLED disk group attribute controls whether preferred read functionality is enabled for a disk group. If preferred read functionality is enabled, then this functionality enables an instance to determine and read from disks at the same site as itself, which can improve performance. For extended clusters, the default value is true. For clusters that are not extended (only one physical site), preferred read is disabled (false). Preferred read status applies to extended, normal, high, and flex redundancy disk groups. This disk group attribute is only defined in a disk group with the Oracle ASM disk group compatibility (COMPATIBLE.ASM) set to 12.2 and higher.
SECTOR_SIZE	512, 4096, or 4K	Sets the physical sector size of a disk group. All disks in the disk group must have this physical sector size. The default value is obtained from the disks that join the disk group. To set this disk group attribute during the creation of a disk group, the COMPATIBLE.ASM and COMPATIBLE.RDBMS disk group attributes must be set to 11.2 or higher. To alter this disk group attribute after a disk group has been created, the COMPATIBLE.ASM disk group attribute must be set to 12.2 or higher.
THIN_PROVISIONED	true or false	Enables (true) or disables (false) the functionality to discard unused storage space after a disk group rebalance is completed. The default value is false.

Table 13-2 (Cont.) Disk Group Attributes

Attribute	Valid Values	Description
CONTENT_HARDCHECK	true or false	CONTENT_HARDCHECK enables or disables Hardware Assisted Resilient Data (HARD) checking when performing data copy operations for rebalancing a disk group. This attribute can only be set when altering a disk group.

[1] Specify at least the first two digits of a valid Oracle Database release number. Refer to *Oracle Database Administrator's Guide* for information on specifying valid version numbers. For example, you can specify compatibility as '11.2' or '12.1'.

> **See Also:**
>
> *Oracle Automatic Storage Management Administrator's Guide* for more information on managing these attribute settings

Examples

The following example assumes that the ASM_DISKSTRING parameter is a superset of /devices/disks/c*, /devices/disks/c* points to at least one device to be used as an Oracle ASM disk, and the Oracle Database user has read/write permission to the disks.

> **See Also:**
>
> *Oracle Automatic Storage Management Administrator's Guide* for information on Oracle ASM and using disk groups to simplify database administration

Creating a Diskgroup: Example

The following statement creates an Oracle ASM disk group dgroup_01 where no redundancy for the disk group is provided by Oracle ASM and includes all disks that match the *search_string*:

```
CREATE DISKGROUP dgroup_01
  EXTERNAL REDUNDANCY
  DISK '/devices/disks/c*';
```

CREATE DOMAIN

Purpose

Use CREATE DOMAIN to create a data use case domain. A use case domain is high-level dictionary object that belongs to a schema and encapsulates a set of optional properties and constraints.

You can define table columns to be associated with a domain, thereby explicitly applying the domain's optional properties and constraints to the columns.

At minimum, a domain must specify a built-in Oracle data type. The qualified domain name should not collide with the qualified user-defined data types, or with Oracle built-in data types.

The domain data type must be a single Oracle data type. For Oracle character data type you must specify a maximum length, one of `VARCHAR2(L [CHAR|BYTE])`, `NVARCHAR2(L)`, `CHAR(L [CHAR|BYTE])`, or `NCHAR(L)`.

Domain-Specific Expressions and Conditions

A domain expression can be one of *simple*, *datetime*, *interval*, *CASE*, *compound*, or *list domain expression* :

- A *simple domain expression* is one of `string`, `number`, `sequence.CURRVAL`, `sequence.NEXTVAL`, `NULL`, or `schema.domain`. It is similar to simple expressions, except that it references domain names instead of column names. It references domain names as qualified names, names of Oracle built-in domains or uses a `PUBLIC` synonym to a domain.
- A *datetime domain expression* is a datetime expression that references domain expressions only.
- An *interval domain expression* is defined just as a regular interval expression, except that it references domain expressions. For example, `(SYSTIMESTAMP - day_of_week) DAY(9) TO SECOND` is an interval domain expression.
- A *compound domain expression* is any of: `(expr)`, `expr op expr` with op `+`, `-`, `*`, `/`, `||`, or `expr COLLATE collation_name`, where `expr` is a domain expression.

  Examples of valid compound domain expressions

  ```
  'email: ' || EmailAddress

  day_of_week + INTERVAL '1' DAY

  TO_CHAR(LastFour(SSN))
  ```

- A *case domain expression* is a like a regular case expression except that it references domain expressions only.

  Examples of valid case domain expressions

  ```
  CASE
    WHEN UPPER(DOMAIN_DISPLAY(day_of_week)) IN ('SAT','SUN')
    THEN 'weekend'
    ELSE 'week day'
  END
  ```

- Similar to the definition of use case domain expressions, a domain condition is like a regular SQL condition, except that it references only domain expressions. You can use the keyword `VALUE` in domain expressions instead of the domain name. For example:

  ```
  CREATE DOMAIN day_of_week AS CHAR(3 CHAR)
     CONSTRAINT day_of_week_c
       CHECK (UPPER(VALUE) IN ('MON', 'TUE', 'WED', 'THU', 'FRI', 'SAT', 'SUN'))
     DEFERRABLE INITIALLY DEFERRED
     DISPLAY SUBSTR(VALUE, 1, 2);
  ```

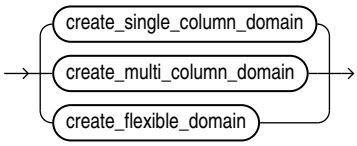
Prerequisites

To create a domain in your own schema, you must have the `CREATE DOMAIN` system privilege.

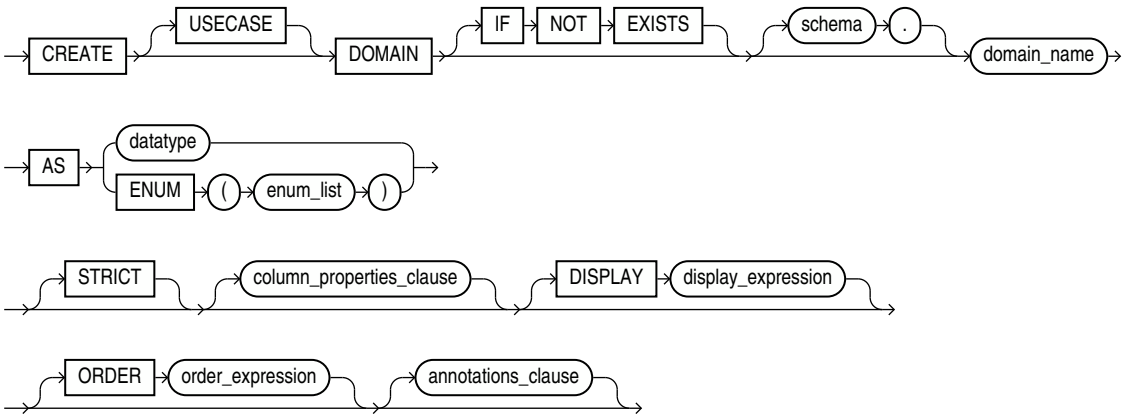
To create a domain in another user's schema, you must have the `CREATE ANY DOMAIN` system privilege.

Syntax

***create_domain*::=**

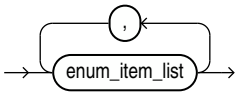


***create_single_column_domain*::=**



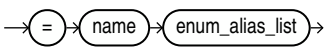
(*datatype*::=)

***enum_list*::=**

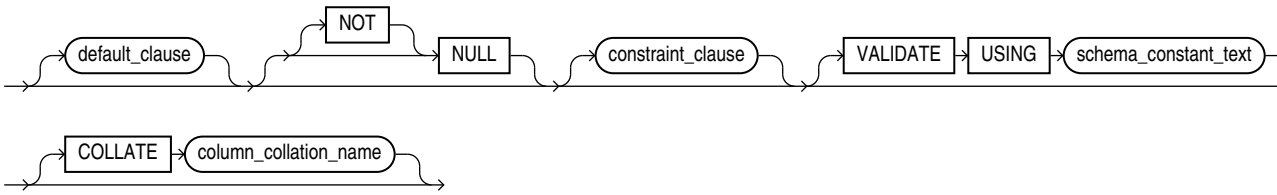


***enum_item_list*::=**

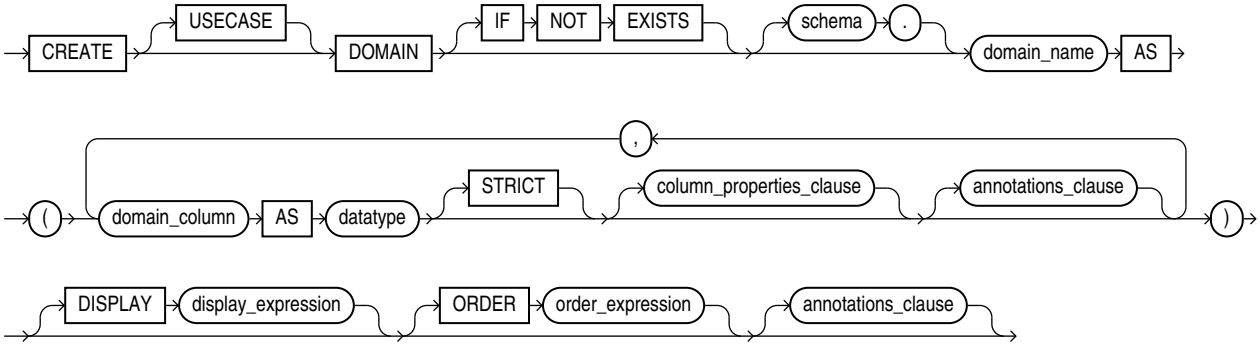
***enum_alias_list*::=**



***column_properties_clause*::=**

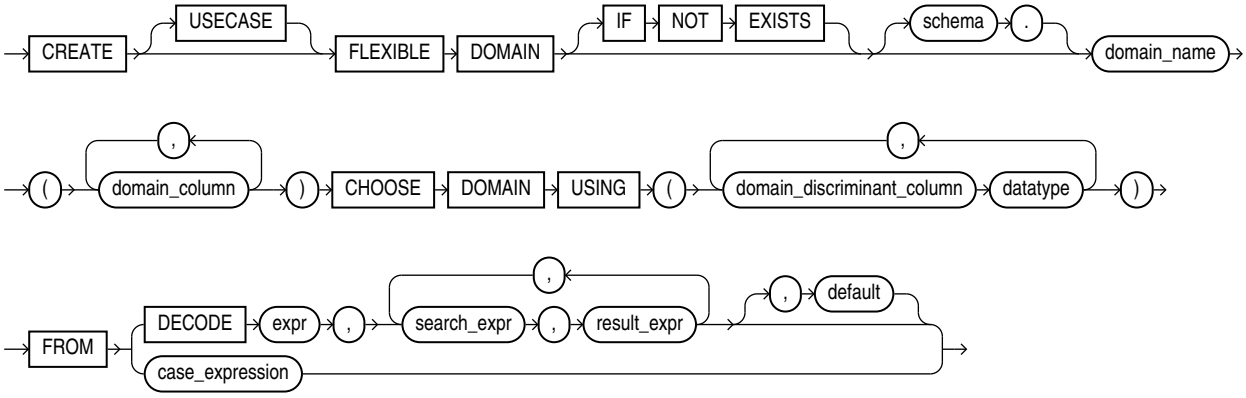


***create_multi_column_domain*::=**



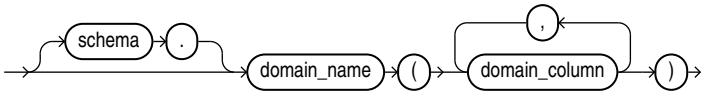
(*datatype*::=)

***create_flexible_domain*::=**

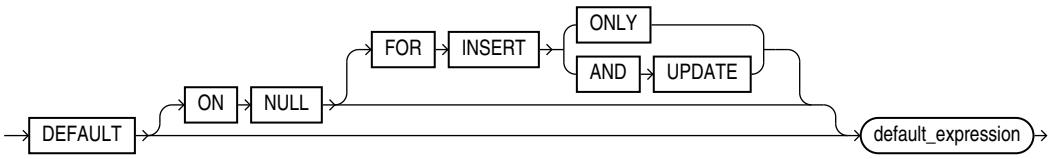


datatype::=,CASE Expressions

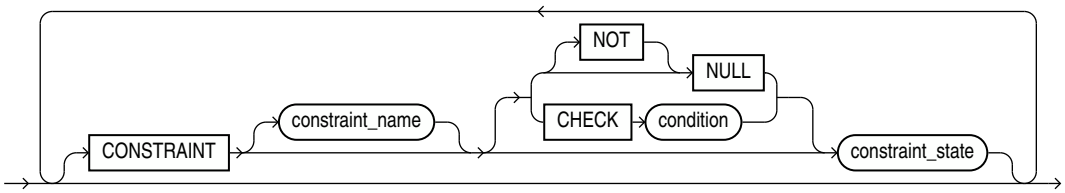
***result_expr*::=**



***default_clause*::=**



***constraint_clause*::=**



***annotations_clause*::=**

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

Semantics

USECASE

This keyword is optional and is provided for semantic clarity. It indicates that the domain is to describe a data use case.

IF NOT EXISTS

Specifying `IF NOT EXISTS` has the following effects:

- If the domain does not exist, a new domain is created at the end of the statement.
- If the domain exists, 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.`

domain_name

domain_name follows the same restrictions as any type name and must not collide with the name of any object in the domain schema, any Oracle supplied data type, and any Oracle supplied domain.

These restrictions apply at a PDB-level in a CDB environment.

Note that domains are schema-level catalog objects, and therefore subject to schema-level object restrictions.

datatype

datatype must be an Oracle built-in data type like:

- `CHAR(L [CHAR|BYTE])`, `NCHAR(L)`, `VARCHAR(L [CHAR|BYTE])`, `VARCHAR2(L [CHAR|BYTE])`, `NVARCHAR2(L)`
- `NUMBER[p, [s]]`, `FLOAT`, `BINARY_FLOAT`, `BINARY_DOUBLE`
- `RAW`, `LONG RAW` (extended included)
- `DATE`, `TIMESTAMP (WITH (LOCAL) TIME ZONE)`, `INTERVAL`
- `BFILE`, `BLOB`, `CLOB`, `NCLOB`
- `JSON` native data type
- `BOOLEAN`

ENUM

Specify `ENUM` to create an enumeration domain. An enumeration domain consists of a set of names and, optionally, a value corresponding to name. The name has to be a valid SQL identifier and every specified value must be a literal or a constant expression. The values can be of any data type that are supported for a data use case domain, but all of them must agree on that data type.

An enumeration domain has a default check-constraint, a display-expression, and an order-expression. This check-constraint cannot be dropped using `ALTER DOMAIN`.

The names inside a enumeration domain can be used wherever a literal is allowed in a scalar SQL expression, and the domain itself can be used in the `FROM` clause of a `SELECT` statement as if it were a table.

The collection of names (*`name`*) in an enumeration domain must be unique. There is no limit on number of names (or their aliases) in a enumeration, besides whatever limits already exist in Oracle SQL.

The data type of all the values (*`value`*) must match. If you do not specify a value, the default value of the first name is 1, and the value of every other unspecified name is one more than the previous value.

The expression defining the default must be one of the enumeration names without any expression besides that.

STRICT

When you specify `STRICT`, table columns linked with the domain must have the same data type limits as the corresponding domain columns. For example, if the data type of a domain column is `NUMBER(10)`, you can only associate it with columns declared as `NUMBER(10)`. Applying the domain to columns of `NUMBER(9)` or `NUMBER(11)` will raise a type error.

If you omit `STRICT`, you can link the domain to columns with type limits greater than or equal to the domain's limit. For example, you can apply a non-strict domain with the data type `NUMBER(10)` to columns with the data type `NUMBER(20)`.

If you associate a column with a domain without specifying the column's data type, then it uses `STRICT` semantics.

default_clause

If you specify the `ON NULL` clause, an implicit `NOT NULL` constraint is added.

default_expression

default_expression must be a domain expression and must conform to all the restrictions on default column expressions of the given data type, when applied to domain expressions:

- *default_expression* cannot contain a SQL function that returns a domain reference, or a nested function invocation, and it cannot be a subquery expression.
- The datatype of *default_expression* must match the domain's specified data type.
- As a domain expression, *default_expression* cannot refer to any table or column. It cannot refer to any other domain name.
- *default_expression* may refer to NEXTVAL and CURRVAL for a sequence. It cannot reference a PL/SQL function.

constraint_clause

Note that domain constraints can have optional names. They are NOT NULL, NULL or CHECK constraints. Multiple such constraint clauses can be specified both at the column and domain level.

The CHECK conditions as well as expressions in ALTER DOMAIN can only refer to domain columns. If the domain has a single column, the column name is either the domain name or the keyword VALUE, but the same expression cannot contain both domain name and VALUE as column name.

constraint_name is optional. When specified, it must not collide with a name of any other constraint in the domain's schema (in the given PDB if in a CDB environment). When not specified, a system-generated name will be used. Domain constraints follow the same rules as table and column-level constraints: a named table or column-level constraint cannot coincide with the name of any other constraint in the same schema. Domain constraints can share names with tables, even in the same schema. They can share names with columns, and it is possible for a constraint to have the same name as the table or column it is defined on.

The CHECK condition must be a domain logical condition and must conform to all the restrictions on CHECK constraints translated to domain expressions:

- It can only refer to *domain_name*, like a CHECK constraint on a column can only refer to a column. It cannot refer to any columns in any table or view, even within the domain schema.
- Subquery or scalar query expressions cannot be used.
- Condition cannot refer to non-deterministic functions (like CURRENT_DATE), or user-defined PL/SQL functions.
- CHECK IS JSON (STRICT) constraints are allowed.

  CHECK IS JSON(VALIDATE USING schema_constant_text) is allowed. You can specify the JSON schema and use it to validate that the JSON column respects the schema definition that is specified. *schema_constant_text* can be a constant literal with the JSON schema text, or a bind variable for the JSON schema text. The bind must be a runtime constant. It cannot be a domain.

  If you use the IS JSON constraint without specifying VALIDATE USING schema_constraint, any JSON value will be accepted. But when you specify a JSON Schema with VALIDATE USING schema_constraint , and the entered input data into the table column does not follow the schema, a JSON schema validation error is raised.

You can use shorthand syntax to specify the JSON schema with `VALIDATE USING schema_constant_text`.

Just as for table and column-level constraints, you can specify only one JSON constraint for a given table column

- The `CHECK` constraint condition is applied to one value at a time, and it is satisfied if the `CHECK` condition, with *`domain_name`* substituted by the value, evaluates to `TRUE` or `UNKNOWN`.

Domain constraints may be enforced in any order.

NULL constraint means that values of the domain are allowed to be NULL, and this is the default.

When *`constraint_state`* is not specified, the constraint is `NOT DEFERABLE INITIALLY IMMEDIATE`.

COLLATE

When collation is specified, it conforms to all the restrictions of column-level or schema-level collation. The data type must be a character data type if collation is specified.

You must ensure that all columns of a domain with a collation specified have the same collation as that of their domain.

If no collation is specified, and the data type is collatable, then the column's collation is used if specified. Otherwise the underlying default data type collation in the domain's schema is used.

An error is raised in the following cases:

- If you create a table with a column in a domain with a collation different than the column's collation.
- If you alter a column to have a collation different than the collation of the column's domain.
- If you alter a domain to add or modify the domain's collation to a value different than the collation of any column marked of the given domain.

You can specify the `COLLATE` 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`.

display_expression

Use *`display_expression`* to format the data according to domain specifications. It can be of any data type allowed as a domain data type. *`display_expression`* must be a domain expression that does not contain table or view columns, subqueries, non-deterministic functions, or PL/SQL functions. It can refer to *`domain_name`*. If you do not specify collation for the expression, then *`display_expression`* uses the domain's collation, if it is specified.

order_expression

Use *`order_expression`* to order and compare values for domain specifications.

`order_expression` must conform to the same restrictions as display_expressions, and additionally must be of a byte or char-comparable data type. It returns *`order_expression`* with *`domain_name`* replaced by expression, if *`order_expression`* is specified for the expression's domain, or expression otherwise

annotations_clause

For examples of the *`annotations_clause`* see the examples at the end.

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

FROM Clause of Create Flexible Domain

expr and *comparison_expr* reference domain discriminant columns in the list *domain_discriminant_column*.

The `FROM` clause for flexible domain is either a `DECODE` or a `CASE` expression that refers only to discriminant column names (in the list following `CHOOSE DOMAIN USING`) in the search expressions and has only domain name followed by a list of columns in the result expressions. The columns in the result expression must be only columns in the domain column list (following `CREATE FLEXIBLE DOMAIN`).

Examples

Create Domain year_of_birth

The following example creates the single column domain `year_of_birth`. The check constraint ensures that the column's value is an integer with a value greater than or equal to 1900. The display clause formats the output of calls to `domain_display` to either `19-YY` or `20-YY`, where `YY` is the last two digits of the value. The order clause sorts calls to `domain_order` in order by the column value minus 1900.

```
CREATE DOMAIN year_of_birth AS NUMBER(4)
      CONSTRAINT CHECK ( (trunc(year_of_birth) = year_of_birth) and (year_of_birth >=
1900) )
      DISPLAY (CASE WHEN year_of_birth < 2000 THEN '19-' ELSE '20-' END) ||
MOD(year_of_birth, 100)
      ORDER year_of_birth-1900 ;
```

Create Domain day_of_week

The following statement creates the single column domain `day_of_week`. The check constraint ensures that its values are one of `MON`, `TUE`, `WED`, `THU`, `FRI`, `SAT`, `SUN`. The initially deferred clause delays validation of these values until commit time. The order clause ensures the values are sorted by day of week instead of alphabetically when calling `domain_order`.

```
CREATE DOMAIN day_of_week AS CHAR(3 CHAR)
   CONSTRAINT day_of_week_c
     CHECK (day_of_week IN ('MON', 'TUE', 'WED', 'THU', 'FRI', 'SAT', 'SUN'))
   INITIALLY DEFERRED
   ORDER CASE day_of_week
      WHEN 'MON' THEN 0
      WHEN 'TUE' THEN 1
      WHEN 'WED' THEN 2
      WHEN 'THU' THEN 3
      WHEN 'FRI' THEN 4
      WHEN 'SAT' THEN 5
      WHEN 'SUN' THEN 6
      ELSE 7
  END;
```

From 23.3 you can associate columns of data type `CHAR(L CHAR)` with the domain with any value for `L`.

Create Domain email

The following example creates the sequence `email_seq`. It then creates the single column domain `email`. This uses the sequence to generate email addresses in the form

nnn@domain.com when you insert null into columns with this domain, where nnn are the numbers generated by the sequence. The constrains ensures that email addresses are of the form sss@sss.sss, where sss is any nonwhitespace character.

The display clause formats the output of calls to domain_display to ---sss.sss, where sss is the nonwhitespace characters after the @ sign.

```
CREATE SEQUENCE IF NOT EXISTS email_seq;

CREATE DOMAIN email AS VARCHAR2(30)
    DEFAULT ON NULL email_seq.NEXTVAL || '@domain.com'
    CONSTRAINT EMAIL_C CHECK (REGEXP_LIKE (email, '^(\S+)\@(\S+)\.(\S+)$'))
    DISPLAY '---' || SUBSTR(email, INSTR(email, '@') + 1);
```

Create a Strict Domain dept_codes

The following statement creates the domain dept_codes. The check constraint ensures its values are greater than 99 excluding 200. It adds the annotation Title with the value "Domain Annotation". You can only link this domain with columns of type NUMBER(3).

```
CREATE DOMAIN dept_codes AS NUMBER(3) STRICT
      CONSTRAINT dept_chk CHECK (dept_codes > 99 AND dept_codes != 200)
      ANNOTATIONS (Title 'Domain Annotation');
```

Create Domain hourly_wages

The following statement creates the single column domain hourly_wages. It defaults to 15 when inserting null into columns with this domain. The check constraint ensures the values are between 7 and 1,000.

The display clause returns its value in the format $999.99 when calling domain_display. The order clause multiplies its value by negative one, so sorting by domain_order sorts from high to low. It has the annotation Title with the value "Domain Annotation".

```
CREATE DOMAIN hourly_wages AS NUMBER(10)
       DEFAULT ON NULL 15
       CONSTRAINT minimal_wage_c
         CHECK (hourly_wages >= 7 and hourly_wages <=1000) ENABLE
       DISPLAY TO_CHAR(hourly_wages, '$999.99')
       ORDER ( -1*hourly_wages )
       ANNOTATIONS (Title 'Domain Annotation');
```

Add Annotations to a Multi Column Domain US_City at Column and Domain Levels

The following statement creates the multicolumn domain US_city. This has three columns: name, state, and zip. All columns have the Address annotation.

The check constraint ensures that the permitted values for state are CA, AZ, and TX and zip is less than 100000. The display clause returns calls to domain_display in the format name ||', '|| state ||', '||TO_CHAR(zip).

The order clause sorts calls to domain_order by the concatenation of state, then zip, then name. The domain has an object level annotation Title with the value "Domain Annotation" and and three column level annotations Address without a value, one for each column.

```
CREATE DOMAIN US_city AS
  (
    name  AS VARCHAR2(30) ANNOTATIONS (Address),
    state AS VARCHAR2(2) ANNOTATIONS (Address),
```

```
    zip AS NUMBER ANNOTATIONS (Address)
  )
  CONSTRAINT City_CK CHECK(state in ('CA','AZ','TX') and zip < 100000)
  DISPLAY name||', '|| state ||', '||TO_CHAR(zip)
  ORDER state||', '||TO_CHAR(zip)||', '||name
  ANNOTATIONS (Title 'Domain Annotation');
```

Create a Flexible Domain

The following examples create the flexible domain `expense_details`. To do this, you must first create the domains `flight_details`, `meals_details`, and `lodging_details`. These are multicolumn domains with check constraints to ensure the domain columns store appropriate values for the expense type.

For `flight_details`, this means the `flight_num` are two strings separated by a dash, and the origin and destination are both three character strings.

For `meals_details`, the restaurant is mandatory, the `meal_type` one of `Breakfast`, `Lunch` or `Dinner`, and `diner_count` is non-null.

For `lodging_details`, the hotel must be non-null and the `nights_count` greater than zero.

The flexible domain `expense_details` then chooses between these based on the value in the typ column.

In the `FROM DECODE` example:

- if `typ` = `Flight`, it uses the `flight_details` domain. The flexible domain columns `val1`, `val2`, and `val3` map to `flight_num`, `origin`, and `destination` in the `flight_details` domain.
- if `typ` = `Meals`, it uses the `meals_details` domain. The flexible domain columns `val1`, `val2`, and `val4` map to `restaurant`, `meal_type`, and `diner_count` respectively in the `meals_details` domain.
- if `typ` = `Lodging`, it uses the `lodging_details` domain. The flexible domain columns `val1` and `val4` map to `hotel` and `nights_count` respectively in the `lodging_details` domain.

In the `FROM CASE` flexible domain:

- If `typ` starts with letters `A-G`, it uses the `flight_details` domain.
- If `typ` = `Meals`, it uses the `meals_details` domain.
- If typ starts with `Lodg`, it uses the `lodging_details` domain.

The column mappings are the same as in the `FROM DECODE` example.

Create Domain flight_details

```
CREATE DOMAIN flight_details AS
  (
   flight_num AS VARCHAR2(100) NOT NULL,
   origin AS VARCHAR2(200)
     CONSTRAINT origin_3_char_c CHECK (LENGTH(origin) = 3),
   destination AS VARCHAR2(200)
     CONSTRAINT dest_3_char_c CHECK (LENGTH(destination) = 3)
  )
  CONSTRAINT flight_c
    CHECK
      (
        flight_num LIKE '%-%' AND
        origin IS NOT NULL AND
```

```
        destination IS NOT NULL
      )
  CONSTRAINT origin_dest_different_c
    CHECK (origin <> destination)
  DISPLAY flight_num||', '||origin||', '||destination
  ORDER flight_num||destination;
```

Create Domain meals_details

```
CREATE DOMAIN meals_details AS
    (
       restaurant AS VARCHAR2(100) NOT NULL,
       meal_type AS VARCHAR2(200),
       diner_count AS NUMBER
    )
    CONSTRAINT meals_c
      CHECK
       (
         restaurant IS NOT NULL AND
         meal_type IN ('Breakfast', 'Lunch', 'Dinner') AND
         diner_count IS NOT NULL
       )
    DISPLAY meal_type||', '||restaurant||', '||diner_count;
```

Create Domain lodging_details

```
 CREATE DOMAIN lodging_details AS
    (
      hotel AS VARCHAR2(100) NOT NULL,
      nights_count AS NUMBER
    )
    CONSTRAINT lodging_c
     CHECK (hotel IS NOT NULL AND nights_count > 0)
    DISPLAY hotel||', '||nights_count;
```

Create a Flexible Domain expense_details Using FROM DECODE

```
CREATE FLEXIBLE DOMAIN expense_details (val1, val2, val3, val4)
    CHOOSE DOMAIN USING (typ VARCHAR2(10))
    FROM DECODE(typ,
              'Flight', flight_details(val1, val2, val3),
              'Meals', meals_details(val1, val2, val4),
              'Lodging', lodging_details(val1, val4));
```

Create a Flexible Domain expense_details Using FROM CASE

```
CREATE FLEXIBLE DOMAIN expense_details (val1, val2, val3, val4)
    CHOOSE DOMAIN USING(typ VARCHAR2(10))
    FROM CASE
        WHEN typ BETWEEN 'A' AND 'G' THEN flight_details(val1, val2, val3)
        WHEN typ = 'Meals' THEN meals_details(val1, val2, val4)
        WHEN typ LIKE 'Lodg%' THEN lodging_details(val1, val4)
      END;
```

Create Domain Specifying a JSON Schema for Validation

The following example creates domain `w2_form` of type `JSON` with a constraint that checks that the value is a JSON object that complies with the provided JSON Schema. The check constraint uses `IS JSON VALIDATE USING` *`schema_constant_text`*:

```
CREATE DOMAIN w2_form AS JSON
  CONSTRAINT CHECK (VALUE IS JSON VALIDATE USING '{
```

```
    "title": "W2_form",
    "type": "object",
    "properties": {
    "social_security_number": {
    "type": "string",
    "description": "The person social security number."
   },
   "wages": {
     "description": "total wages",
     "type": "number",
     "minimum": 0
   },
   "social_security_wages": {
     "type": "number",
     "description": "wages subject to social security tax"
   },
   "Federal Income Tax Withheld": {
     "type": "number",
     "description": "withheld of tax to federal income tax"
   },
   "Social Security Tax Withheld": {
     "type": "number",
     "description": "withheld of social security tax"
   }
   },
   "required": [
     "social_security_number",
     "wages",
     "Federal Income Tax Withheld"
    ]
   }'
 );
```

The following statement creates table `tax_report` where column `w2_form` is associated with domain `w2_form` to ensure that its content conforms to the schema defined in the domain`w2_form`:

```
CREATE TABLE tax_report(id NUMBER, income JSON DOMAIN w2_form);
```

Before the data is inserted into the income column, it is checked against the JSON Schema. If the data is not valid, an error is raised.

Example of valid data:

```
INSERT INTO tax_report VALUES
  (1, '{"wages": 100, "social_security_number": "111", "Federal Income Tax Withheld":10}'
   );
1 row created
```

Example of invalid data:

```
INSERT INTO tax_report VALUES
  (2, '{"wages": 100}'
   );
ERROR at line 1:
ORA-40875: JSON schema validation error
```

Create a Domain Specifying a JSON Schema Using Shorthand Syntax

```
CREATE DOMAIN w2_form AS JSON VALIDATE USING '{
```

```
  "title": "W2_form",
  "type": "object",
  "properties": {
    "social_security_number": {
    "type": "string",
    "description": "The person social security number."
  },
  "wages": {
    "description": "total wages",
    "type": "number",
    "minimum": 0
  },
  "social_security_wages": {
    "type": "number",
    "description": "wages subject to social security tax"
  },
  "Federal Income Tax Withheld": {
    "type": "number",
    "description": "withheld of tax to federal income tax"
  },
  "Social Security Tax Withheld": {
    "type": "number",
    "description": "withheld of social security tax"
  }
 },
 "required": [
   "social_security_number",
   "wages",
   "Federal Income Tax Withheld"
  ]
 }';
```

Example: Create a Domain with an Annotation Stored as JSON and Query its Value

The following example creates a domain with an annotation `allowed_operations` specified as a JSON string which contains a nested JSON object:

```
CREATE DOMAIN email AS VARCHAR2(30)
    CONSTRAINT EMAIL_C CHECK (REGEXP_LIKE (email, '^(\S+)\@(\S+)\.(\S+)$'))
    DISPLAY '---' || SUBSTR(email, INSTR(email, '@') + 1)
    ANNOTATIONS(allowed_operations
'{
    "allowed_operations": {
        "title": "Allowed operations",
        "operations": [
            "Sort",
            "Group By",
            "Picklist"
        ]
    }
}');
```

Now you can run the following query to retrieve values for the annotation `allowed_operations`:

```
SELECT jt.* FROM user_annotations_usage a,
  JSON_TABLE (annotation_value,
    '$.allowed_operations.operations[*]'
    COLUMNS (value VARCHAR2(50 CHAR) PATH '$')) jt
  WHERE annotation_name = 'ALLOWED_OPERATIONS'
  AND object_name = 'EMAIL' ;
```

The output is:

```
VALUE
----------------------------------------
Sort
Group By
Picklist
```

Example: Create a Domain with an Annotation Stored as JSON and Query its Value

The following example creates a domain with the annotation `display_units` specified as a JSON string containg an array to store the possible display units:

```
CREATE DOMAIN temperature AS NUMBER(3)
ANNOTATIONS (display_units '{ "units": ["celsius", "fahrenheit"] }');
```

Now you can query for the values of the annotation:

```
SELECT jt.* FROM user_annotations_usage,
  JSON_TABLE(annotation_value, '$.units[*]'
    COLUMNS (value VARCHAR2(30 CHAR) PATH '$')) jt
  WHERE annotation_name = 'DISPLAY_UNITS'
  AND object_name = 'TEMPERATURE';
```

The output is:

```
VALUE
-------------------------------------------
celsius
fahrenheit
```

Example: JSON Schema Using a Use Case Domain

You can register a JSON schema as a use case domain.

The following example creates domain `dj5` as a JSON schema object:

```
CREATE DOMAIN dj5 AS JSON CONSTRAINT dj5chk
    CHECK (dj5 IS JSON validate
          '{
           "type": "object",
           "properties": {
             "a": {
               "type": "number"
              }
           }
         }'
    );
```

You can then create a table `jtab` and associate column `jcol` with the domain `dj5`:

```
CREATE TABLE jtab(
      id   NUMBER PRIMARY KEY,
     jcol JSON DOMAIN dj5
    );
```

Examples for ENUM Domains

Example: Create ENUM Domain order_status

The following example creates an enumeration domain `order_status` with a collection of names `New`, `Open`, `Shipped`, `Closed`, `Cancelled`:

```
CREATE DOMAIN order_status AS
  ENUM (
    New ,
    Open ,
    Shipped ,
    Closed ,
    Cancelled
  );
```

Example: Query Enumeration Domain order_status

Unlike a regular domain, the enumeration domain `order_status` can be treated as a table and queried via `SELECT` as follows:

```
SELECT * FROM order_status;
```

The result is:

```
ENUM_NAME     ENUM_VALUE
- - - - -    - - - - - -
NEW                    1
OPEN                   2
SHIPPED                3
CLOSED                 4
CANCELLED              5
```

Example: Enumeration Domain order_status as data type of Column:

Like a regular single-column domain, an enumeration domain can be used to define the data type of a column in a table. In the example below, the enumeration domain `order_status` is used as the data type of column `status` in table `orders`:

```
CREATE TABLE orders (
     id      NUMBER,
     cust    VARCHAR2(100),
     status  ORDER_STATUS
);
```

Using `DESCRIBE` on the `orders` table shows the `status` column as a numeric column with a single column domain `order_status`:

```
DESCRIBE orders;
```

The result is:

```
Name    Null ?  Type
----    ------  ----
ID              NUMBER
CUST            VARCHAR2 (100)
STATUS          NUMBER SCOTT.ORDER_STATUS
```

You can construct each row to insert into the the `orders` table using the appropriate `order_status`:

```
INSERT INTO orders VALUES
  (1, 'Costco', order_status.open ),
  (2, 'BMW', order_status.closed ),
  (3, 'Nestle', order_status.shipped );

 3 rows created .
```

Use the `domain_display` function to list the rows in the `orders` table:

```
SELECT ID, DOMAIN_DISPLAY(STATUS) FROM orders;
```

The result is:

```
ID        STATUS
---       ------
1         OPEN
2         CLOSED
3         SHIPPED
```

The actual values stored in the status column are the values you associated with the status when you created the enumerated domain `order_status`, 2 for `OPEN`, 4 for `CLOSED`, and 3 for `SHIPPED`:

```
SELECT ID, STATUS FROM orders;
```

The result is:

```
ID        STATUS
---       ------
1         2
2         4
3         3
```

Since the underlying data type of the status column is a number, you can directly update the column with a numeric value as long as is passes the domain check-constraint:

```
UPDATE orders SET STATUS = 2 WHERE STATUS = 5;

1 ROW UPDATED.
```

Since enumeration names are placeholds for literal values, you can use them anywhere where SQL allows literals:

```
SELECT 2 * ORDER_STATUS.CANCELLED;
```

The result is:

```
2*ORDER_STATUS.CANCELLED
------------------------
                      10
```

Example: Create ENUM Domain days_of_week

The following example creates an enumeration domain `days_of_week` with a collection of names that comprises the days of the week.

Each value has a pair of names, only the first name of the pair has an assigned value. There are two names for each value, the full day name and the first two letters of the day's name. The names `Sunday` and `Su` both have the value zero. The value then increases by one for each pair of names. So `Monday` and `Mo` have the value 1, `Tuesday` and `Tu` have the value 2 and so on up to `Saturday` and `Sa` which have the value 6.

```
CREATE DOMAIN days_of_week AS
  ENUM (
    Sunday     = Su  = 0,
    Monday     = Mo,
    Tuesday    = Tu,
    Wednesday  = We,
    Thursday   = Th,
    Friday     = Fr,
    Saturday   = Sa
  );
```

CREATE EDITION

Purpose

This statement creates a new edition as a child of an existing edition. An edition makes it possible to have two or more versions of the same editionable objects in the database. When you create an edition, it immediately inherits all of the editionable objects of its parent edition. The following object types are editionable:

- Synonym
- View
- Function
- Procedure
- Package (specification and body)
- Type (specification and body)
- Library
- Trigger

An editionable object is an object of one of the above editionable object types in an editions-enabled schema. The ability to have multiple versions of these objects in the database greatly facilitates online application upgrades.

> **Note:**
>
> All database object types not listed above are not editionable. Changes to object types that are not editionable are immediately visible across all editions in the database.

Every newly created or upgraded Oracle Database has one default edition named `ORA$BASE`, which serves as the parent of the first edition created with a `CREATE EDITION` statement. You can subsequently designate a user-defined edition as the database default edition using an `ALTER DATABASE DEFAULT EDITION` statement.

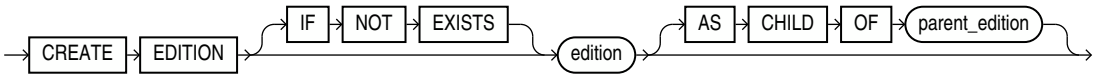
> **See Also:**
>
> - *Oracle Database Development Guide* for a more complete discussion of editionable object types and editions
> - The `ALTER DATABASE` "DEFAULT EDITION Clause" for information on designating an edition as the default edition for the database

Prerequisites

To create an edition, you must have the `CREATE ANY EDITION` system privilege, granted either directly or through a role. To create an edition as a child of another edition, you must have the `USE` object privilege on the parent edition.

Syntax

***create_edition*::=**



Semantics

edition

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

To view the editions that have been created for the database, query the `EDITION_NAME` column of the `DBA_OBJECTS` or `ALL_OBJECTS` data dictionary view.

When you create an edition, the system automatically grants you the `USE` object privilege `WITH GRANT OPTION` on the edition you create.

> **Note:**
>
> Oracle strongly recommends that you do not name editions with the prefixes `ORA`, `ORACLE`, `SYS`, `DBA`, and `DBMS`, as these prefixes are reserved for internal use.

IF NOT EXISTS

Specifying `IF NOT EXISTS` has the following effects:

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

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

AS CHILD OF Clause

If you use this clause, then the new edition is created as a child of *parent_edition*. If you omit this clause, then the new edition is created as a child of the leaf edition. At the time of its creation, the new edition inherits all editioned objects from its parent edition.

Restriction on Editions

An edition can have only one child edition. If you specify for *parent_edition* an edition that already has a child edition, then an error is returned.

Examples

The following very simple examples are intended to show the syntax for creating and working with an edition. For realistic examples of using editions refer to *Oracle Database Development Guide*.

In the following statements, the user HR is given the privileges needed to create and use an edition:

```
GRANT CREATE ANY EDITION, DROP ANY EDITION to HR;
Grant succeeded.

ALTER USER hr ENABLE EDITIONS;
User altered.
```

HR creates a new edition TEST_ED for testing purposes:

```
CREATE EDITION test_ed;
```

HR then creates an editioning view ed_view in the default edition ORA$BASE for testing purposes, first verifying that the current edition is the default edition:

```
SELECT SYS_CONTEXT('userenv', 'current_edition_name') FROM DUAL;
SYS_CONTEXT('USERENV','CURRENT_EDITION_NAME')
--------------------------------------------------------------------------------
ORA$BASE
1 row selected.

CREATE EDITIONING VIEW e_view AS
  SELECT last_name, first_name, email FROM employees;
View created.

DESCRIBE e_view
 Name                                    Null?    Type
 ----------------------------------------- -------- ----------------------------
 LAST_NAME                               NOT NULL VARCHAR2(25)
 FIRST_NAME                                       VARCHAR2(20)
 EMAIL                                   NOT NULL VARCHAR2(25)
```

The view is then actualized in the TEST_ED edition when HR uses the TEST_ED edition and re-creates the view in a different form:

```
ALTER SESSION SET EDITION = TEST_ED;
Session altered.

CREATE OR REPLACE EDITIONING VIEW e_view AS
  SELECT last_name, first_name, email, salary FROM employees;

View created.
```

The view in the TEST_ED edition has an additional column:

```
DESCRIBE e_view
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------------
 LAST_NAME                                 NOT NULL VARCHAR2(25)
 FIRST_NAME                                         VARCHAR2(20)
 EMAIL                                     NOT NULL VARCHAR2(25)
 SALARY                                             NUMBER(8,2)
```

The view in the ORA$BASE edition remains isolated from the test environment:

```
ALTER SESSION SET EDITION = ora$base;
Session altered.

DESCRIBE e_view;

 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------------
 LAST_NAME                                 NOT NULL VARCHAR2(25)
 FIRST_NAME                                         VARCHAR2(20)
 EMAIL                                     NOT NULL VARCHAR2(25)
```

Even if the view is dropped in the test environment, it remains in the ORA$BASE edition:

```
ALTER SESSION SET EDITION = TEST_ED;
Session altered.

DROP VIEW e_view;
View dropped.

ALTER SESSION SET EDITION = ORA$BASE;
Session altered.

DESCRIBE e_view;
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------------
 LAST_NAME                                 NOT NULL VARCHAR2(25)
 FIRST_NAME                                         VARCHAR2(20)
 EMAIL                                     NOT NULL VARCHAR2(25)
```

When the testing of upgrade that necessitated the TEST_ED edition is complete, the edition can be dropped:

```
DROP EDITION TEST_ED;
```

CREATE FLASHBACK ARCHIVE

Purpose

Use the CREATE FLASHBACK ARCHIVE statement to create a flashback archive, which provides the ability to automatically track and archive transactional data changes to specified database objects. A flashback archive consists of multiple tablespaces and stores historic data from all transactions against tracked tables. The data is stored in internal history tables.

Flashback data archives retain historical data for the time duration specified using the RETENTION parameter. Historical data can be queried using the Flashback Query AS OF clause. Archived historic data that has aged beyond the specified retention period is automatically purged.

Flashback data archives retain historical data across data definition language (DDL) changes to tables enabled for flashback archive. Flashback data archives supports many common DDL statements, including some DDL statements that alter table definitions or incur data movement. DDL statements that are not supported result in error ORA-55610.

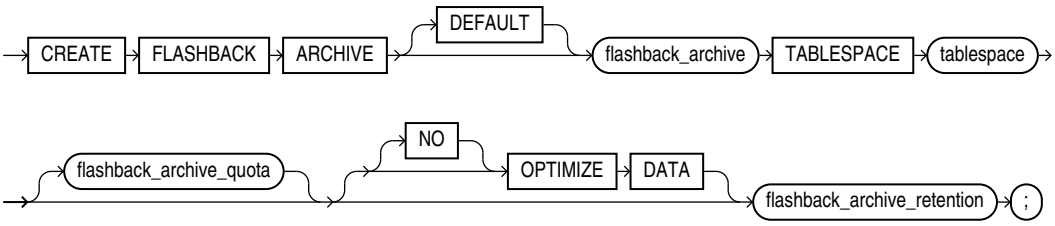
> **See Also:**
>
> - *Oracle Database Development Guide* for general information on using Flashback Time Travel
> - The `CREATE TABLE` *flashback_archive_clause* for information on designating a table as a tracked table
> - ALTER FLASHBACK ARCHIVE for information on changing the quota and retention attributes of the flashback archive, as well as adding or changing tablespace storage for the flashback archive

Prerequisites

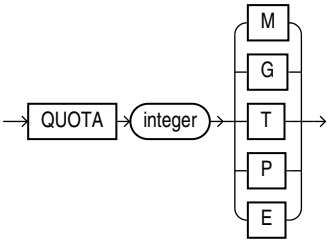
You must have the `FLASHBACK ARCHIVE ADMINISTER` system privilege to create a flashback archive. In addition, you must have the `CREATE TABLESPACE` system privilege to create a flashback archive, as well as sufficient quota on the tablespace in which the historical information will reside. To designate a flashback archive as the system default flashback archive, you must be logged in as `SYSDBA`.

Syntax

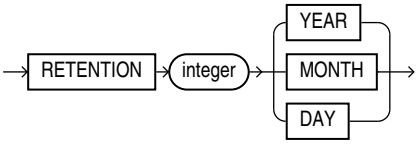
***create_flashback_archive*::=**



***flashback_archive_quota*::=**



***flashback_archive_retention*::=**



Semantics

DEFAULT

You must be logged in as `SYSDBA` to specify `DEFAULT`. Use this clause to designate this flashback archive as the default flashback archive for the database. When a `CREATE TABLE` or `ALTER TABLE` statement specifies the *`flashback_archive_clause`* without specifying a flashback archive name, the database uses the default flashback archive to store data from that table.

You cannot specify this clause if a default flashback archive already exists. However, you can replace an existing default flashback archive using the `ALTER FLASHBACK ARCHIVE ... SET DEFAULT` clause.

> **See Also:**
>
> The `CREATE TABLE` *flashback_archive_clause* for more information

flashback_archive

Specify the name of the flashback archive. The name must satisfy the requirements specified in "Database Object Naming Rules ".

TABLESPACE Clause

Specify the tablespace where the archived data for this flashback archive is to be stored. You can specify only one tablespace with this clause. However, you can subsequently add tablespaces to the flashback archive with an `ALTER FLASHBACK ARCHIVE` statement.

flashback_archive_quota

Specify the amount of space in the initial tablespace to be reserved for the archived data. If the space for archiving in a flashback archive becomes full, then DML operations on tracked tables that use this flashback archive will fail. The database issues an out-of-space alert when the content of the flashback archive is 90% of the specified quota, to allow time to purge old data or add additional quota. If you omit this clause, then the flashback archive has unlimited quota on the specified tablespace.

[NO] OPTIMIZE DATA

Specify `OPTIMIZE DATA` to enable optimization for flashback archive history tables. This instructs the database to optimize the storage of data in history tables using any of the following features: Advanced Row Compression, Advanced LOB Compression, Advanced LOB Deduplication, segment-level compression tiering, and row-level compression tiering. To specify this clause, you must have a license for the Advanced Compression option.

Specify `NO OPTIMIZE DATA` to instruct the database not to optimize the storage of data in history tables. This is the default.

flashback_archive_retention

Specify the length of time in months, days, or years that the archived data should be retained in the flashback archive. If the length of time causes the flashback archive to become full, then the database responds as described in *flashback_archive_quota*.

Examples

The following statement creates two flashback archives for testing purposes. The first is designated as the default for the database. For both of them, the space quota is 1 megabyte, and the archive retention is one day.

```
CREATE FLASHBACK ARCHIVE DEFAULT test_archive1
   TABLESPACE example
   QUOTA 1 M
   RETENTION 1 DAY;

CREATE FLASHBACK ARCHIVE test_archive2
   TABLESPACE example
   QUOTA 1 M
   RETENTION 1 DAY;
```

The next statement alters the default flashback archive to extend the retention period to 1 month:

```
ALTER FLASHBACK ARCHIVE test_archive1
   MODIFY RETENTION 1 MONTH;
```

The next statement specifies tracking for the `oe.customers` table. The flashback archive is not specified, so data will be archived in the default flashback archive, `test_archive1`:

```
ALTER TABLE oe.customers
   FLASHBACK ARCHIVE;
```

The next statement specifies tracking for the `oe.orders` table. In this case, data will be archived in the specified flashback archive, `test_archive2`:

```
ALTER TABLE oe.orders
   FLASHBACK ARCHIVE test_archive2;
```

The next statement drops `test_archive2` flashback archive:

```
DROP FLASHBACK ARCHIVE test_archive2;
```

CREATE FUNCTION

Purpose

Functions are defined in 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 FUNCTION` statement to create a standalone stored function or a call specification.

- A **stored function** (also called a **user function** or **user-defined function**) is a set of PL/SQL statements you can call by name. Stored functions are very similar to procedures,

except that a function returns a value to the environment in which it is called. User functions can be used as part of a SQL expression.

- A **call specification** declares a JavaScript method, a Java method or a third-generation language (3GL) routine so that it can be called from PL/SQL. You can also use the `CALL` SQL statement to call such a method or routine. The call specification tells Oracle Database which Java method, JavaScript method, or which named function in which shared library, to invoke when a call is made. It also tells the database what type conversions to make for the arguments and return value.

> **Note:**
>
> You can also create a function as part of a package using the `CREATE PACKAGE` statement.

> **See Also:**
>
> - CREATE PROCEDURE for a general discussion of procedures and functions, CREATE PACKAGE for information on creating packages, ALTER FUNCTION and DROP FUNCTION for information on modifying and dropping a function
> - CREATE LIBRARY for information on shared libraries
> - *Oracle Database Development Guide* for more information about registering external functions
> - *JavaScript Developer's Guide*
> - CREATE MLE MODULE

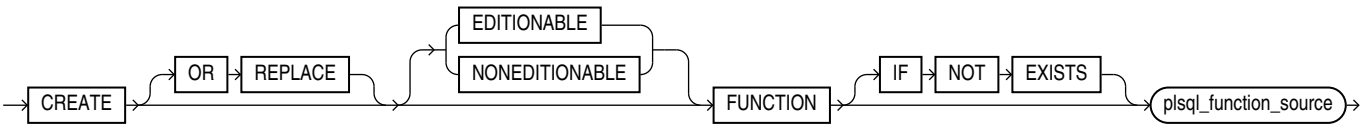
Prerequisites

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

Syntax

Functions 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_function*::=**



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

Semantics

OR REPLACE

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

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

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

> **See Also:**
>
> `ALTER FUNCTION` for information on recompiling functions using SQL

IF NOT EXISTS

Specifying `IF NOT EXISTS` has the following effects:

- If the function does not exist, a new function is created at the end of the statement.
- If the function exists, this is the function 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 function is an editioned or noneditioned object if editioning is enabled for the schema object type `FUNCTION` in *`schema`*. The default is `EDITIONABLE`. For information about editioned and noneditioned objects, see *Oracle Database Development Guide*.

plsql_function_source

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

CREATE HIERARCHY

Purpose

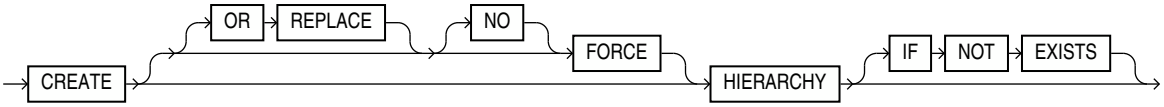
Use the `CREATE HIERARCHY` statement to create a hierarchy. A hierarchy specifies the hierarchical relationships among the levels of an attribute dimension.

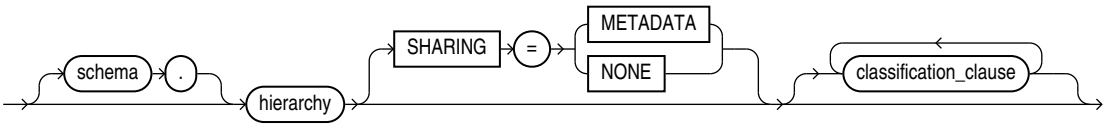
Prerequisites

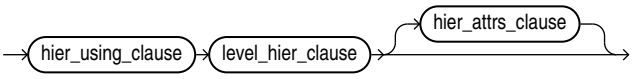
To create a hierarchy in your own schema, you must have the `CREATE HIERARCHY` system privilege. To create a hierarchy in another user's schema, you must have the `CREATE ANY HIERARCHY` system privilege.

Syntax

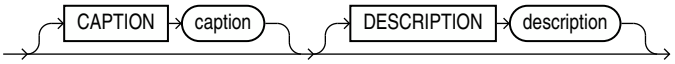
***create_hierarchy*::=**

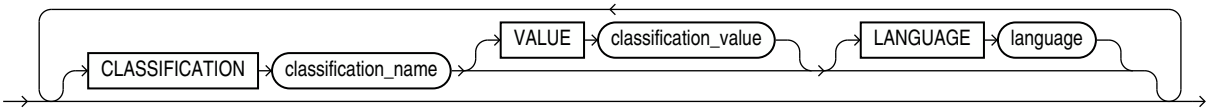




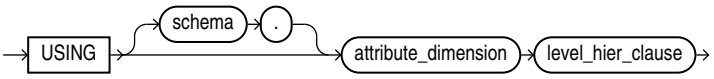


***classification_clause*::=**

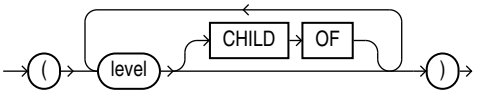




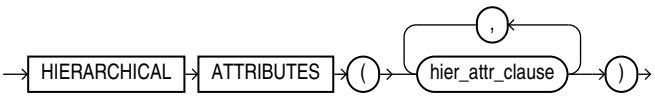
***hier_using_clause*::=**



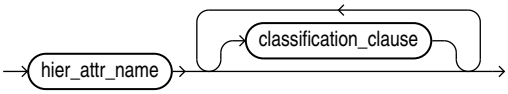
***level_hier_clause*::=**



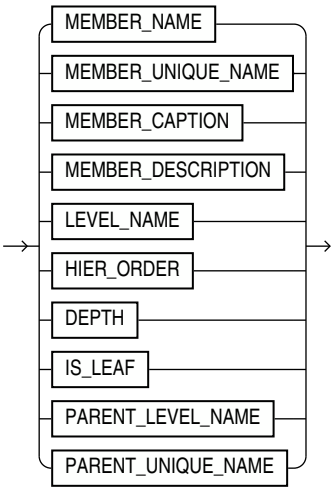
***hier_attrs_clause*::=**



***hier_attr_clause*::=**



***hier_attr_name*::=**



Semantics

OR REPLACE

Specify `OR REPLACE` to replace an existing definition of a hierarchy with a different definition.

FORCE and NOFORCE

Specify `FORCE` to force the creation of the hierarchy even if it does not successfully compile. If you specify `NOFORCE`, then the hierarchy must compile successfully, otherwise an error occurs. The default is `NOFORCE`.

IF NOT EXISTS

Specifying `IF NOT EXISTS` has the following effects:

- If the hierarchy does not exist, a new hierarchy is created at the end of the statement.
- If the hierarchy exists, this is the hierarchy 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 in which to create the hierarchy. If you do not specify a schema, then Oracle Database creates the hierarchy in your own schema.

hierarchy

Specify a name for the hierarchy.

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.

classification_clause

Use the classification clause to specify values for the `CAPTION` or `DESCRIPTION` classifications and to specify user-defined classifications. Classifications provide descriptive metadata that applications may use to provide information about analytic views and their components.

You may specify any number of classifications for the same object. A classification can have a maximum length of 4000 bytes.

For the `CAPTION` and `DESCRIPTION` classifications, you may use the DDL shortcuts `CAPTION '`*`caption`*`'` and `DESCRIPTION '`*`description`*`'` or the full classification syntax.

You may vary the classification values by language. To specify a language for the `CAPTION` or `DESCRIPTION` classification, you must use the full syntax. If you do not specify a language, then the language value for the classification is `NULL`. The language value must either be `NULL` or a valid `NLS_LANGUAGE` value.

hier_using_clause

Specify the attribute dimension that has the members of the hierarchy.

level_hier_clause

Specify the organization of the hierarchy levels.

hier_attrs_clause

Specify classifications that contain descriptive metadata for the hierarchical attributes. A *`hier_attr_clause`* for a given *`hier_attr_name`* may appear only once in the list.

All hierarchies always contain all of the hierarchical attributes, but a hierarchical attribute does not have descriptive metadata associated with it unless you specify it with this clause.

hier_attr_clause

Specify a hierarchical attribute and provide one or more classifications for it.

hier_attr_name

Specify a hierarchical attribute.

Examples

The following example creates the `TIME_HIER` hierarchy:

```
CREATE OR REPLACE HIERARCHY time_hier  -- Hierarchy name
USING time_attr_dim               -- Refers to TIME_ATTR_DIM attribute
dimension
 (month CHILD OF                  -- Months in the attribute dimension
 quarter CHILD OF
 year);
```

The following example creates the `PRODUCT_HIER` hierarchy:

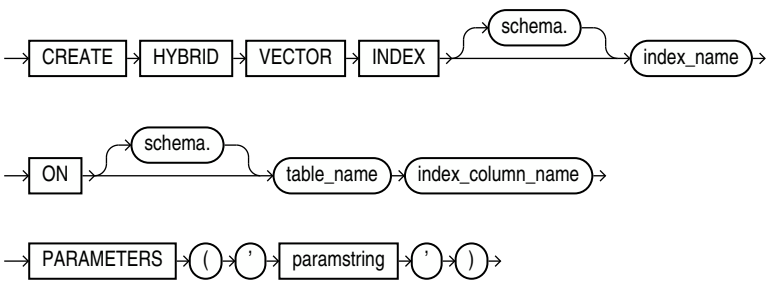
```
CREATE OR REPLACE HIERARCHY product_hier
USING product_attr_dim
 (category
  CHILD OF department);
```

The following example creates the `GEOGRAPHY_HIER` hierarchy:

```
CREATE OR REPLACE HIERARCHY geography_hier
USING geography_attr_dim
 (state_province
  CHILD OF country
  CHILD OF region);
```

CREATE HYBRID VECTOR INDEX

Syntax



Semantics

This command is part of the Oracle Text product.

Use the `CREATE HYBRID VECTOR INDEX` statement to create a hybrid vector index, which allows you to index and query documents using a combination of full-text search and similarity search to achieve higher quality search results.

Hybrid Vector Index for JSON

You can create a hybrid vector index on a JSON column. In this case *`index_column_name`* must be a column of type `JSON`, and *`paramstring`* must contain the JSON-specific path element.

> **See Also:**
>
> - For full semantics and usage details, refer to Oracle Text SQL Statements and Operators of the *Oracle Text Reference*.
> - Indexes for JSON DATA

CREATE INDEX

Purpose

Use the `CREATE INDEX` statement to create an index on:

- One or more columns of a table, a partitioned table, an index-organized table, or a cluster
- One or more scalar typed object attributes of a table or a cluster
- A nested table storage table for indexing a nested table column

An **index** is a schema object that contains an entry for each value that appears in the indexed column(s) of the table or cluster and provides direct, fast access to rows. The maximum size of a single index entry is dependent on the block size of the database.

Oracle Database supports several types of index:

- **Normal indexes.** (By default, Oracle Database creates B-tree indexes.)
- **Bitmap indexes**, which store rowids associated with a key value as a bitmap.
- **Partitioned indexes**, which consist of partitions containing an entry for each value that appears in the indexed column(s) of the table.
- **Function-based indexes**, which are based on expressions. They enable you to construct queries that evaluate the value returned by an expression, which in turn may include built-in or user-defined functions.
- **Domain indexes**, which are instances of an application-specific index of type *`indextype`*.

Note:

- *Oracle Database Concepts* for a discussion of indexes
- *Oracle Database Reference* for more information about the limits related to index size
- *Oracle Database Reference* for information on how index creation inherits compression attributes
- ALTER INDEX and DROP INDEX

Prerequisites

To create an index in your own schema, one of the following conditions must be true:

- The table or cluster to be indexed must be in your own schema.
- You must have the `INDEX` object privilege on the table to be indexed.
- You must have the `CREATE ANY INDEX` system privilege.

To create an index in another schema, you must have the `CREATE ANY INDEX` system privilege. Also, the owner of the schema to contain the index must have either the `UNLIMITED TABLESPACE` system privilege or space quota on the tablespaces to contain the index or index partitions.

To create a function-based index, in addition to the prerequisites for creating a conventional index, if the index is based on user-defined functions, then those functions must be marked `DETERMINISTIC`. A function-based index is executed with the credentials of the index owner, so the index owner must have the `EXECUTE` object privilege on the function.

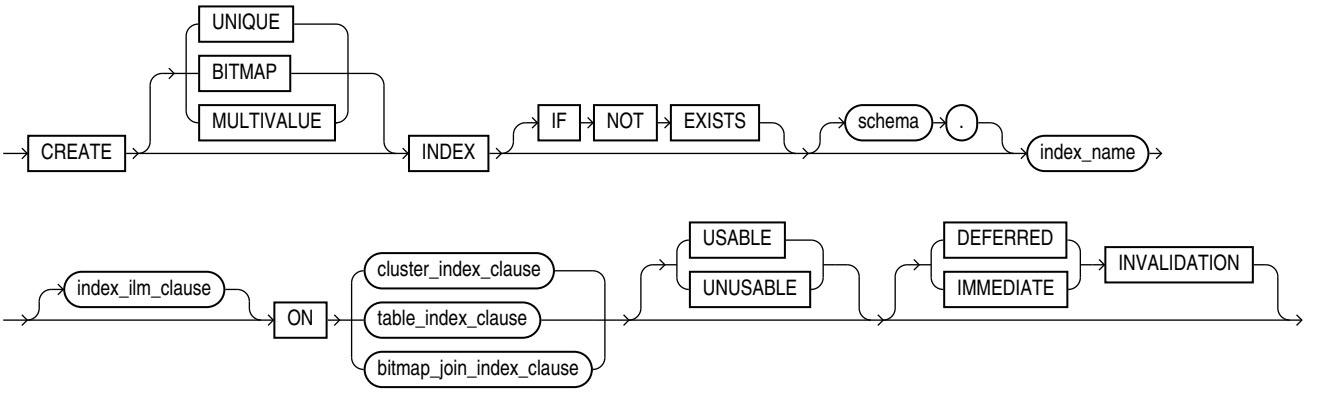
To create a domain index in your own schema, in addition to the prerequisites for creating a conventional index, you must also have the `EXECUTE` object privilege on the indextype. If you are creating a domain index in another user's schema, then the index owner also must have the `EXECUTE` object privilege on the indextype and its underlying implementation type. Before creating a domain index, you should first define the indextype.

See Also:

CREATE INDEXTYPE

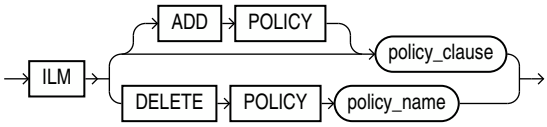
Syntax

***create_index*::=**

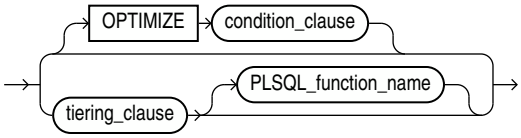


(*cluster_index_clause*::=, *table_index_clause*::=, *bitmap_join_index_clause*::=)

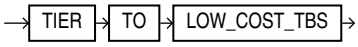
***index_ilm_clause* ::=**



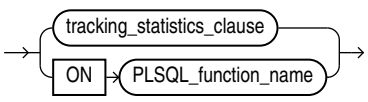
***policy_clause* ::=**



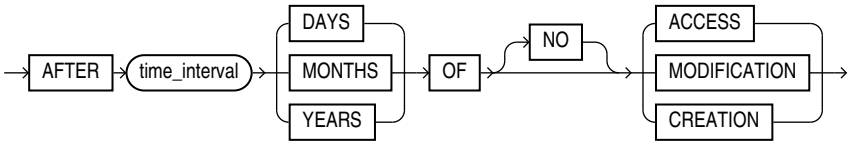
***tiering_clause* ::=**



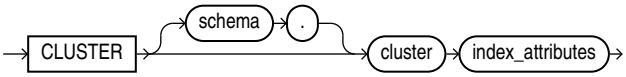
***condition_clause* ::=**



***tracking_statistics_clause* ::=**

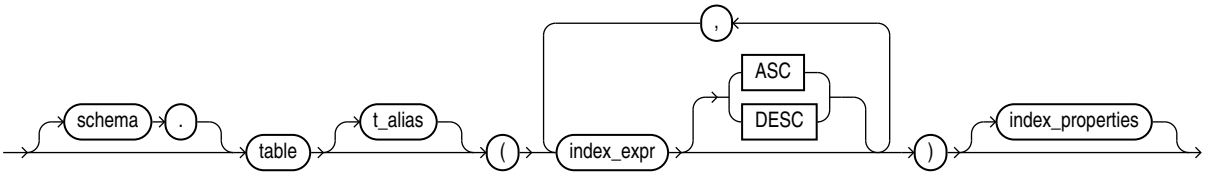


***cluster_index_clause*::=**



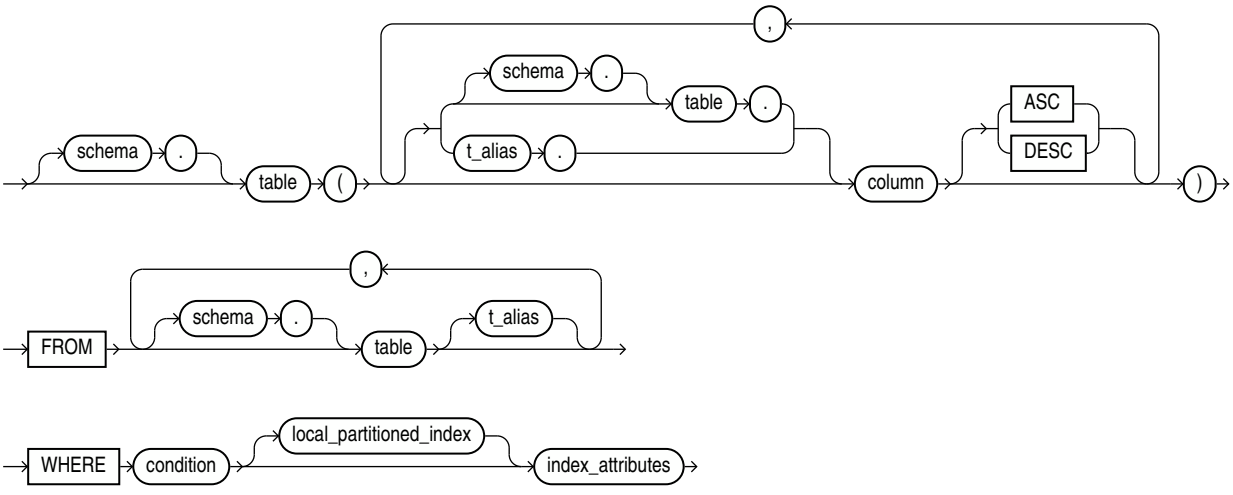
(*index_attributes*::=)

***table_index_clause*::=**



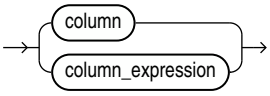
(*index_properties*::=)

***bitmap_join_index_clause*::=**

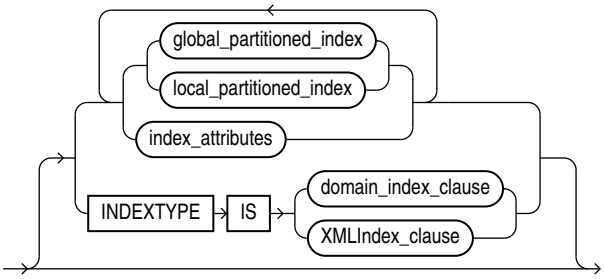


(*local_partitioned_index*::=, *index_attributes*::=)

***index_expr*::=**

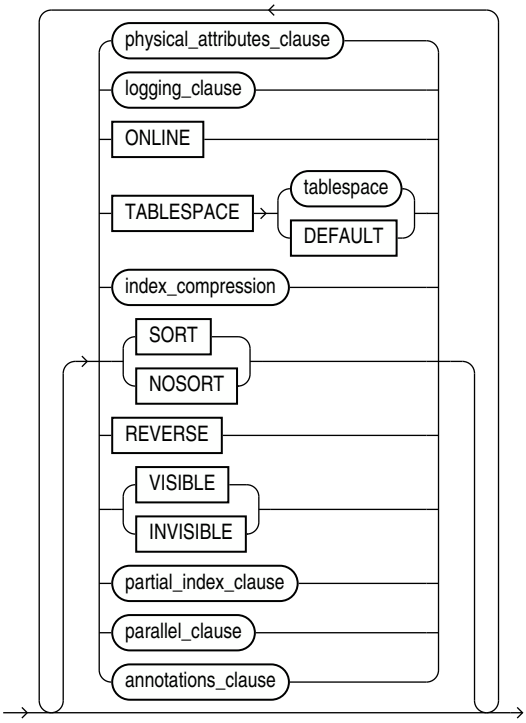


***index_properties*::=**



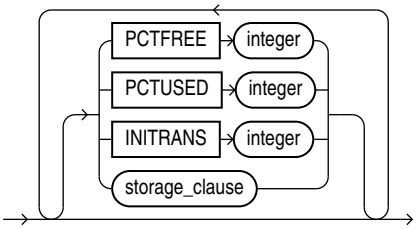
(*global_partitioned_index*::=, *local_partitioned_index*::=, *index_attributes*::=, *domain_index_clause*::=, *XMLIndex_clause*::=)

***index_attributes*::=**



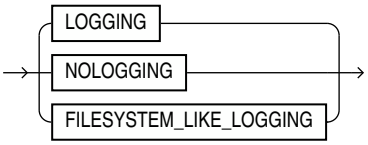
(*physical_attributes_clause*::=, *logging_clause*::=, *index_compression*::=, *partial_index_clause*::=, *parallel_clause*::=, *annotations_clause*)

***physical_attributes_clause*::=**

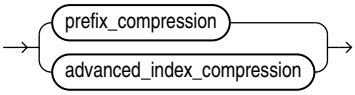


(*storage_clause*::=)

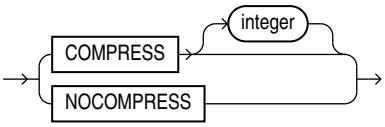
***logging_clause*::=**



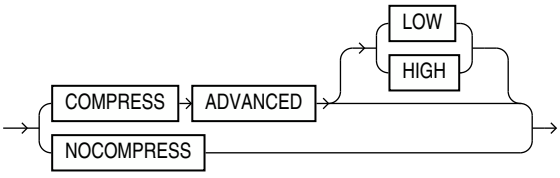
***index_compression*::=**



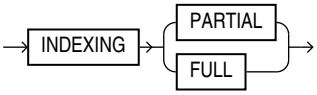
***prefix_compression*::=**



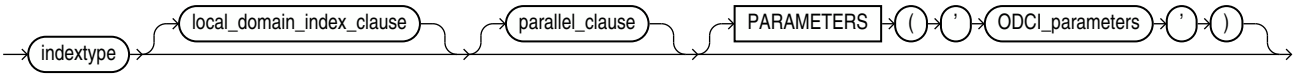
***advanced_index_compression*::=**



***partial_index_clause*::=**

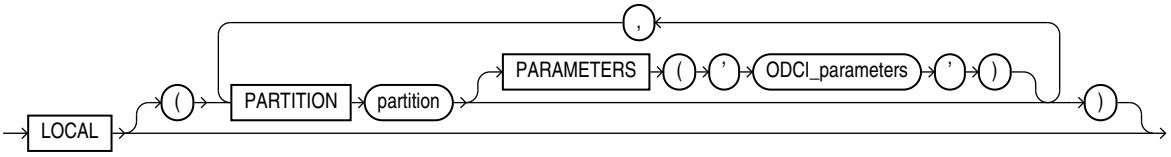


***domain_index_clause*::=**

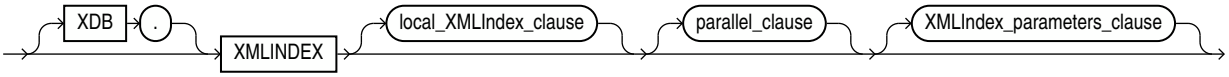


(*parallel_clause*::=)

***local_domain_index_clause*::=**

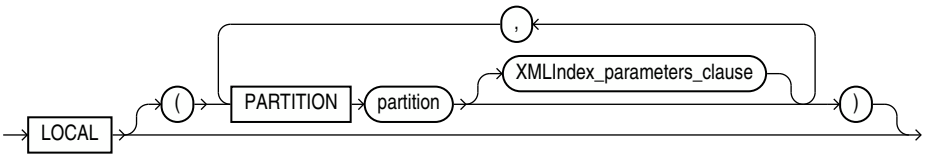


***XMLIndex_clause*::=**



(The *XMLIndex_parameters_clause* is documented in *Oracle XML DB Developer's Guide*.)

***local_XMLIndex_clause*::=**

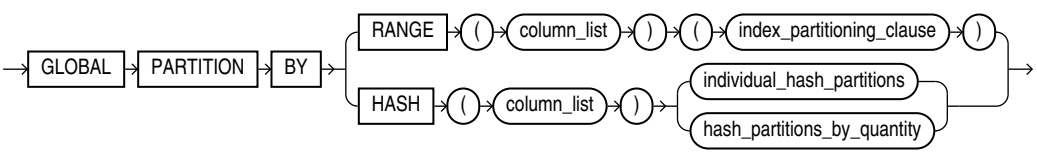


(The *XMLIndex_parameters_clause* is documented in *Oracle XML DB Developer's Guide*.)

***annotations_clause*::=**

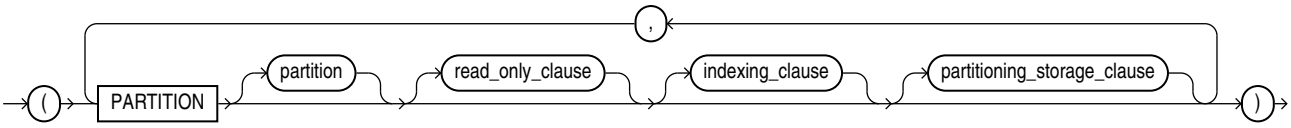
For the full syntax and semantics of the *annotations_clause* see *annotations_clause*.

***global_partitioned_index*::=**



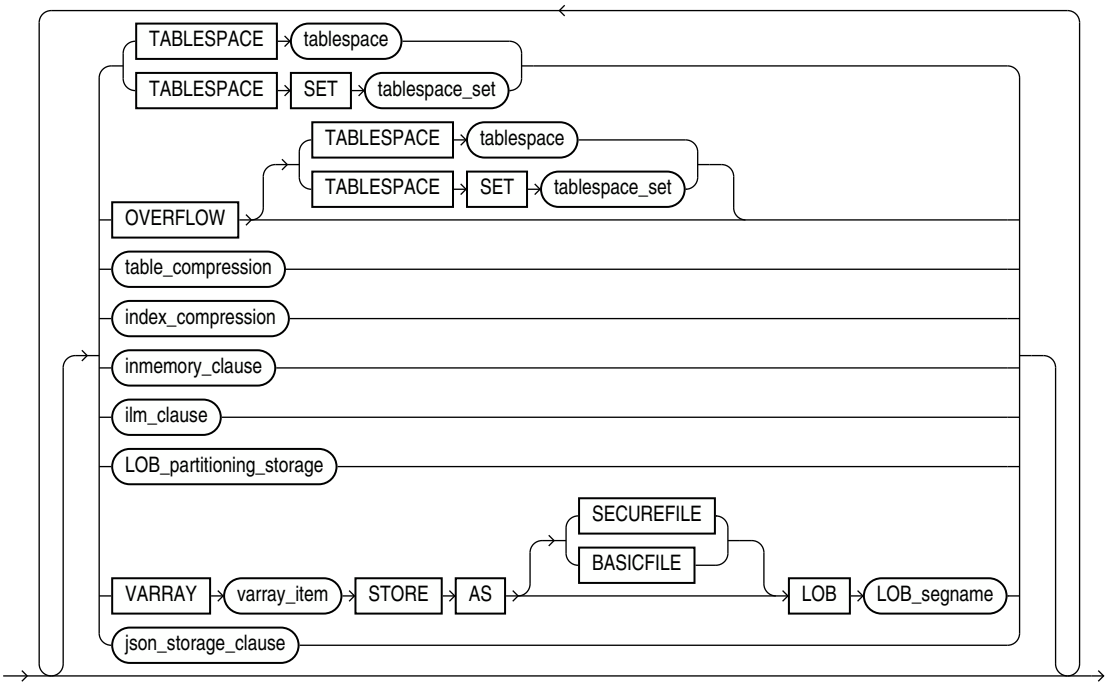
(*index_partitioning_clause*::=, *individual_hash_partitions*::=, *hash_partitions_by_quantity*::=)

***individual_hash_partitions*::=**



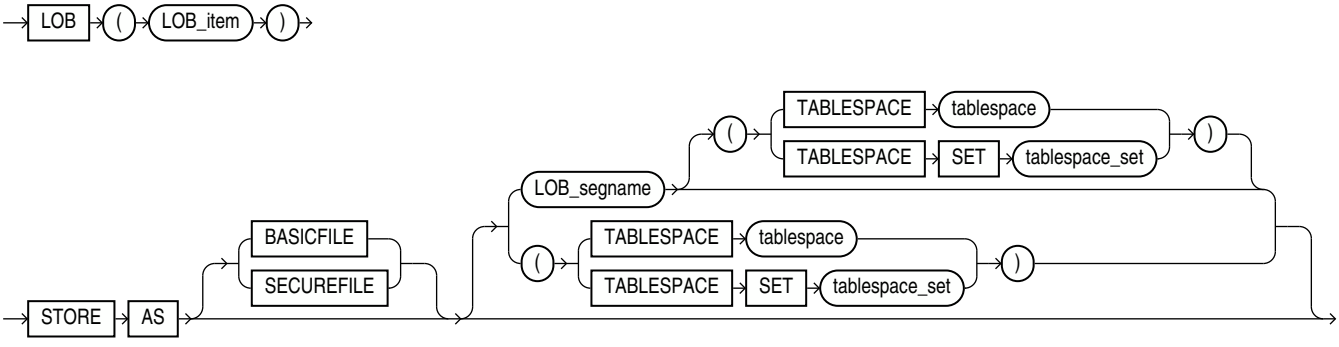
(*read_only_clause* and *indexing_clause*: not supported in *table_index_clause*, *partitioning_storage_clause*::=)

***partitioning_storage_clause*::=**



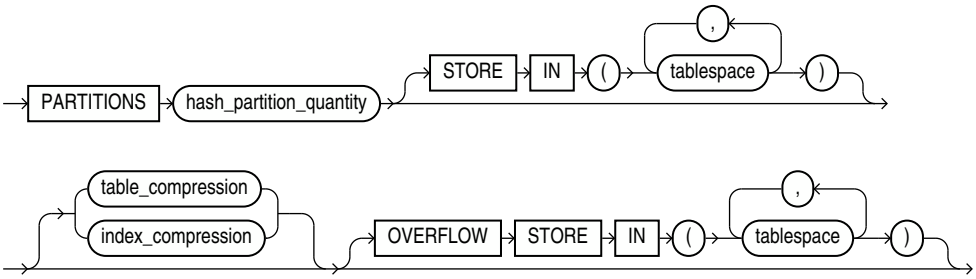
(TABLESPACE SET, *table_compression*, *inmemory_clause*, and *ilm_clause* not supported with CREATE INDEX, *index_compression*::=, *LOB_partitioning_storage*::=)

***LOB_partitioning_storage*::=**

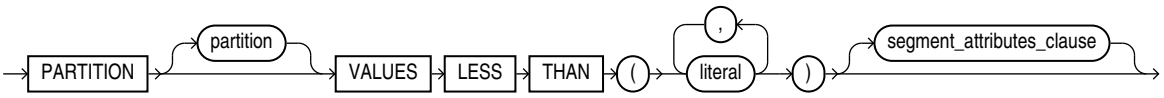


(`TABLESPACE SET`: not supported with `CREATE INDEX`)

***hash_partitions_by_quantity*::=**

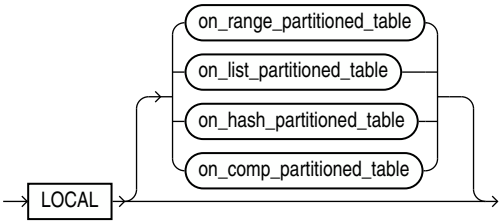


***index_partitioning_clause*::=**



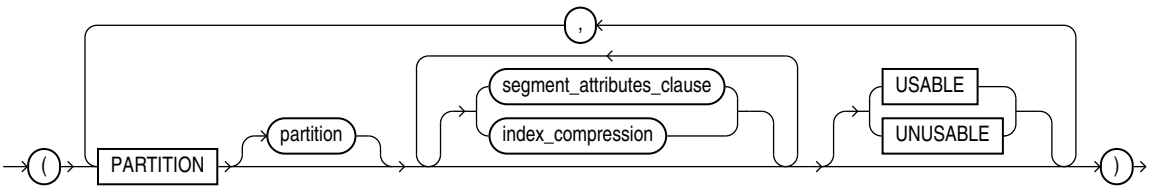
(*segment_attributes_clause*::=)

***local_partitioned_index*::=**



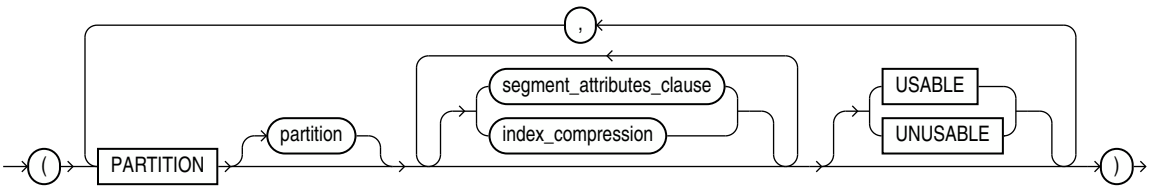
(*on_range_partitioned_table*::=, *on_list_partitioned_table*::=, *on_hash_partitioned_table*::=, *on_comp_partitioned_table*::=)

***on_range_partitioned_table*::=**



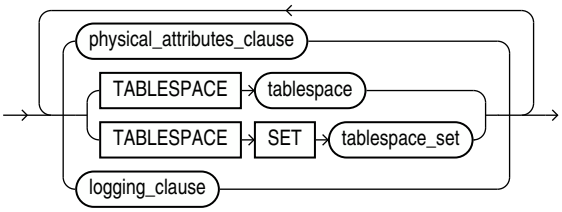
(*segment_attributes_clause*::=)

***on_list_partitioned_table*::=**



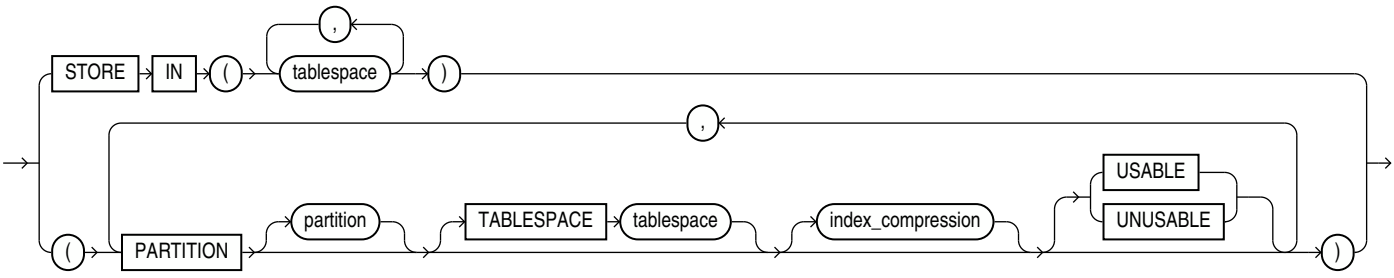
(*segment_attributes_clause*::=)

***segment_attributes_clause*::=**

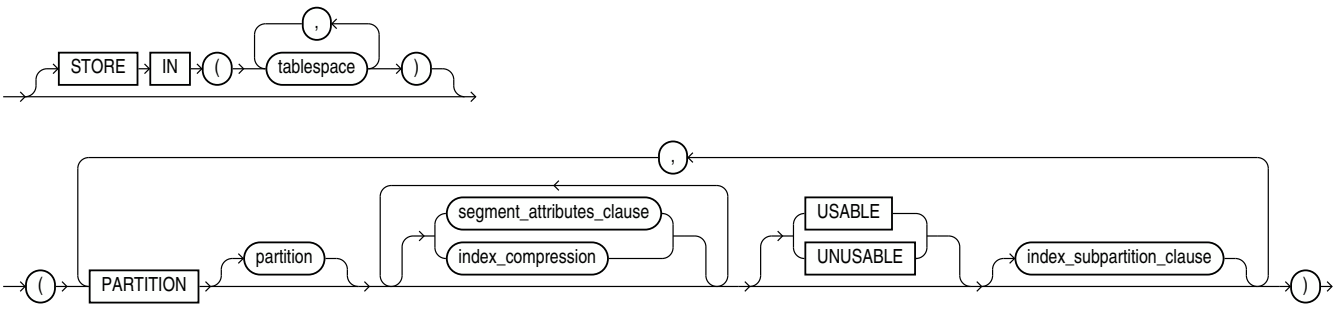


(*physical_attributes_clause*::=, `TABLESPACE SET`: not supported with `CREATE INDEX`, *logging_clause*::=

***on_hash_partitioned_table*::=**

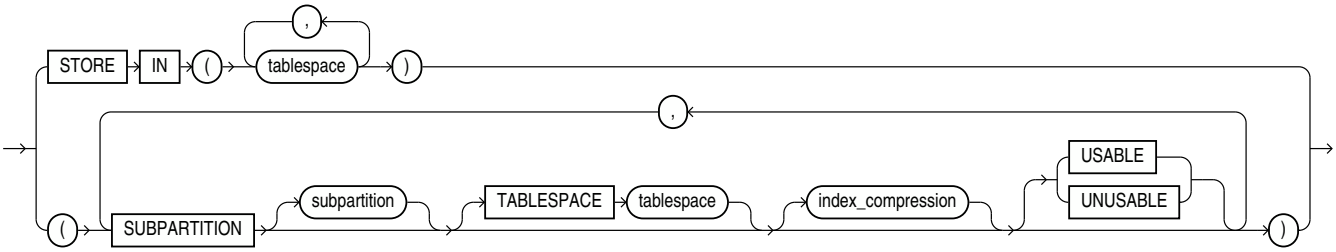


***on_comp_partitioned_table*::=**

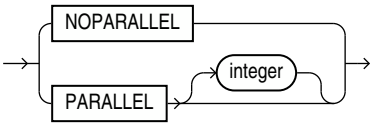


(*segment_attributes_clause*::=, *index_compression*::=, *index_subpartition_clause*::=)

***index_subpartition_clause*::=**



***parallel_clause*::=**



Semantics

UNIQUE

Specify `UNIQUE` to indicate that the value of the column (or columns) upon which the index is based must be unique.

Restrictions on Unique Indexes

Unique indexes are subject to the following restrictions:

- You cannot specify both `UNIQUE` and `BITMAP`.
- You cannot specify `UNIQUE` for a domain index.

> **See Also:**
>
> "Unique Constraints " for information on the conditions that satisfy a unique constraint

BITMAP

Specify `BITMAP` to indicate that *`index`* is to be created with a bitmap for each distinct key, rather than indexing each row separately. Bitmap indexes store the rowids associated with a key value as a bitmap. Each bit in the bitmap corresponds to a possible rowid. If the bit is set, then it means that the row with the corresponding rowid contains the key value. The internal representation of bitmaps is best suited for applications with low levels of concurrent transactions, such as data warehousing.

> **Note:**
>
> Oracle does not index table rows in which all key columns are null except in the case of bitmap indexes. Therefore, if you want an index on all rows of a table, then you must either specify `NOT NULL` constraints for the index key columns or create a bitmap index.

Restrictions on Bitmap Indexes

Bitmap indexes are subject to the following restrictions:

- You cannot specify `BITMAP` when creating a global partitioned index.
- You cannot create a bitmap secondary index on an index-organized table unless the index-organized table has a mapping table associated with it.
- You cannot specify both `UNIQUE` and `BITMAP`.
- You cannot specify `BITMAP` for a domain index.
- A bitmap index can have a maximum of 30 columns.

> **See Also:**
>
> - *Oracle Database Concepts* and *Oracle Database SQL Tuning Guide* for more information about using bitmap indexes
> - CREATE TABLE for information on mapping tables
> - "Bitmap Index Examples"

MULTIVALUE

Use the `MULTIVALUE` keyword to create a multivalue index on JSON data using simple dot-notation syntax to specify the path to the indexed data.

Example

The multivalue index created here indexes the values of top-level field `credit_score`. If the `credit_score` value targeted by a query is an array, then the index can be picked up for any array elements that are numbers. If the value is a scalar, then the index can be picked up if the scalar is a number.

```
CREATE MULTIVALUE INDEX mvi_1 ON mytable t
      (t.jcol.credit_score.numberOnly());
```

> **See Also:**
>
> - Indexes for JSON Data
> - Simple Dot-Notation Access to JSON Data

IF NOT EXISTS

Specifying `IF NOT EXISTS` has the following effects:

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

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

schema

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

index_name

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

> **See Also:**
>
> Creating an Index: Example and Creating an Index on an XMLType Table: Example

index_ilm_clause

With Oracle Database Release 20c you can use the *index_ilm_clause* to add or delete an ILM policy to an index.

You can also add an ILM policy to an index after you create it with the `ALTER INDEX` statement.

When you create an index with an ILM policy, you can add only one new policy. To add more policies to an index, or to modify existing policies on the index you must use the `ALTER INDEX` statement.

You cannot modify an ILM policy at the index partition level. The index level modification will be cascaded to all partitions.

Examples

```
CREATE INDEX [schema.]empno_idx ILM_POLICY
```

Restrictions

You cannot add an ILM policy on cluster indexes and IOTs.

You cannot add an ILM policy on domain indexes and bitmap indexes.

policy_clause

The `OPTIMIZE` index policy chooses the appropriate action if the policy condtion is met.

You can create ILM policies on objects in the same schema.

If you want to move the ILM policy to a different tablespace, you must ensure that you have the necessary permissions for all the tablespaces mentioned in the ILM policy.

You must also ensure that you have the necessary storage on the target tablespaces for storage tiering jobs.

cluster_index_clause

Use the *cluster_index_clause* to identify the cluster for which a cluster index is to be created. If you do not qualify cluster with *schema*, then Oracle Database assumes the cluster is in your current schema. You cannot create a cluster index for a hash cluster.

> **See Also:**
>
> CREATE CLUSTER and "Creating a Cluster Index: Example"

table_index_clause

Specify the table on which you are defining the index. If you do not qualify *table* with *schema*, then Oracle Database assumes the table is contained in your own schema.

You create an index on a nested table column by creating the index on the nested table storage table. Include the NESTED_TABLE_ID pseudocolumn of the storage table to create a UNIQUE index, which effectively ensures that the rows of a nested table value are distinct.

> **See Also:**
>
> "Indexes on Nested Tables: Example"

You can perform DDL operations (such as ALTER TABLE, DROP TABLE, CREATE INDEX) on a temporary table only when no session is bound to it. A session becomes bound to a temporary table by performing an INSERT operation on the table. A session becomes unbound to the temporary table by issuing a TRUNCATE statement or at session termination, or, for a transaction-specific temporary table, by issuing a COMMIT or ROLLBACK statement.

Restrictions on the *table_index_clause*

This clause is subject to the following restrictions:

- If *index* is locally partitioned, then *table* must be partitioned.
- If *table* is index-organized, then this statement creates a secondary index. The index contains the index key and the logical rowid of the index-organized table. The logical rowid excludes columns that are also part of the index key. You cannot specify REVERSE for this secondary index, and the combined size of the index key and the logical rowid should be less than the block size.
- If *table* is a temporary table, then *index* will also be temporary with the same scope (session or transaction) as *table*. The following restrictions apply to indexes on temporary tables:
  - The only part of *index_properties* you can specify is *index_attributes*.
  - Within *index_attributes*, you cannot specify the *physical_attributes_clause*, the *parallel_clause*, the *logging_clause*, or TABLESPACE.
  - You cannot create a domain index or a partitioned index on a temporary table.

- You cannot create an index on an external table.

> **See Also:**
>
> CREATE TABLE and *Oracle Database Concepts* for more information on temporary tables

t_alias

Specify a correlation name (alias) for the table upon which you are building the index.

> **Note:**
>
> This alias is required if the *index_expr* references any object type attributes or object type methods. See "Creating a Function-based Index on a Type Method: Example" and "Indexing on Substitutable Columns: Examples".

index_expr

For *index_expr*, specify the column or column expression upon which the index is based.

You can create multiple indexes on the same set of columns, column expressions, or both if the following conditions are met:

- The indexes are of different types, use different partitioning, or have different uniqueness properties.
- Only one of the indexes is `VISIBLE` at any given time.

> **See Also:**
>
> *Oracle Database Administrator's Guide* for more information on creating multiple indexes

column

Specify the name of one or more columns in the table. A bitmap index can have a maximum of 30 columns. Other indexes can have as many as 32 columns. These columns define the **index key**.

If a unique index is local nonprefixed (see *local_partitioned_index*), then the index key must contain the partitioning key.

> **See Also:**
>
> *Oracle Database VLDB and Partitioning Guide* for information on prefixed and nonprefixed indexes

You can create an index on a scalar object attribute column or on the system-defined NESTED_TABLE_ID column of the nested table storage table. If you specify an object attribute column, then the column name must be qualified with the table name. If you specify a nested table column attribute, then it must be qualified with the outermost table name, the containing column name, and all intermediate attribute names leading to the nested table column attribute.

When you create an index on a column or expression with a declared or derived named collation other than BINARY, or a declared or derived pseudo-collation USING_NLS_SORT_CI or USING_NLS_SORT_AI, the database creates a functional index on the function NLSSORT. See *Oracle Database Globalization Support Guide* for more information.

Creating an Index on an Extended Data Type Column

If *column* is an extended data type column, then you may receive a "maximum key length exceeded" error when attempting to create the index. The maximum key length for an index varies depending on the database block size and some additional index metadata stored in a block. For example, for databases that use the Oracle standard 8K block size, the maximum key length is approximately 6400 bytes.

To work around this situation, you must shorten the length of the values you want to index, using one of the following methods:

- Create a function-based index to shorten the values stored in the extended data type column as part of the expression used for the index definition.
- Create a virtual column to shorten the values stored in the extended data type column as part of the expression used for the virtual column definition and build a normal index on the virtual column. Using a virtual column also enables you to leverage functionality for regular columns, such as collecting statistics and using constraint and triggers.

For both methods you can use either the SUBSTR or STANDARD_HASH function to shorten the values of the extended data type column to build an index. These methods have the following advantages and disadvantages:

- Use the SUBSTR function to return a substring, or prefix, of *column* that is an acceptable length for the index key. This type of index can be used for equality, IN-list, and range predicates on the original column without the need to specify the SUBSTR column as part of the predicate. Refer to SUBSTR for more information.
- Using the STANDARD_HASH function is likely to create an index that is more compact than the substring-based index and may result in fewer unnecessary index accesses. This type of index can be used for equality and IN-list predicates on the original column without the need to specify the STANDARD_HASH column as part of the predicate. Refer to STANDARD_HASH for more information.

The following example shows how to create a function-based index on an extended data type column:

```
CREATE INDEX index ON table (SUBSTR(column, 1, n));
```

For n, specify a prefix length that is large enough to differentiate between values in *column*.

The following example shows how to create a virtual column for an extended data type column, and then create an index on the virtual column:

```
ALTER TABLE table ADD (new_hash_column AS (STANDARD_HASH(column)));
CREATE INDEX index ON table (new_hash_column);
```

> **See Also:**
>
> "Extended Data Types" for more information on extended data types

Restrictions on Index Columns

The following restrictions apply to index columns:

- You cannot create an index on columns or attributes whose type is user-defined, `LONG`, `LONG RAW`, LOB, or `REF`, except that Oracle Database supports an index on `REF` type columns or attributes that have been defined with a `SCOPE` clause.
- Only normal (B-tree) indexes can be created on encrypted columns, and they can only be used for equality searches.

column_expression

Specify an expression built from columns of *table*, constants, SQL functions, and user-defined functions. When you specify *column_expression*, you create a **function-based index**.

> **See Also:**
>
> "Column Expressions ", "Notes on Function-based Indexes", "Restrictions on Function-based Indexes", and "Function-Based Index Examples"

Name resolution of the function is based on the schema of the index creator. User-defined functions used in *column_expression* are fully name resolved during the `CREATE INDEX` operation.

After creating a function-based index, collect statistics on both the index and its base table using the `DBMS_STATS` package. Such statistics will enable Oracle Database to correctly decide when to use the index.

Function-based unique indexes can be useful in defining a conditional unique constraint on a column or combination of columns. Refer to "Using a Function-based Index to Define Conditional Uniqueness: Example" for an example.

> **See Also:**
>
> *Oracle Database PL/SQL Packages and Types Reference* for more information on the `DBMS_STATS` package

Notes on Function-based Indexes

The following notes apply to function-based indexes:

- When you subsequently query a table that uses a function-based index, Oracle Database will not use the index unless the query filters out nulls. However, Oracle Database will use a function-based index in a query even if the columns specified in the `WHERE` clause are in a different order than their order in the *column_expression* that defined the function-based index.

> **See Also:**
>
> "Function-Based Index Examples"

- If the function on which the index is based becomes invalid or is dropped, then Oracle Database marks the index `DISABLED`. Queries on a `DISABLED` index fail if the optimizer chooses to use the index. DML operations on a `DISABLED` index fail unless the index is also marked `UNUSABLE` **and** the parameter `SKIP_UNUSABLE_INDEXES` is set to `true`. Refer to ALTER SESSION for more information on this parameter.
- If a public synonym for a function, package, or type is used in *`column_expression`*, and later an actual object with the same name is created in the table owner's schema, then Oracle Database disables the function-based index. When you subsequently enable the function-based index using `ALTER INDEX ... ENABLE` or `ALTER INDEX ... REBUILD`, the function, package, or type used in the *`column_expression`* continues to resolve to the function, package, or type to which the public synonym originally pointed. It will not resolve to the new function, package, or type.
- If the definition of a function-based index generates internal conversion to character data, then use caution when changing NLS parameter settings. Function-based indexes use the current database settings for NLS parameters. If you reset these parameters at the session level, then queries using the function-based index may return incorrect results. Two exceptions are the collation parameters (`NLS_SORT` and `NLS_COMP`). Oracle Database handles the conversions correctly even if these have been reset at the session level.
- Oracle Database cannot convert data in all cases, even when conversion is explicitly requested. For example, an attempt to convert the string `'105 lbs'` from `VARCHAR2` to `NUMBER` using the `TO_NUMBER` function fails with an error. Therefore, if *`column_expression`* contains a data conversion function such as `TO_NUMBER` or `TO_DATE`, and if a subsequent `INSERT` or `UPDATE` statement includes data that the conversion function cannot convert, then the index will cause the `INSERT` or `UPDATE` statement to fail.
- If *`column_expression`* contains a datetime format model, then the function-based index expression defining the column may contain format elements that are different from those specified. For example, define a function-based index using the `yyyy` datetime format element:

  ```
  CREATE INDEX cust_eff_ix ON customers
    (NVL(cust_eff_to, TO_DATE('9000-01-01 00:00:00', 'yyyy-mm-dd hh24:mi:ss')));
  ```

  Query the `ALL_IND_EXPRESSIONS` view to see that the function-based index expression defining the column uses the `syyyy` datetime format element:

  ```
  SELECT column_expression
    FROM all_ind_expressions
    WHERE index_name='CUST_EFF_IX';

  COLUMN_EXPRESSION
  --------------------------------------------------------------------------------
  NVL("CUST_EFF_TO",TO_DATE(' 9000-01-01 00:00:00', 'syyyy-mm-dd hh24:mi:ss'))
  ```

Restrictions on Function-based Indexes

Function-based indexes are subject to the following restrictions:

- The value returned by the function referenced in *`column_expression`* is subject to the same restrictions as are the index columns of a B-tree index. Refer to "Restrictions on Index Columns".

- Any user-defined function referenced in *column_expression* must be declared as DETERMINISTIC.
- For a function-based globally partitioned index, the *column_expression* cannot be the partitioning key.
- The *column_expression* can be any of the forms of expression described in Column Expressions .
- All functions must be specified with parentheses, even if they have no parameters. Otherwise Oracle Database interprets them as column names.
- Any function you specify in *column_expression* must return a repeatable value. For example, you cannot specify the SYSDATE or USER function or the ROWNUM pseudocolumn.

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

ASC | DESC

Use ASC or DESC to indicate whether the index should be created in ascending or descending order. Indexes on character data are created in ascending or descending order of the character values in the database character set.

Oracle Database treats descending indexes as if they were function-based indexes. As with other function-based indexes, the database does not use descending indexes until you first analyze the index and the table on which the index is defined. See the *column_expression* clause of this statement.

Ascending unique indexes allow multiple NULL values. However, in descending unique indexes, multiple NULL values are treated as duplicate values and therefore are not permitted.

Restriction on Ascending and Descending Indexes

You cannot specify either of these clauses for a domain index. You cannot specify DESC for a reverse index. Oracle Database ignores DESC if *index* is bitmapped or if the COMPATIBLE initialization parameter is set to a value less than 8.1.0.

index_attributes

Specify the optional index attributes.

physical_attributes_clause

Use the *physical_attributes_clause* to establish values for physical and storage characteristics for the index.

If you omit this clause, then Oracle Database sets PCTFREE to 10 and INITRANS to 2.

Restriction on Index Physical Attributes

You cannot specify the PCTUSED parameter for an index.

> **See Also:**
>
> *physical_attributes_clause* and storage_clause for a complete description of these clauses

TABLESPACE

For *tablespace*, specify the name of the tablespace to hold the index, index partition, or index subpartition. If you omit this clause, then Oracle Database creates the index in the default tablespace of the owner of the schema containing the index.

For a local index, you can specify the keyword DEFAULT in place of *tablespace*. New partitions or subpartitions added to the local index will be created in the same tablespace(s) as the corresponding partitions or subpartitions of the underlying table.

index_compression

The *index_compression* clauses let you enable or disable index compression for the index. Specify the COMPRESS clause of *prefix_compression* to enable prefix compression for the index, specify the COMPRESS ADVANCED clause of *advanced_index_compression* to enable advanced index compression for the index, or specify the NOCOMPRESS clause of either *prefix_compression* or *advanced_index_compression* to disable compression for the index. The default is NOCOMPRESS.

If you want to use compression for a partitioned index, then you must create the index with compression enabled at the index level. You can subsequently enable and disable the compression setting for individual partitions of such a partitioned index. You can also enable and disable compression when rebuilding individual partitions. You can modify an existing nonpartitioned index to enable or disable compression only when rebuilding the index.

prefix_compression

Specify COMPRESS to enable **prefix compression**, also known as key compression, which eliminates repeated occurrence of key column values. Use *integer* to specify the prefix length (number of prefix columns to compress). You can specify prefix compression for indexes that are nonunique or unique indexes of at least two columns.

- For unique indexes, the range of valid prefix length values is from 1 to the number of key columns minus 1. The default prefix length is the number of key columns minus 1.
- For nonunique indexes, the range of valid prefix length values is from 1 to the number of key columns. The default prefix length is the number of key columns.

advanced_index_compression

Specify this clause to enable **advanced index compression**. Advanced index compression improves compression ratios significantly while still providing efficient access to indexes. Therefore, advanced index compression works well on all supported indexes, including those indexes that are not good candidates for prefix compression.

- COMPRESS ADVANCED LOW - This level compresses the index less than the HIGH level, but provides faster access to the index. You can specify this clause for indexes that are nonunique or unique indexes of at least two columns. Before enabling COMPRESS ADVANCED LOW, the database must be at 12.1.0 or higher compatibility level.
- COMPRESS ADVANCED HIGH - This level compresses the index more than the LOW level, but provides slower access to the index. You can specify this clause for indexes that are

nonunique or unique indexes of one or more columns. Before enabling `COMPRESS ADVANCED HIGH`, the database must be at `12.2.0` or higher compatibility level.

If you omit the `LOW` and `HIGH` keywords, then the default is `HIGH`.

Restrictions on Index Compression

The following restrictions apply to index compression:

- You cannot specify prefix compression or advanced index compression for a bitmap index.
- You cannot specify advanced index compression for index-organized tables.

> **See Also:**
>
> - DB_INDEX_COMPRESSION_INHERITANCE for more on how index creation inherits compression attributes
> - *Oracle Database Administrator's Guide* for more information on prefix compression and advanced index compression
> - "Compressing an Index: Example"

partial_index_clause

You can specify this clause only when creating an index on a partitioned table. Specify `INDEXING FULL` to create a **full index**. Specify `INDEXING PARTIAL` to create a **partial index**. The default is `INDEXING FULL`.

A full index includes all partitions in the underlying table, regardless of their indexing properties. A partial index includes only partitions in the underlying table with an indexing property of `ON`.

If a partial index is a local partitioned index, then index partitions that correspond with table partitions with an indexing property of `ON` are marked `USABLE`. Index partitions that correspond with table partitions with an indexing property of `OFF` are marked `UNUSABLE`.

If the underlying table is a composite-partitioned table, then the preceding conditions for index partitions and table partitions apply instead to index subpartitions and table subpartitions.

Restrictions on Partial Indexes

Partial indexes are subject to the following restrictions:

- The underlying table of a partial index cannot be a nonpartitioned table.
- Unique indexes cannot be partial indexes. This applies to indexes created with the `CREATE UNIQUE INDEX` statement and indexes that are implicitly created when you specify a unique constraint on one or more columns.

> **See Also:**
>
> `CREATE TABLE` *indexing_clause* for information on the indexing property

SORT | NOSORT

By default, Oracle Database sorts indexes in ascending order when it creates the index. You can specify `NOSORT` to indicate to the database that the rows are already stored in the database in ascending order, so that Oracle Database does not have to sort the rows when creating the index. If the rows of the indexed column or columns are not stored in ascending order, then the database returns an error. For greatest savings of sort time and space, use this clause immediately after the initial load of rows into a table. If you specify neither of these keywords, then `SORT` is the default.

Restrictions on NOSORT

This parameter is subject to the following restrictions:

- You cannot specify `REVERSE` with this clause.
- You cannot use this clause to create a cluster index partitioned or bitmap index.
- You cannot specify this clause for a secondary index on an index-organized table.

REVERSE

Specify `REVERSE` to store the bytes of the index block in reverse order, excluding the rowid.

Restrictions on Reverse Indexes

Reverse indexes are subject to the following restrictions:

- You cannot specify `NOSORT` with this clause.
- You cannot reverse a bitmap index or an index on an index-organized table.

VISIBLE | INVISIBLE

Use this clause to specify whether the index is visible or invisible to the optimizer. An invisible index is maintained by DML operations, but it is not be used by the optimizer during queries unless you explicitly set the parameter `OPTIMIZER_USE_INVISIBLE_INDEXES` to `TRUE` at the session or system level.

To determine whether an existing index is visible or invisible to the optimizer, you can query the `VISIBILITY` column of the `USER_`, `DBA_`, `ALL_INDEXES` data dictionary views.

> **See Also:**
>
> *Oracle Database Administrator's Guide* for more information on this feature

logging_clause

Specify whether the creation of the index will be logged (`LOGGING`) or not logged (`NOLOGGING`) in the redo log file. This setting also determines whether subsequent Direct Loader (SQL*Loader) and direct-path `INSERT` operations against the index are logged or not logged. `LOGGING` is the default.

If *`index`* is nonpartitioned, then this clause specifies the logging attribute of the index.

If *`index`* is partitioned, then this clause determines:

- The default value of all partitions specified in the `CREATE` statement, unless you specify the *`logging_clause`* in the `PARTITION` description clause
- The default value for the segments associated with the index partitions
- The default value for local index partitions or subpartitions added implicitly during subsequent `ALTER TABLE ... ADD PARTITION` operations

The logging attribute of the index is independent of that of its base table.

If you omit this clause, then the logging attribute is that of the tablespace in which it resides.

> **See Also:**
>
> - *logging_clause* for a full description of this clause
> - *Oracle Database VLDB and Partitioning Guide* for more information about logging and parallel DML
> - "Creating an Index in NOLOGGING Mode: Example"

ONLINE

Specify `ONLINE` to indicate that DML operations on the table will be allowed during creation of the index.

Restrictions on Online Index Building

Online index building is subject to the following restrictions:

- Parallel DML is not supported during online index building. If you specify `ONLINE` and then issue parallel DML statements, then Oracle Database returns an error.
- You can specify `ONLINE` for a bitmap index or a cluster index as long as you set `COMPATIBLE` to 10 or higher.
- You cannot specify `ONLINE` for a conventional index on a `UROWID` column.
- For a nonunique secondary index on an index-organized table, the number of index key columns plus the number of primary key columns that are included in the logical rowid in the index-organized table cannot exceed 32. The logical rowid excludes columns that are part of the index key.

> **See Also:**
>
> *Oracle Database Concepts* for a description of online index building and rebuilding

parallel_clause

Specify the *`parallel_clause`* if you want creation of the index to be parallelized.

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

Index Partitioning Clauses

Use the *global_partitioned_index* clause and the *local_partitioned_index* clauses to partition *index*.

The storage of partitioned database entities in tablespaces of different block sizes is subject to several restrictions. Refer to *Oracle Database VLDB and Partitioning Guide* for a discussion of these restrictions.

> **See Also:**
>
> "Partitioned Index Examples"

annotations_clause

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

global_partitioned_index

The *global_partitioned_index* clause lets you specify that the partitioning of the index is user defined and is not equipartitioned with the underlying table. By default, nonpartitioned indexes are global indexes.

You can partition a global index by range or by hash. In both cases, you can specify up to 32 columns as partitioning key columns. The partitioning column list must specify a left prefix of the index column list. If the index is defined on columns `a`, `b`, and `c`, then for the columns you can specify `(a, b, c)`, or `(a, b)`, or `(a, c)`, but you cannot specify `(b, c)` or `(c)` or `(b, a)`. If you specify a partition name, then it must conform to the rules for naming schema objects and their parts as described in "Database Object Naming Rules ". If you omit the partition names, then Oracle Database assigns names of the form `SYS_Pn`.

GLOBAL PARTITION BY RANGE

Use this clause to create a range-partitioned global index. Oracle Database will partition the global index on the ranges of values from the table columns you specify in the column list.

> **See Also:**
>
> "Creating a Range-Partitioned Global Index: Example"

GLOBAL PARTITION BY HASH

Use this clause to create a hash-partitioned global index. Oracle Database assigns rows to the partitions using a hash function on values in the partitioning key columns.

> **See Also:**
>
> The `CREATE TABLE` clause *hash_partitions* for information on the two methods of hash partitioning and "Creating a Hash-Partitioned Global Index: Example"

Restrictions on Global Partitioned Indexes

Global partitioned indexes are subject to the following restrictions:

- The partitioning key column list cannot contain the `ROWID` pseudocolumn or a column of type `ROWID`.
- The only property you can specify for hash partitions is tablespace storage. Therefore, you cannot specify LOB or varray storage clauses in the *partitioning_storage_clause* of *individual_hash_partitions*.
- You cannot specify the `OVERFLOW` clause of *hash_partitions_by_quantity*, as that clause is valid only for index-organized table partitions.
- In the *partitioning_storage_clause*, you cannot specify *table_compression* or the *inmemory_clause*, but you can specify *index_compression*.

> **Note:**
>
> If your enterprise has or will have databases using different character sets, then use caution when partitioning on character columns. The sort sequence of characters is not identical in all character sets.

> **See Also:**
>
> *Oracle Database Globalization Support Guide* for more information on character set support

index_partitioning_clause

Use this clause to describe the individual index partitions. The number of repetitions of this clause determines the number of partitions. If you omit *partition*, then Oracle Database generates a name with the form `SYS_Pn`.

For `VALUES LESS THAN` (*value_list*), specify the noninclusive upper bound for the current partition in a global index. The value list is a comma-delimited, ordered list of literal values corresponding to the column list in the *global_partitioned_index* clause. Always specify `MAXVALUE` as the value of the last partition.

> **Note:**
>
> If the index is partitioned on a `DATE` column, and if the date format does not specify the first two digits of the year, then you must use the `TO_DATE` function with a 4-character format mask for the year. The date format is determined implicitly by `NLS_TERRITORY` or explicitly by `NLS_DATE_FORMAT`. Refer to *Oracle Database Globalization Support Guide* for more information on these initialization parameters.

> **See Also:**
>
> "Range Partitioning Example"

local_partitioned_index

The *local_partitioned_index* clauses let you specify that the index is partitioned on the same columns, with the same number of partitions and the same partition bounds as *table*. For composite-partitioned tables, this clause lets you specify that the index is subpartitioned on the same columns, with the same number of subpartitions and the same subpartition bounds as *table*. Oracle Database automatically maintains local index partitioning as the underlying table is repartitioned.

If you specify only the keyword LOCAL and do not specify a subclause, then Oracle Database creates each index partition in the same tablespace as its corresponding table partition and assigns it the same name as its corresponding table partition. If *table* is a composite-partitioned table, then Oracle Database creates each index subpartition in the same tablespace as its corresponding table subpartition and assigns it the same name as its corresponding table subpartition.

If you specify a partition name, then it must conform to the rules for naming schema objects and their parts as described in "Database Object Naming Rules ". If you omit a partition name, then Oracle Database generates a name that is consistent with the corresponding table partition. If the name conflicts with an existing index partition name, then the database uses the form SYS_Pn.

on_range_partitioned_table

This clause lets you specify the names and attributes of index partitions on a range-partitioned table. If you specify this clause, then the number of PARTITION clauses must be equal to the number of table partitions, and in the same order.

You cannot specify prefix compression for an index partition unless you have specified prefix compression for the index.

For more information on the USABLE and UNUSABLE clauses, refer to USABLE | UNUSABLE.

on_list_partitioned_table

The *on_list_partitioned_table* clause is identical to on_range_partitioned_table.

on_hash_partitioned_table

This clause lets you specify names and tablespace storage for index partitions on a hash-partitioned table.

If you specify any PARTITION clauses, then the number of these clauses must be equal to the number of table partitions. You can optionally specify tablespace storage for one or more individual partitions. If you do not specify tablespace storage either here or in the STORE IN clause, then the database stores each index partition in the same tablespace as the corresponding table partition.

The STORE IN clause lets you specify one or more tablespaces across which Oracle Database will distribute all the index hash partitions. The number of tablespaces need not equal the number of index partitions. If the number of index partitions is greater than the number of tablespaces, then the database cycles through the names of the tablespaces.

For more information on the USABLE and UNUSABLE clauses, refer to USABLE | UNUSABLE.

on_comp_partitioned_table

This clause lets you specify the name and attributes of index partitions on a composite-partitioned table.

The STORE IN clause is valid only for range-hash or list-hash composite-partitioned tables. It lets you specify one or more default tablespaces across which Oracle Database will distribute all index hash subpartitions for all partitions. You can override this storage by specifying different default tablespace storage for the subpartitions of an individual partition in the second STORE IN clause in the *index_subpartition_clause*.

For range-range, range-list, and list-list composite-partitioned tables, you can specify default attributes for the range or list subpartitions in the PARTITION clause. You can override this storage by specifying different attributes for the range or list subpartitions of an individual partition in the SUBPARTITION clause of the *index_subpartition_clause*.

You cannot specify prefix compression for an index partition unless you have specified prefix compression for the index.

For more information on the USABLE and UNUSABLE clauses, refer to USABLE | UNUSABLE.

index_subpartition_clause

This clause lets you specify names and tablespace storage for index subpartitions in a composite-partitioned table.

The STORE IN clause is valid only for hash subpartitions of a range-hash and list-hash composite-partitioned table. It lets you specify one or more tablespaces across which Oracle Database will distribute all the index hash subpartitions. The SUBPARTITION clause is valid for all subpartition types.

If you specify any SUBPARTITION clauses, then the number of those clauses must be equal to the number of table subpartitions. If you specify a subpartition name, then it must conform to the rules for naming schema objects and their parts as described in "Database Object Naming Rules ". If you omit *subpartition*, then the database generates a name that is consistent with the corresponding table subpartition. If the name conflicts with an existing index subpartition name, then the database uses the form SYS_SUBP*n*.

The number of tablespaces need not equal the number of index subpartitions. If the number of index subpartitions is greater than the number of tablespaces, then the database cycles through the names of the tablespaces.

If you do not specify tablespace storage for subpartitions either in the *on_comp_partitioned_table* clause or in the *index_subpartition_clause*, then Oracle Database uses the tablespace specified for *index*. If you also do not specify tablespace storage for *index*, then the database stores the subpartition in the same tablespace as the corresponding table subpartition.

For more information on the USABLE and UNUSABLE clauses, refer to CREATE INDEX ... USABLE | UNUSABLE.

domain_index_clause

Use the *domain_index_clause* to indicate that *index* is a domain index, which is an instance of an application-specific index of type *indextype*.

Creating a domain index requires a number of preceding operations. You must first create an implementation type for an indextype. You must also create a functional implementation and

then create an operator that uses the function. Next you create an indextype, which associates the implementation type with the operator. Finally, you create the domain index using this clause. Refer to Extended Examples, which contains an example of creating a simple domain index, including all of these operations.

index_expr

In the *index_expr* (in *table_index_clause*), specify the table columns or object attributes on which the index is defined. You can define multiple domain indexes on a single column only if the underlying indextypes are different and the indextypes support a disjoint set of user-defined operators.

Restrictions on Domain Indexes

Domain indexes are subject to the following restrictions:

- The *index_expr* (in *table_index_clause*) can specify only a single column, and the column cannot be of data type REF, varray, nested table, LONG, or LONG RAW.
- You cannot create a bitmap or unique domain index.
- You cannot create a domain index on a temporary table.
- You can create local domain indexes on only range-, list-, hash-, and interval-partitioned tables, with one exception: You cannot create a local domain index on an automatic list-partitioned table.
- Domain indexes can be created only on table columns declared with collation BINARY, USING_NLS_COMP, USING_NLS_SORT, or USING_NLS_SORT_CS. See *Oracle Database Globalization Support Guide* for more information.

indextype

For *indextype*, specify the name of the indextype. This name should be a valid schema object that has already been created.

If you have installed Oracle Text, then you can use various built-in indextypes to create Oracle Text domain indexes. For more information on Oracle Text and the indexes it uses, refer to *Oracle Text Reference*.

> **See Also:**
>
> CREATE INDEXTYPE

local_domain_index_clause

Use this clause to specify that the index is a local index on a partitioned table.

- The PARTITIONS clause lets you specify names for the index partitions. The number of partitions you specify must match the number of partitions in the base table. If you omit this clause, then the database creates the partitions with system-generated names of the form SYS_P*n*.
- The PARAMETERS clause lets you specify the parameter string specific to an individual partition. If you omit this clause, then the parameter string associated with the index is also associated with the partition.

parallel_clause

Use the *parallel_clause* to parallelize creation of the domain index. For a nonpartitioned domain index, Oracle Database passes the explicit or default degree of parallelism to the ODCIIndexCreate cartridge routine, which in turn establishes parallelism for the index. For local domain indexes, this clause causes the index partitions to be created in parallel.

> **See Also:**
>
> *Oracle Database Data Cartridge Developer's Guide* for complete information on the Oracle Data Cartridge Interface (ODCI) routines

PARAMETERS

In the PARAMETERS clause, specify the parameter string that is passed uninterpreted to the appropriate ODCI indextype routine. The maximum length of the parameter string is 1000 characters.

When you specify this clause at the top level of the syntax, the parameters become the default parameters for the index partitions. If you specify this clause as part of the *local_domain_index_clause*, then you override any default parameters with parameters for the individual partition.

After the domain index is created, Oracle Database invokes the appropriate ODCI routine. If the routine does not return successfully, then the domain index is marked FAILED. The only operations supported on an failed domain index are DROP INDEX and (for non-local indexes) REBUILD INDEX.

> **See Also:**
>
> *Oracle Database Data Cartridge Developer's Guide* for information on the Oracle Data Cartridge Interface (ODCI) routines

XMLIndex_clause

The *XMLIndex_clause* lets you define an XMLIndex index, typically on a column contain XML data. An XMLIndex index is a type of domain index designed specifically for the domain of XML data.

XMLIndex_parameters_clause

This clause lets you specify information about the path table and about the secondary indexes corresponding to the components of XMLIndex. This clause also lets you specify information about the structured component of the index. The maximum length of the parameter string is 1000 characters.

When you specify this clause at the top level of the syntax, the parameters become the parameters of the index and the default parameters for the index partitions. If you specify this clause as part of the *local_xmlindex_clause* clause, then you override any default parameters with parameters for the individual partition.

> **See Also:**
>
> *Oracle XML DB Developer's Guide* for the syntax and semantics of the *XMLIndex_parameters_clause*, as well as detailed information about the use of XMLIndex

bitmap_join_index_clause

Use the *bitmap_join_index_clause* to define a **bitmap join index**. A bitmap join index is defined on a single table. For an index key made up of dimension table columns, it stores the fact table rowids corresponding to that key. In a data warehousing environment, the table on which the index is defined is commonly referred to as a **fact table**, and the tables with which this table is joined are commonly referred to as **dimension tables**. However, a star schema is not a requirement for creating a join index.

ON

In the ON clause, first specify the fact table, and then inside the parentheses specify the columns of the dimension tables on which the index is defined.

FROM

In the FROM clause, specify the joined tables.

WHERE

In the WHERE clause, specify the join condition.

If the underlying fact table is partitioned, then you must also specify one of the *local_partitioned_index* clauses (see *local_partitioned_index*).

Restrictions on Bitmap Join Indexes

In addition to the restrictions on bitmap indexes in general (see BITMAP), the following restrictions apply to bitmap join indexes:

- You cannot create a bitmap join index on a temporary table.
- No table may appear twice in the FROM clause.
- You cannot create a function-based join index.
- The dimension table columns must be either primary key columns or have unique constraints.
- If a dimension table has a composite primary key, then each column in the primary key must be part of the join.
- You cannot specify the *local_partitioned_index* clause unless the fact table is partitioned.
- A bitmap join index definition can only reference columns with collation BINARY, USING_NLS_COMP, USING_NLS_SORT, or USING_NLS_SORT_CS. For any of these collations, index keys are collated and the join condition is evaluated using the BINARY collation. See *Oracle Database Globalization Support Guide* for more information.

> **Note:**
>
> *Oracle Database Data Warehousing Guide* for information on fact and dimension tables and on using bitmap indexes in a data warehousing environment

USABLE | UNUSABLE

You can specify the `USABLE` and `UNUSABLE` keywords:

- For an index, in the `CREATE INDEX` statement
- For an index partition, in the *`on_range_partitioned_table`*, *`on_list_partitioned_table`*, *`on_hash_partitioned_table`*, and *`on_comp_partitioned_table`* clauses
- For an index subpartition, in the *`index_subpartition_clause`*

For nonpartitioned indexes, specify `UNUSABLE` to create an index in an unusable state. An unusable index must be rebuilt, or dropped and re-created, before it can be used. Specify `USABLE` to create an index in a usable state. `USABLE` is the default.

For partitioned indexes, specify `USABLE` or `UNUSABLE` as follows:

- If you specify `UNUSABLE` for the index, then all index partitions are marked `UNUSABLE`.
- If you specify `USABLE` for the index, then all index partitions are marked `USABLE`.
- If you do not specify `USABLE` or `UNUSABLE` for the index, then all index partitions are marked `USABLE`. The exception is a local partial index. If you specify the `LOCAL` and `INDEXING PARTIAL` clauses, and do not specify `USABLE` or `UNUSABLE`, then each index partition is marked `USABLE` if the indexing property of its corresponding table partition is `ON`, or `UNUSABLE` if the indexing property of its corresponding table partition is `OFF`.

You can override the preceding conditions by specifying `USABLE` or `UNUSABLE` for a specific index partition.

If the underlying table is a composite-partitioned table, then the preceding conditions for index partitions and table partitions apply instead to index subpartitions and table subpartitions.

After you create a partitioned index, you can choose to rebuild specific index partitions or subpartitions to make them `USABLE`. Doing so can be useful if you want to maintain indexes only on some index partitions or subpartitions—for example, if you want to enable index access for new partitions but not for old partitions.

When an index, or some partitions or subpartitions of an index, are created `UNUSABLE`, no segment is allocated for the unusable object. The unusable index or index partition consumes no space in the database.

If an index, or some partitions or subpartitions of the index, are marked `UNUSABLE`, then the index will be considered as an access path by the optimizer only under the following circumstances: the optimizer must know at compile time which partitions are to be accessed, and all of those partitions to be accessed must be marked `USABLE`. Therefore, the query cannot contain any bind variables.

Restrictions on USABLE | UNUSABLE

The following restrictions apply when marking an index `USABLE` or `UNUSABLE`:

- You cannot specify this clause for an index on a temporary table.

- Unusable indexes or index partitions will still have a segment under the following conditions:
  - The index (or index partition) is owned by `SYS`, `SYSTEM`, `PUBLIC`, `OUTLN`, or `XDB`
  - The index (or index partition) is stored in dictionary-managed tablespaces
  - The global partitioned or nonpartitioned index on a partitioned table becomes unusable due to a partition maintenance operation

{ DEFERRED | IMMEDIATE } INVALIDATION

This clause lets you control when the database invalidates dependent cursors while creating the index. It has the same semantics here as for the `ALTER INDEX` statement. Refer to { DEFERRED | IMMEDIATE } INVALIDATION in the documentation on `ALTER INDEX` for the full semantics of this clause.

Examples

General Index Examples

Creating an Index: Example

The following statement shows how the sample index `ord_customer_ix` on the `customer_id` column of the sample table `oe.orders` was created:

```
CREATE INDEX ord_customer_ix
   ON orders (customer_id);
```

Compressing an Index: Example

To create the `ord_customer_ix_demo` index with the `COMPRESS` clause, you might issue the following statement:

```
CREATE INDEX ord_customer_ix_demo
   ON orders (customer_id, sales_rep_id)
   COMPRESS 1;
```

The index will compress repeated occurrences of `customer_id` column values.

Creating an Index in NOLOGGING Mode: Example

If the sample table `orders` had been created using a fast parallel load (so all rows were already sorted), then you could issue the following statement to quickly create an index.

```
/* Unless you first sort the table oe.orders, this example fails
   because you cannot specify NOSORT unless the base table is
   already sorted.
*/
CREATE INDEX ord_customer_ix_demo
   ON orders (order_mode)
   NOSORT
   NOLOGGING;
```

Creating a Cluster Index: Example

To create an index for the `personnel` cluster, which was created in "Creating a Cluster: Example", issue the following statement:

```
CREATE INDEX idx_personnel ON CLUSTER personnel;
```

No index columns are specified, because cluster indexes are automatically built on all the columns of the cluster key. For cluster indexes, all rows are indexed.

Creating an Index on an XMLType Table: Example

The following example creates an index on the area element of the `xwarehouses` table (created in "XMLType Table Examples"):

```
CREATE INDEX area_index ON xwarehouses e
   (EXTRACTVALUE(VALUE(e),'/Warehouse/Area'));
```

Such an index would greatly improve the performance of queries that select from the table based on, for example, the square footage of a warehouse, as shown in this statement:

```
SELECT e.getClobVal() AS warehouse
   FROM xwarehouses e
   WHERE EXISTSNODE(VALUE(e),'/Warehouse[Area>50000]') = 1;
```

> **See Also:**
>
> EXISTSNODE and VALUE

Function-Based Index Examples

The following examples show how to create and use function-based indexes.

Creating a Function-Based Index: Example

The following statement creates a function-based index on the `employees` table based on an uppercase evaluation of the `last_name` column:

```
CREATE INDEX upper_ix ON employees (UPPER(last_name));
```

See the "Prerequisites " for the privileges and parameter settings required when creating function-based indexes.

To increase the likelihood that Oracle Database will use the index rather than performing a full table scan, be sure that the value returned by the function is not null in subsequent queries. For example, this statement will use the index, unless some other condition exists that prevents the optimizer from doing so:

```
SELECT first_name, last_name
   FROM employees WHERE UPPER(last_name) IS NOT NULL
   ORDER BY UPPER(last_name);
```

Without the `WHERE` clause, Oracle Database may perform a full table scan.

In the next statements showing index creation and subsequent query, Oracle Database will use index `income_ix` even though the columns are in reverse order in the query:

```
CREATE INDEX income_ix
   ON employees(salary + (salary*commission_pct));

SELECT first_name||' '||last_name "Name"
   FROM employees
   WHERE (salary*commission_pct) + salary > 15000
   ORDER BY employee_id;
```

Creating a Function-Based Index on a LOB Column: Example

The following statement uses the `text_length` function to create a function-based index on a LOB column in the sample `pm` schema. See *Oracle Database PL/SQL Language Reference* for

the example that creates this function. The example selects rows from the sample table `print_media` where that `CLOB` column has fewer than 1000 characters.

```
CREATE INDEX src_idx ON print_media(text_length(ad_sourcetext));

SELECT product_id FROM print_media
   WHERE text_length(ad_sourcetext) < 1000
   ORDER BY product_id;

PRODUCT_ID
----------
      2056
      2268
      3060
      3106
```

Creating a Function-based Index on a Type Method: Example

This example entails an object type `rectangle` containing two number attributes: `length` and `width`. The `area()` method computes the area of the rectangle.

```
CREATE TYPE rectangle AS OBJECT
( length   NUMBER,
  width    NUMBER,
  MEMBER FUNCTION area RETURN NUMBER DETERMINISTIC
);

CREATE OR REPLACE TYPE BODY rectangle AS
  MEMBER FUNCTION area RETURN NUMBER IS
  BEGIN
   RETURN (length*width);
  END;
END;
```

Now, if you create a table `rect_tab` of type `rectangle`, you can create a function-based index on the `area()` method as follows:

```
CREATE TABLE rect_tab OF rectangle;
CREATE INDEX area_idx ON rect_tab x (x.area());
```

You can use this index efficiently to evaluate a query of the form:

```
SELECT * FROM rect_tab x WHERE x.area() > 100;
```

Using a Function-based Index to Define Conditional Uniqueness: Example

The following statement creates a unique function-based index on the `oe.orders` table that prevents a customer from taking advantage of promotion ID 2 ("blowout sale") more than once:

```
CREATE UNIQUE INDEX promo_ix ON orders
   (CASE WHEN promotion_id =2 THEN customer_id ELSE NULL END,
    CASE WHEN promotion_id = 2 THEN promotion_id ELSE NULL END);

INSERT INTO orders (order_id, order_date, customer_id, order_total, promotion_id)
   VALUES (2459, systimestamp, 106, 251, 2);
1 row created.

INSERT INTO orders (order_id, order_date, customer_id, order_total, promotion_id)
   VALUES (2460, systimestamp+1, 106, 110, 2);
insert into orders (order_id, order_date, customer_id, order_total, promotion_id)
*
ERROR at line 1:
ORA-00001: unique constraint (OE.PROMO_IX) violated
```

The objective is to remove from the index any rows where the promotion_id is not equal to 2. Oracle Database does not store in the index any rows where all the keys are NULL. Therefore, in this example, both customer_id and promotion_id are mapped to NULL unless promotion_id is equal to 2. The result is that the index constraint is violated only if promotion_id is equal to 2 for two rows with the same customer_id value.

Partitioned Index Examples

Creating a Range-Partitioned Global Index: Example

The following statement creates a global prefixed index cost_ix on the sample table sh.sales with three partitions that divide the range of costs into three groups:

```
CREATE INDEX cost_ix ON sales (amount_sold)
   GLOBAL PARTITION BY RANGE (amount_sold)
      (PARTITION p1 VALUES LESS THAN (1000),
       PARTITION p2 VALUES LESS THAN (2500),
       PARTITION p3 VALUES LESS THAN (MAXVALUE));
```

Creating a Hash-Partitioned Global Index: Example

The following statement creates a hash-partitioned global index cust_last_name_ix on the sample table sh.customers with four partitions:

```
CREATE INDEX cust_last_name_ix ON customers (cust_last_name)
  GLOBAL PARTITION BY HASH (cust_last_name)
  PARTITIONS 4;
```

Creating an Index on a Hash-Partitioned Table: Example

The following statement creates a local index on the category_id column of the hash_products partitioned table (which was created in "Hash Partitioning Example"). The STORE IN clause immediately following LOCAL indicates that hash_products is hash partitioned. Oracle Database will distribute the hash partitions between the tbs1 and tbs2 tablespaces:

```
CREATE INDEX prod_idx ON hash_products(category_id) LOCAL
   STORE IN (tbs_01, tbs_02);
```

The creator of the index must have quota on the tablespaces specified. See CREATE TABLESPACE for examples that create tablespaces tbs_01 and tbs_02.

Creating an Index on a Composite-Partitioned Table: Example

The following statement creates a local index on the composite_sales table, which was created in "Composite-Partitioned Table Examples". The STORAGE clause specifies default storage attributes for the index. However, this default is overridden for the five subpartitions of partitions q3_2000 and q4_2000, because separate TABLESPACE storage is specified.

The creator of the index must have quota on the tablespaces specified. See CREATE TABLESPACE for examples that create tablespaces tbs_02 and tbs_03.

```
CREATE INDEX sales_ix ON composite_sales(time_id, prod_id)
   STORAGE (INITIAL 1M)
   LOCAL
   (PARTITION q1_1998,
    PARTITION q2_1998,
    PARTITION q3_1998,
    PARTITION q4_1998,
    PARTITION q1_1999,
    PARTITION q2_1999,
    PARTITION q3_1999,
```

```
    PARTITION q4_1999,
    PARTITION q1_2000,
    PARTITION q2_2000
      (SUBPARTITION pq2001, SUBPARTITION pq2002,
       SUBPARTITION pq2003, SUBPARTITION pq2004,
       SUBPARTITION pq2005, SUBPARTITION pq2006,
       SUBPARTITION pq2007, SUBPARTITION pq2008),
    PARTITION q3_2000
      (SUBPARTITION c1 TABLESPACE tbs_02,
       SUBPARTITION c2 TABLESPACE tbs_02,
       SUBPARTITION c3 TABLESPACE tbs_02,
       SUBPARTITION c4 TABLESPACE tbs_02,
       SUBPARTITION c5 TABLESPACE tbs_02),
    PARTITION q4_2000
      (SUBPARTITION pq4001 TABLESPACE tbs_03,
       SUBPARTITION pq4002 TABLESPACE tbs_03,
       SUBPARTITION pq4003 TABLESPACE tbs_03,
       SUBPARTITION pq4004 TABLESPACE tbs_03)
);
```

Bitmap Index Examples

The following creates a bitmap index on the table `oe.hash_products`, which was created in "Hash Partitioning Example":

```
CREATE BITMAP INDEX product_bm_ix
   ON hash_products(list_price)
   LOCAL(PARTITION ix_p1 TABLESPACE tbs_01,
         PARTITION ix_p2,
         PARTITION ix_p3 TABLESPACE tbs_02,
         PARTITION ix_p4 TABLESPACE tbs_03)
   TABLESPACE tbs_04;
```

Because `hash_products` is a partitioned table, the bitmap join index must be locally partitioned. In this example, the user must have quota on tablespaces specified. See CREATE TABLESPACE for examples that create tablespaces `tbs_01`, `tbs_02`, `tbs_03`, and `tbs_04`.

The next series of statements shows how one might create a bitmap join index on a fact table using a join with a dimension table.

```
CREATE TABLE hash_products
    ( product_id          NUMBER(6)
    , product_name        VARCHAR2(50)
    , product_description VARCHAR2(2000)
    , category_id         NUMBER(2)
    , weight_class        NUMBER(1)
    , warranty_period     INTERVAL YEAR TO MONTH
    , supplier_id         NUMBER(6)
    , product_status      VARCHAR2(20)
    , list_price          NUMBER(8,2)
    , min_price           NUMBER(8,2)
    , catalog_url         VARCHAR2(50)
    , CONSTRAINT          pk_product_id PRIMARY KEY (product_id)
    , CONSTRAINT          product_status_lov_demo
                          CHECK (product_status in ('orderable'
                                                    ,'planned'
                                                    ,'under development'
                                                    ,'obsolete')
 ) )
 PARTITION BY HASH (product_id)
 PARTITIONS 5
 STORE IN (example);
```

```
CREATE TABLE sales_quota
    ( product_id          NUMBER(6)
    , customer_name       VARCHAR2(50)
    , order_qty           NUMBER(6)
  ,CONSTRAINT u_product_id UNIQUE(product_id)
 );

CREATE BITMAP INDEX product_bm_ix
   ON hash_products(list_price)
   FROM hash_products h, sales_quota s
   WHERE h.product_id = s.product_id
   LOCAL(PARTITION ix_p1 TABLESPACE example,
         PARTITION ix_p2,
         PARTITION ix_p3 TABLESPACE example,
         PARTITION ix_p4,
         PARTITION ix_p5 TABLESPACE example)
   TABLESPACE example;
```

Indexes on Nested Tables: Example

The sample table `pm.print_media` contains a nested table column `ad_textdocs_ntab`, which is stored in storage table `textdocs_nestedtab`. The following example creates a unique index on storage table `textdocs_nestedtab`:

```
CREATE UNIQUE INDEX nested_tab_ix
      ON textdocs_nestedtab(NESTED_TABLE_ID, document_typ);
```

Including pseudocolumn `NESTED_TABLE_ID` ensures distinct rows in nested table column `ad_textdocs_ntab`.

Indexing on Substitutable Columns: Examples

You can build an index on attributes of the declared type of a substitutable column. In addition, you can reference the subtype attributes by using the appropriate `TREAT` function. The following example uses the table `books`, which is created in "Substitutable Table and Column Examples". The statement creates an index on the `salary` attribute of all employee authors in the `books` table:

```
CREATE INDEX salary_i
   ON books (TREAT(author AS employee_t).salary);
```

The target type in the argument of the `TREAT` function must be the type that added the attribute being referenced. In the example, the target of `TREAT` is `employee_t`, which is the type that added the `salary` attribute.

If this condition is not satisfied, then Oracle Database interprets the `TREAT` function as any functional expression and creates the index as a function-based index. For example, the following statement creates a function-based index on the `salary` attribute of part-time employees, assigning nulls to instances of all other types in the type hierarchy.

```
CREATE INDEX salary_func_i ON persons p
   (TREAT(VALUE(p) AS part_time_emp_t).salary);
```

You can also build an index on the type-discriminant column underlying a substitutable column by using the `SYS_TYPEID` function.

> **Note:**
>
> Oracle Database uses the type-discriminant column to evaluate queries that involve the `IS OF type` condition. The cardinality of the typeid column is normally low, so Oracle recommends that you build a bitmap index in this situation.

The following statement creates a bitmap index on the typeid of the author column of the books table:

```
CREATE BITMAP INDEX typeid_i ON books (SYS_TYPEID(author));
```

> **See Also:**
>
> - *Oracle Database PL/SQL Language Reference* to see the creation of the type hierarchy underlying the `books` table
> - the functions TREAT and SYS_TYPEID and the condition "IS OF *type* Condition "

CREATE INDEXTYPE

Purpose

Use the `CREATE INDEXTYPE` statement to create an **indextype**, which is an object that specifies the routines that manage a domain (application-specific) index. Indextypes reside in the same namespace as tables, views, and other schema objects. This statement binds the indextype name to an implementation type, which in turn specifies and refers to user-defined index functions and procedures that implement the indextype.

> **See Also:**
>
> *Oracle Database Data Cartridge Developer's Guide* for more information on implementing indextypes

Prerequisites

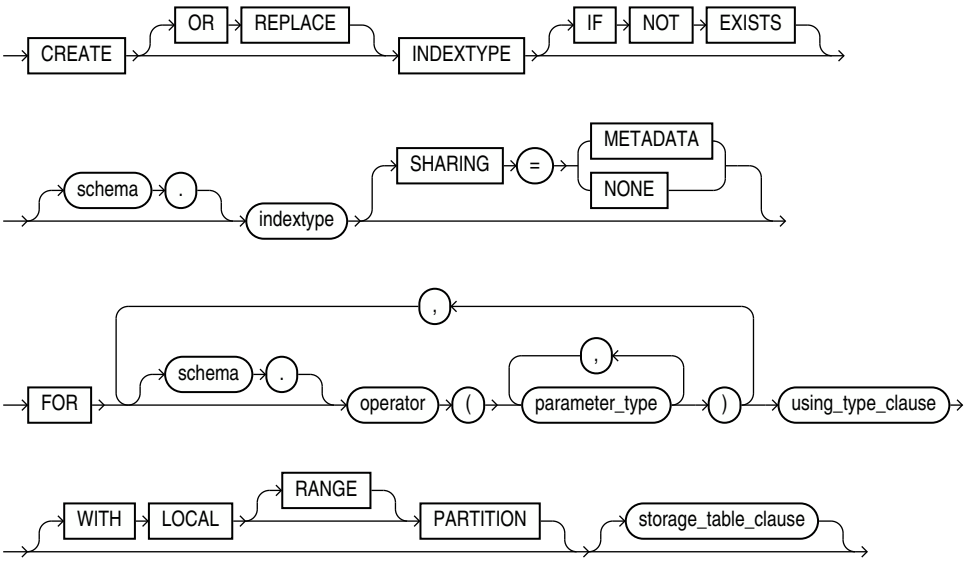
To create an indextype in your own schema, you must have the `CREATE INDEXTYPE` system privilege. To create an indextype in another schema, you must have the `CREATE ANY INDEXTYPE` system privilege. In either case, you must have the `EXECUTE` object privilege on the implementation type and the supported operators.

An indextype supports one or more operators, so before creating an indextype, you must first design the operator or operators to be supported and provide functional implementation for those operators.

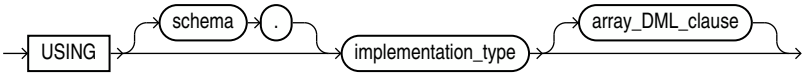
> **See Also:**
>
> CREATE OPERATOR

Syntax

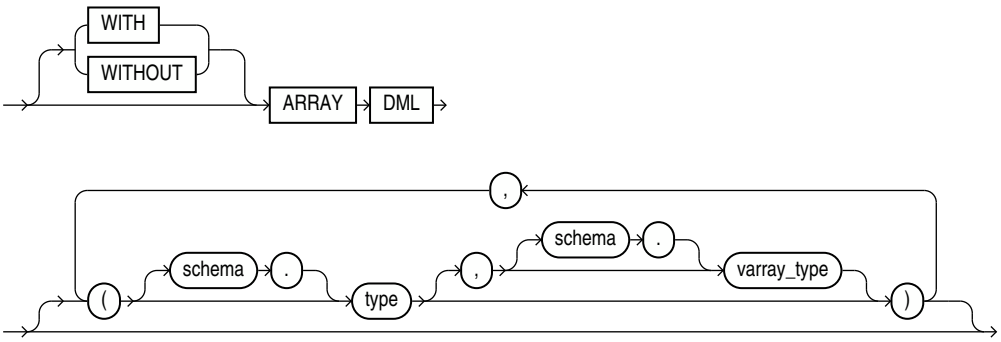
***create_indextype*::=**



***using_type_clause*::=**



***array_DML_clause*::=**



***storage_table_clause*::=**

Semantics

OR REPLACE

Specify `OR REPLACE` to re-create the indextype if it already exists. You can use this clause to change the definition of an existing indextype 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 indextype does not exist, a new indextype is created at the end of the statement.
- If the indextype exists, this is the indextype 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 name of the schema in which the indextype resides. If you omit *`schema`*, then Oracle Database creates the indextype in your own schema.

indextype

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

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.

FOR Clause

Use the `FOR` clause to specify the list of operators supported by the indextype.

- For *schema*, specify the schema containing the operator. If you omit *schema*, then Oracle assumes the operator is in your own schema.
- For *operator*, specify the name of the operator supported by the indextype.

  All the operators listed in this clause must be valid operators.
- For *parameter_type*, list the types of parameters to the operator.

using_type_clause

The USING clause lets you specify the type that provides the implementation for the new indextype.

For *implementation_type*, specify the name of the type that implements the appropriate Oracle Data Cartridge Interface (ODCI).

- You must specify a valid type that implements the routines in the ODCI.
- The implementation type must reside in the same schema as the indextype.

> **See Also:**
>
> *Oracle Database Data Cartridge Developer's Guide* for additional information on this interface

WITH LOCAL PARTITION

Use this clause to indicate that the indextype can be used to create local domain indexes on range-, list-, hash-, and interval-partitioned tables. You use this clause in combination with the *storage_table_clause* in several ways (see *storage_table_clause*).

- The recommended method is to specify WITH LOCAL PARTITION WITH SYSTEM MANAGED STORAGE TABLES. This combination uses *system-managed* storage tables, which are the preferred storage management, and lets you create local domain indexes on range-, list-, hash-, and interval-partitioned tables. In this case the RANGE keyword is optional and ignored, because it is no longer needed if you specify WITH LOCAL PARTITION WITH SYSTEM MANAGED STORAGE TABLES.
- You can specify WITH LOCAL RANGE PARTITION (including the RANGE keyword) and omit the *storage_table* clause. Local domain indexes on range-partitioned tables are supported with *user-managed* storage tables for backward compatibility. Oracle does not recommend this combination because it uses the less efficient user-managed storage tables.

If you omit this clause entirely, then you cannot subsequently use this indextype to create a local domain index on a range, list-, hash-, or interval-partitioned table.

storage_table_clause

Use this clause to specify how storage tables and partition maintenance operations for indexes built on this indextype are managed:

- Specify WITH SYSTEM MANAGED STORAGE TABLES to indicate that the storage of statistics data is to be managed by the system. The type you specify in *statistics_type* should be storing the statistics related information in tables that are maintained by the system. Also, the indextype you specify must already have been created or altered to support the WITH SYSTEM MANAGED STORAGE TABLES clause.

- Specify `WITH USER MANAGED STORAGE TABLES` to indicate that the tables that store the user-defined statistics will be managed by the user. This is the default behavior.

> **See Also:**
>
> *Oracle Database Data Cartridge Developer's Guide* for more information about storage tables for domain indexes

array_DML_clause

Use this clause to let the indextype support the array interface for the `ODCIIndexInsert` method.

type* and *varray_type

If the data type of the column to be indexed is a user-defined object type, then you must specify this clause to identify the varray *`varray_type`* that Oracle should use to hold column values of *`type`*. If the indextype supports a list of types, then you can specify a corresponding list of varray types. If you omit *`schema`* for either *`type`* or *`varray_type`*, then Oracle assumes the type is in your own schema.

If the data type of the column to be indexed is a built-in system type, then any varray type specified for the indextype takes precedence over the ODCI types defined by the system.

> **See Also:**
>
> *Oracle Database Data Cartridge Developer's Guide* for more information on the ODCI array interface

Examples

Creating an Indextype: Example

The following statement creates an indextype named `position_indextype` and specifies the `position_between` operator that is supported by the indextype and the `position_im` type that implements the index interface. Refer to "Using Extensible Indexing " for an extensible indexing scenario that uses this indextype:

```
CREATE INDEXTYPE position_indextype
   FOR position_between(NUMBER, NUMBER, NUMBER)
   USING position_im;
```

CREATE INMEMORY JOIN GROUP

Purpose

Use the `CREATE INMEMORY JOIN GROUP` statement to create a join group, which is an object that specifies frequently joined columns from the same table or different tables. Such columns typically contain values of compatible data types that fall in similar ranges. When you create a join group, Oracle Database stores special metadata for the columns in the global dictionary, which enables the database to optimize join queries for the columns. In order to achieve this

optimization, the table columns must be populated in the In-Memory Column Store (IM column store).

Creating a join group for tables causes the current In-Memory contents of these tables to be invalidated. Subsequent repopulation causes the In-Memory Compression Units (IMCUs) of the tables to be re-encoded with the global dictionary. Thus, Oracle recommends that you first create the join group, and then populate the tables.

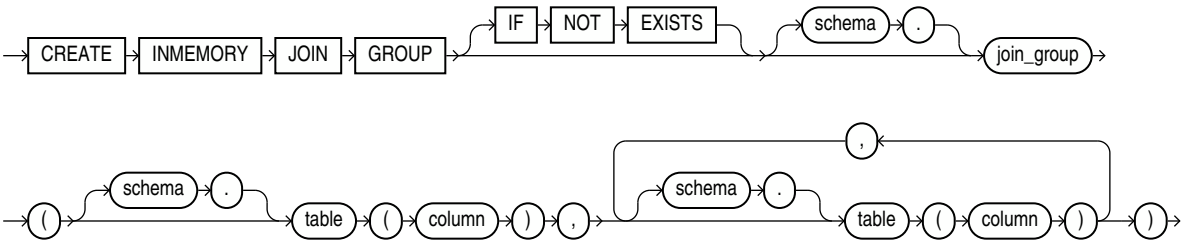
> **See Also:**
>
> - ALTER INMEMORY JOIN GROUP and DROP INMEMORY JOIN GROUP
> - *Oracle Database In-Memory Guide* for more information on join groups

Prerequisites

To create a join group in another user's schema, or to include in the join group a column in a table in another user's schema, you must have the `CREATE ANY TABLE` system privilege.

Syntax

***create_inmemory_join_group*::=**



Semantics

IF NOT EXISTS

Specifying `IF NOT EXISTS` has the following effects:

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

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

schema

Specify the schema to contain the join group. If you omit *`schema`*, then the database creates the join group in your own schema.

join_group

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

schema

Specify the schema of the table that contains a column to be included in the join group If you omit *schema*, then Oracle Database assumes the table is in your own schema.

table

Specify the name of the table that contains a column to be included in the join group.

column

Specify the name of a column to be included in the join group. A join group can contain columns in the same table or different tables.

Restrictions on Join Groups

The following restrictions apply to join groups:

- A join group must contain at least 1 column.
- A join group can contain at most 255 columns.
- A table column can be a member of at most one join group.
- Oracle Active Data Guard does not support join groups.

Examples

The following statement creates a join group named `prod_id1` in the `oe` schema. Both tables involved in this join group reside in the `oe` schema.

```
CREATE INMEMORY JOIN GROUP prod_id1
  (inventories(product_id), order_items(product_id));
```

The following statement creates a join group named `prod_id2` in the `oe` schema. The table `inventories` resides in the `oe` schema and the table `online_media` resides in the `pm` schema.

```
CREATE INMEMORY JOIN GROUP prod_id2
  (inventories(product_id), pm.online_media(product_id));
```

CREATE JAVA

Purpose

Use the `CREATE JAVA` statement to create a schema object containing a Java source, class, or resource.

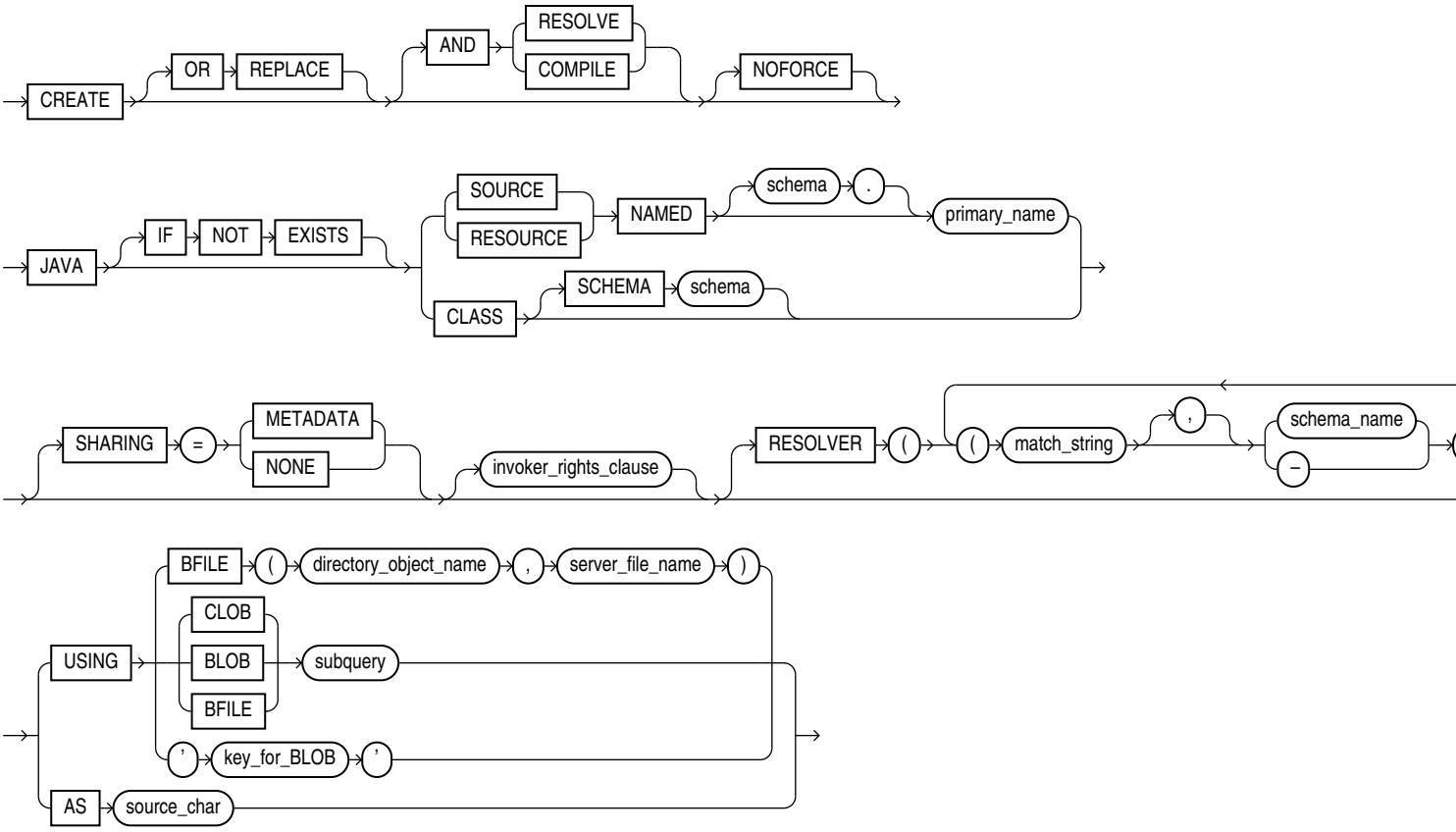
> **See Also:**
>
> - *Oracle Database Java Developer's Guide* for Java concepts and information about Java stored procedures
> - *Oracle Database JDBC Developer's Guide* for information on JDBC

Prerequisites

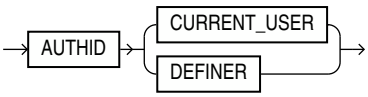
To create or replace a schema object containing a Java source, class, or resource in your own schema, you must have `CREATE PROCEDURE` system privilege. To create or replace such a schema object in another user's schema, you must have `CREATE ANY PROCEDURE` system privilege.

Syntax

***create_java*::=**



***invoker_rights_clause*::=**



Semantics

OR REPLACE

Specify `OR REPLACE` to re-create the schema object containing the Java class, source, or resource if it already exists. Use this clause to change the definition of an existing object without dropping, re-creating, and regranting object privileges previously granted.

If you redefine a Java schema object and specify `RESOLVE` or `COMPILE`, then Oracle Database recompiles or resolves the object. Whether or not the resolution or compilation is successful, the database invalidates classes that reference the Java schema object.

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

> **See Also:**
>
> ALTER JAVA for additional information

IF NOT EXISTS

Specifying `IF NOT EXISTS` has the following effects:

- If the object does not exist, a new object is created at the end of the statement.
- If the object exists, this is the object 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.`

RESOLVE | COMPILE

`RESOLVE` and `COMPILE` are synonymous keywords. They specify that Oracle Database should attempt to resolve the Java schema object that is created if this statement succeeds.

- When applied to a class, resolution of referenced names to other class schema objects occurs.
- When applied to a source, source compilation occurs.

Restriction on RESOLVE and COMPILE

You cannot specify these keywords for a Java resource.

NOFORCE

Specify `NOFORCE` to roll back the results of this `CREATE` command if you have specified either `RESOLVE` or `COMPILE` and the resolution or compilation fails. If you do not specify this option, then Oracle Database takes no action if the resolution or compilation fails, and the created schema object remains.

JAVA SOURCE Clause

Specify `JAVA SOURCE` to load a Java source file.

JAVA CLASS Clause

Specify `JAVA CLASS` to load a Java class file.

JAVA RESOURCE Clause

Specify `JAVA RESOURCE` to load a Java resource file.

NAMED Clause

The `NAMED` clause is required for a Java source or resource. The *`primary_name`* must be enclosed in double quotation marks and its length must not exceed 4000 bytes in the database character set.

- For a Java source, this clause specifies the name of the schema object in which the source code is held. A successful `CREATE JAVA SOURCE` statement will also create additional schema objects to hold each of the Java classes defined by the source.
- For a Java resource, this clause specifies the name of the schema object to hold the Java resource.

Use double quotation marks to preserve a lower- or mixed-case *`primary_name`*.

If you do not specify *`schema`*, then Oracle Database creates the object in your own schema.

Restrictions on NAMED Java Classes

The `NAMED` clause is subject to the following restrictions:

- You cannot specify `NAMED` for a Java class.
- The *`primary_name`* cannot contain a database link.

SCHEMA Clause

The `SCHEMA` clause applies only to a Java class. This optional clause specifies the schema in which the object containing the Java file will reside. If you do not specify this clause, then Oracle Database creates the object in your own schema.

SHARING

This clause applies only when creating a Java schema object in an application root. 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. To determine how the Java schema object is shared, specify one of the following sharing attributes:

- `METADATA` - A metadata link shares the Java schema object's metadata, but its data is unique to each container. This type of Java schema object is referred to as a **metadata-linked application common object**.
- `NONE` - The Java schema object is not shared.

If you omit this clause, then the database uses the value of the `DEFAULT_SHARING` initialization parameter to determine the sharing attribute of the Java schema object. If the `DEFAULT_SHARING` initialization parameter does not have a value, then the default is `METADATA`.

You cannot change the sharing attribute of a Java schema object after it is created.

> **See Also:**
>
> - *Oracle Database Reference* for more information on the `DEFAULT_SHARING` initialization parameter
> - *Oracle Database Administrator's Guide* for complete information on creating application common objects

invoker_rights_clause

Use the *invoker_rights_clause* to indicate whether the methods of the class execute with the privileges and in the schema of the user who owns the class or with the privileges and in the schema of `CURRENT_USER`.

This clause also determines how Oracle Database resolves external names in queries, DML operations, and dynamic SQL statements in the member functions and procedures of the type.

AUTHID CURRENT_USER

`CURRENT_USER` indicates that the methods of the class execute with the privileges of `CURRENT_USER`. This clause is the default and creates an **invoker-rights class**.

This clause also specifies that external names in queries, DML operations, and dynamic SQL statements resolve in the schema of `CURRENT_USER`. External names in all other statements resolve in the schema in which the methods reside.

AUTHID DEFINER

`DEFINER` indicates that the methods of the class execute with the privileges of the owner of the schema in which the class resides, and that external names resolve in the schema where the class resides. This clause creates a **definer-rights class**.

> **See Also:**
>
> - *Oracle Database Java Developer's Guide*
> - *Oracle Database PL/SQL Language Reference* for information on how `CURRENT_USER` is determined

RESOLVER Clause

The `RESOLVER` clause lets you specify a mapping of the fully qualified Java name to a Java schema object, where:

- *match_string* is either a fully qualified Java name, a wildcard that can match such a Java name, or a wildcard that can match any name.
- *schema_name* designates a schema to be searched for the corresponding Java schema object.
- A dash (-) as an alternative to *schema_name* indicates that if *match_string* matches a valid Java name, Oracle Database can leave the name unresolved. The resolution succeeds, but the name cannot be used at run time by the class.

This mapping is stored with the definition of the schema objects created in this command for use in later resolutions (either implicit or in explicit `ALTER JAVA ... RESOLVE` statements).

USING Clause

The `USING` clause determines a sequence of character data (`CLOB` or `BFILE`) or binary data (`BLOB` or `BFILE`) for the Java class or resource. Oracle Database uses the sequence of characters to define one file for a Java class or resource, or one source file and one or more derived classes for a Java source.

BFILE Clause

Specify the directory and filename of a previously created file on the operating system (*`directory_object_name`*) and server file (*`server_file_name`*) containing the sequence. `BFILE` is usually interpreted as a character sequence by `CREATE JAVA SOURCE` and as a binary sequence by `CREATE JAVA CLASS` or `CREATE JAVA RESOURCE`.

CLOB | BLOB | BFILE *subquery*

Specify a subquery that selects a single row and column of the type specified (`CLOB`, `BLOB`, or `BFILE`). The value of the column makes up the sequence of characters.

> **Note:**
>
> In earlier releases, the `USING` clause implicitly supplied the keyword `SELECT`. This is no longer the case. However, the subquery without the keyword `SELECT` is still supported for backward compatibility.

key_for_BLOB

The *`key_for_BLOB`* clause supplies the following implicit query:

```
SELECT LOB FROM CREATE$JAVA$LOB$TABLE
   WHERE NAME = 'key_for_BLOB';
```

Restriction on the *key_for_BLOB* Clause

For you to use this case, the table `CREATE$JAVA$LOB$TABLE` must exist in the current schema and must have a column LOB of type `BLOB` and a column `NAME` of type `VARCHAR2`.

AS *source_char*

Specify a sequence of characters for a Java source.

Examples

Creating a Java Class Object: Example

The following statement creates a schema object containing a Java class using the name found in a Java binary file:

```
CREATE JAVA CLASS USING BFILE (java_dir, 'Agent.class')
/
```

This example assumes the directory object `java_dir`, which points to the operating system directory containing the Java class `Agent.class`, already exists. In this example, the name of the class determines the name of the Java class schema object.

Creating a Java Source Object: Example

The following statement creates a Java source schema object:

```
CREATE JAVA SOURCE NAMED "Welcome" AS
   public class Welcome {
      public static String welcome() {
         return "Welcome World";   } }
/
```

Creating a Java Resource Object: Example

The following statement creates a Java resource schema object named `apptext` from a `bfile`:

```
CREATE JAVA RESOURCE NAMED "appText"
   USING BFILE (java_dir, 'textBundle.dat')
/
```

CREATE JSON RELATIONAL DUALITY VIEW

Purpose

JSON-relational duality views expose data in relational tables as JSON documents. The documents are materialized on demand, not stored. Duality views give your data a conceptual and an operational duality as it is organized both relationally and hierarchically. You can base different duality views on data stored in one or more of the same tables, providing different JSON hierarchies over the same, shared data. This means that applications can access (create, query, modify) the same data as a collection of JSON documents or as a set of related tables and columns, and both approaches can be employed at the same time.

A flex column in a table underlying a JSON-relational duality view lets you add and redefine fields of the document object produced by that table. This provides a certain kind of schema flexibility to a duality view, and to the documents it supports. For more information on flex columns in a table underlying a JSON-relational duality view see JSON Data Stored in JSON-Relational Duality Views of the *JSON-Relational Duality Developer's Guide*

You define a duality view against a set of tables related by primary key (PK), foreign key (FK) or unique key constraints (UK). The following rules apply:

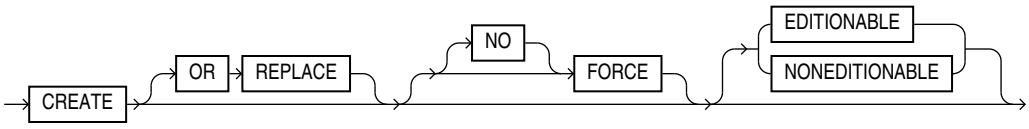
- The primary or unique key constraints must be declared in the database but need not be enforced (can be `RELY` constraints). Foreign key constraints are not required to be declared in the database.
- The relationships type can be 1-to-1, 1-to-N and N-to-M (using a mapping table with two FKs). The N-to-M relationship can be thought of as the combination of 1-to-N and 1-to-1 relationship
- Columns of two or more tables with 1-to-1 or N-to-1 relationships can be merged into the same JSON object via `UNNEST`. Otherwise a nested JSON object is created.
- Tables with a 1-to-N relationship create a nested JSON array.
- A duality view has only one column of type `JSON`.
- Each row in the duality view is one JSON object, which is typically a hierarchy of nested objects and arrays.
- Each application object is built from values originating from one or multiple rows from the underlying tables of that view. Typically, each table contributes to one (nested) JSON object.

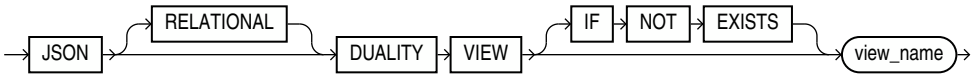
> **Note:**
>
> The SQL data types allowed for a column in a table underlying a duality view are `BINARY_DOUBLE`, `BINARY_FLOAT`, `BLOB`, `BOOLEAN`, `CHAR`, `CLOB`, `DATE`, `JSON`, `INTERVAL DAY TO SECOND`, `INTERVAL YEAR TO MONTH`, `NCHAR`, `NCLOB`, `NUMBER`, `NVARCHAR2`, `VARCHAR2`, `RAW`, `TIMESTAMP`, `TIMESTAMP WITH TIME ZONE`, and `VECTOR`. An error is raised if you specify any other column data type.

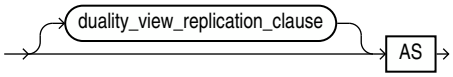
> **See Also:**
>
> *JSON-Relational Duality Developer's Guide*

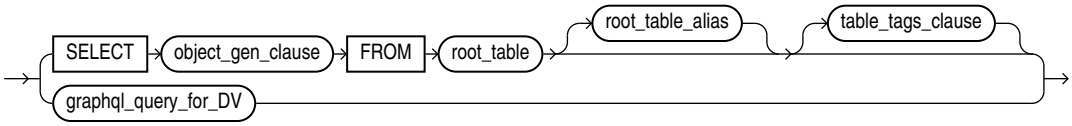
Syntax

***create_json_relational_duality_view*::=**

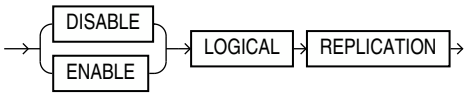




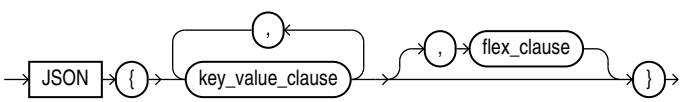




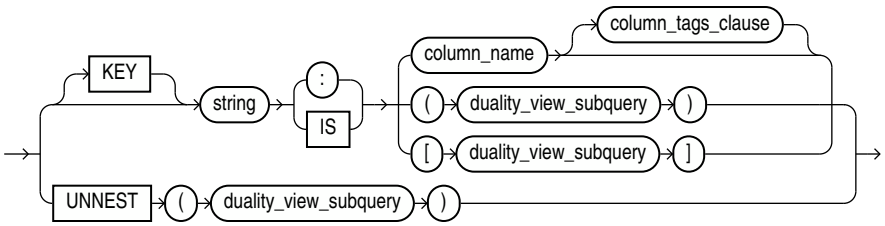
duality_view_replication_clause



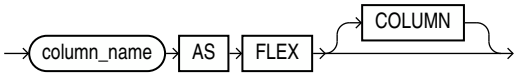
***object_gen_clause*::=**



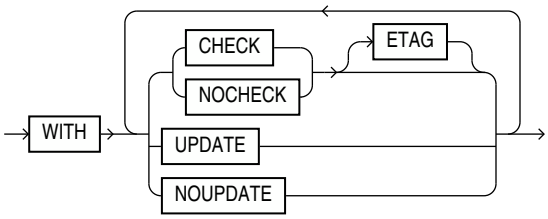
***key_value_clause*::=**



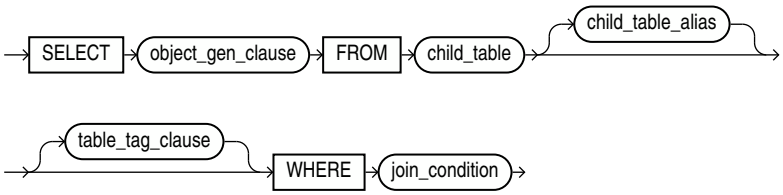
***flex_clause*::=**



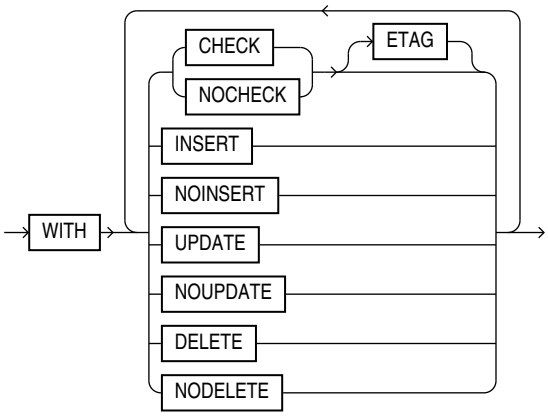
***column_tags_clause*::=**



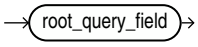
***duality_view_subquery*::=**



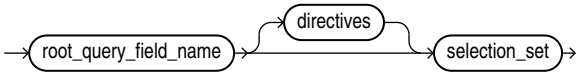
***table_tags_clause*::=**



***graphql_query_for_DV*::=**

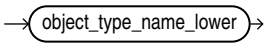


***root_query_field*::=**

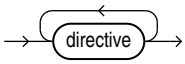


(*directives*::=, *selection_set*::=)

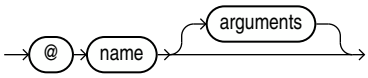
***root_query_field_name*::=**



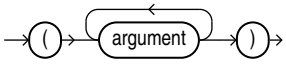
***directives*::=**



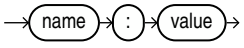
***directive*::=**



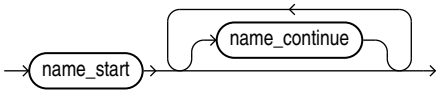
***arguments*::=**



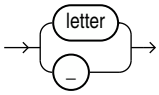
***argument*::=**



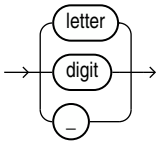
***name*::=**



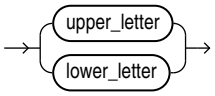
***name_start*::=**



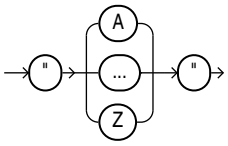
***name_continue*::=**



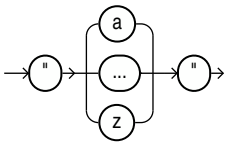
***letter*::=**



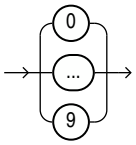
***upper_letter*::=**



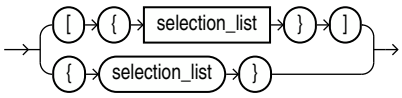
***lower_letter*::=**



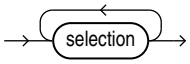
***digit*::=**



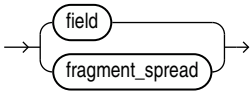
***selection_set*::=**



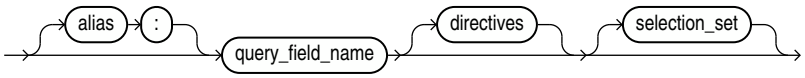
***selection_list*::=**



***selection*::=**

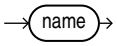


***field*::=**

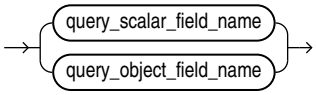


(*directives*::=, *selection_set*::=)

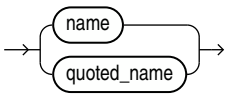
***alias*::=**



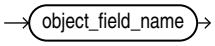
***query_field_name*::=**



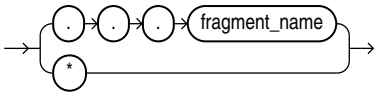
***query_scalar_field_name*::=**



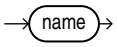
***query_object_field_name*::=**



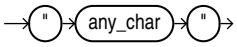
***fragment_spread*::=**



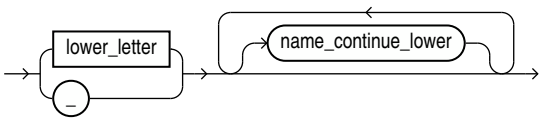
fragment_name



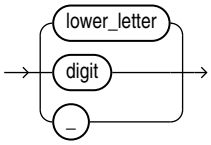
***quoted_name*::=**



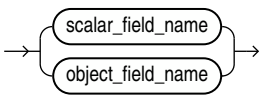
***lower_case_name*::=**



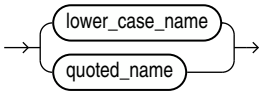
***name_continue_lower*::=**



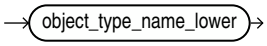
***field_name*::=**



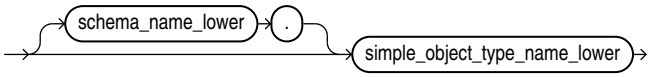
***scalar_field_name*::=**



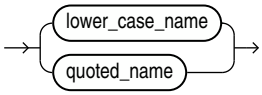
***object_field_name*::=**



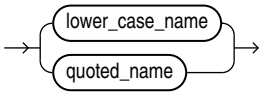
***object_field_name_lower*::=**



***schema_name_lower*::=**



***simple_object_type_name_lower*::=**



Semantics

The JSON realtional duality view has only one column of data type `JSON`. The column contains the JSON object which is a representation of a application object. The column name is always `DATA`.

The duality view is read-only by default. This means that the following annotations are in effect: `NOINSERT`, `NODELETE`, `NOUPDATE`.

OR REPLACE

Specify `OR REPLACE` to re-create the view if it already exists. You can use this clause to change the definition of an existing view 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 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.

You can have only one of `OR REPLACE` or `IF NOT EXISTS` in a statement at a time. Using both in the 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

duality_view_replication_clause

To enable logical replication on a duality view use `CREATE JSON RELATIONAL DUALITY VIEW ENABLE LOGICAL REPLICATION`.

To disable logical replication on a duality view use `CREATE JSON RELATIONAL DUALITY VIEW DISABLE LOGICAL REPLICATION`

> **Note:**
>
> On a multi instance RAC database, you must run the `ALTER SYSTEM ENABLE RAC TWO_STAGE ROLLING UPDATES ALL` DDL, before you can enable or disable logical replication.
>
> You must run `ALTER SYSTEM ENABLE RAC TWO_STAGE ROLLING UPDATES ALL` after patching all the RAC instances.
>
> After you run `ALTER SYSTEM ENABLE RAC TWO_STAGE ROLLING UPDATES ALL` you cannot perform an online downgrade (unpatch) of your RAC database to `DBRU23.5` or lower. You must take a downtime.
>
> On a single instance database, you do not need to run `ALTER SYSTEM ENABLE RAC TWO_STAGE ROLLING UPDATES ALL`.

root_table

`root_table` refers to the top level table which the duality view is defined on. It is the only table specified in the `FROM` clause of the top level `SELECT` statement.

key_value_clause

You must have one key named `_id` that points to the column(s) that identify the JSON document, usually a primary-key column.

See Document-Identifier Field for Duality Views of the *JSON-Relational Duality Developer's Guide* .

table_tags_clause

You can mark the view as updatable using the following keyword inside a `WITH` clause:

- `WITH INSERT`
- `WITH UPDATE`

- `WITH DELETE`

You can combine keywords together without commas, for example: `WITH INSERT UPDATE`

column_tags_clause

You can mark individual columns with `WITH UPDATE` or `WITH NOUPDATE`. This supercedes table-level annotation.

Column Properties for Updatability

If the `FROM` clause is marked with such keywords, then this sets the default for all columns of the table in the `FROM` clause. You can change the default setting on an individual column. If a the `FROM` clause is specified as `WITH` (`INSERT`, `UPDATE`, `DELETE`) and a column overrides this default with `NOUPDATE`, then updates are not allowed.

Column Properties for ETAGs

Individual columns as well as a `FROM` clause can be specified to take part in the `CHECK ETAG` calculation or not. An `ETAG` is a hash value for all the columns' values in one JSON object and is used to detect changes. A column without `ETAG` can be changed without this change impacting other operations. By default all columns participate in `ETAG` calculation. Using `NOCHECK ETAG` a column can be excluded from `ETAG` calculation.

graphql_query_for DV

`graphql_query_for_DV` is a special kind of shorthand query operation definition in GraphQL.

- The *`root_query_field`* is the single top-level selection field of this shorthand query.

  For brevity, *`graphql_query_for_dv`* omits the pair of curly brackets of the top-level*`selection_set`* of a general shorthand query operation.
- *`selection_set`* syntax augments the selection set defined in the GraphQL specification with the option of optional square brackets around the selection list.
- *`selection`* in *`selection_list`* can be only *`field`* or *`fragment_spread`* .
- *`field directives`* : conform to the GraphQL specification. Only supported custom directives are allowed. @skip and @include are NOT supported.
- *`argument`* conforms to the GraphQL specification.
- *`root_query_field_name`* corresponds to the root table.
- *`name`* has the same syntax as the GraphQL specification.
- *`quoted_name`*: The field names in a GraphQL query for DV allow quoted and un-quoted names. As a convention, un-quoted field names are in lower case only. *`any_char`* is any character allowed in a quoted identifier in SQL .
- *`scalar_field_name`* corresponds to a column name of a table.
- *`object_field_name`* corresponds to a related table name. In addition you can use quoted names, and fully qualified table names with dot-concatenation.

Examples

> **See Also:**
>
> *Introduction To Car-Racing Duality Views Example* of the *JSON-Relational Duality Developer's Guide.*

Example 1: Create a Duality View of Orders

The following example is a view of an orders view object `ORDERS_OV` with the following information:

- Order information such as order status from the `Orders` table
- A `CustomerInfo` singleton descendant consisting of customer details from the `Customer` table
- An `OrderItems` array descendant consisting of a list of order items from the `OrderItems` table.
- Each order item, in turn, consists of `ItemInfo` and `ShipmentInfo` singletons from the `Products` and `Shipment` tables respectively.

```
CREATE OR REPLACE JSON RELATIONAL DUALITY VIEW ORDERS_OV AS
SELECT JSON { 'OrderId'   : ord.order_id,
              'OrderTime' : ord.order_datetime,
             'OrderStatus' : ord.order_status,
              'CustomerInfo' :
              (SELECT JSON{'CustomerId'    : cust.customer_id,
                                  'CustomerName'  : cust.full_name,
                                  'CustomerEmail' :
cust.email_address }

               FROM CUSTOMERS cust
                     WHERE cust.customer_id = ord.customer_id),
             'OrderItems' : (SELECT JSON_ARRAYAGG(
                   JSON { 'OrderItemId' : oi.line_item_id,
                                       'Quantity'    :
oi.quantity,
                                       'ProductInfo' : <subquery from product>
                                    'ShipmentInfo' : <subquery from
shipments>)

                            })
                            FROM ORDER_ITEMS oi
                            WHERE ord.order_id = oi.order_id)
}
FROM ORDERS ord;
```

Example 2: Create an Updatable View

To make the view updatable, one has to add `INSERT` or `UPDATE` or `DELETE` or any combination of these to either the `FROM` clause or individual column. The following allows to update the order, only read the customer and insert and update (not delete) order items.

```
CREATE OR REPLACE JSON RELATIONAL DUALITY VIEW ORDERS_OV AS
SELECT JSON { 'OrderId'     : ord.order_id,
              'OrderTime'.  : ord.order_datetime,
```

```
              'OrderStatus' : ord.order_status,
               'CustomerInfo' :
              (SELECT JSON{'CustomerId'    : cust.customer_id,
                           'CustomerName'  : cust.full_name,
                           'CustomerEmail' : cust.email_address WITH
NOCHECK}
                FROM CUSTOMERS c WITH CHECK
                      WHERE cust.customer_id = ord.customer_id),
               'OrderItems' : (SELECT JSON_ARRAYAGG(
                     JSON { 'OrderItemId' : oi.line_item_id,
                                          'Quantity'    : oi.quantity,
                                          'ProductInfo' : <subquery from product>
                                          'ShipmentInfo' : <subquery from
shipments>)

                               })
                               FROM ORDER_ITEMS oi WITH INSERT, UPDATE
                               WHERE ord.order_id = oi.order_id)
}
FROM ORDERS ord WITH INSERT UPDATE DELETE;
```