15

SQL Statements: CREATE SEQUENCE to DROP CLUSTER

This chapter contains the following SQL statements:

CREATE SEQUENCE

Purpose

A sequence is a database object used to produce unique integers, which are commonly used to populate a synthetic primary key column in a table. The sequence number always increases, typically by 1, and each new entry is placed on the right-most leaf block of the index.

Use the `CREATE SEQUENCE` statement to create a sequence to automatically generate primary key values.

When a sequence number is generated, the sequence is incremented, independent of the transaction committing or rolling back. If two users concurrently increment the same sequence, then the sequence numbers each user acquires may have gaps, because sequence numbers are being generated by the other user. One user can never acquire the sequence number generated by another user. After a sequence value is generated by one user, that user can continue to access that value regardless of whether the sequence is incremented by another user.

Sequence numbers are generated independently of tables, so the same sequence can be used for one or for multiple tables. It is possible that individual sequence numbers will appear to be skipped, because they were generated and used in a transaction that ultimately rolled back. Additionally, a single user may not realize that other users are drawing from the same sequence.

After a sequence is created, you can access its values in SQL statements with the `CURRVAL` pseudocolumn, which returns the current value of the sequence, or the `NEXTVAL` pseudocolumn, which increments the sequence and returns the new value.

See Also:

- Pseudocolumns for more information on using the `CURRVAL` and `NEXTVAL`
- "How to Use Sequence Values " for information on using sequences
- ALTER SEQUENCE or DROP SEQUENCE for information on modifying or dropping a sequence

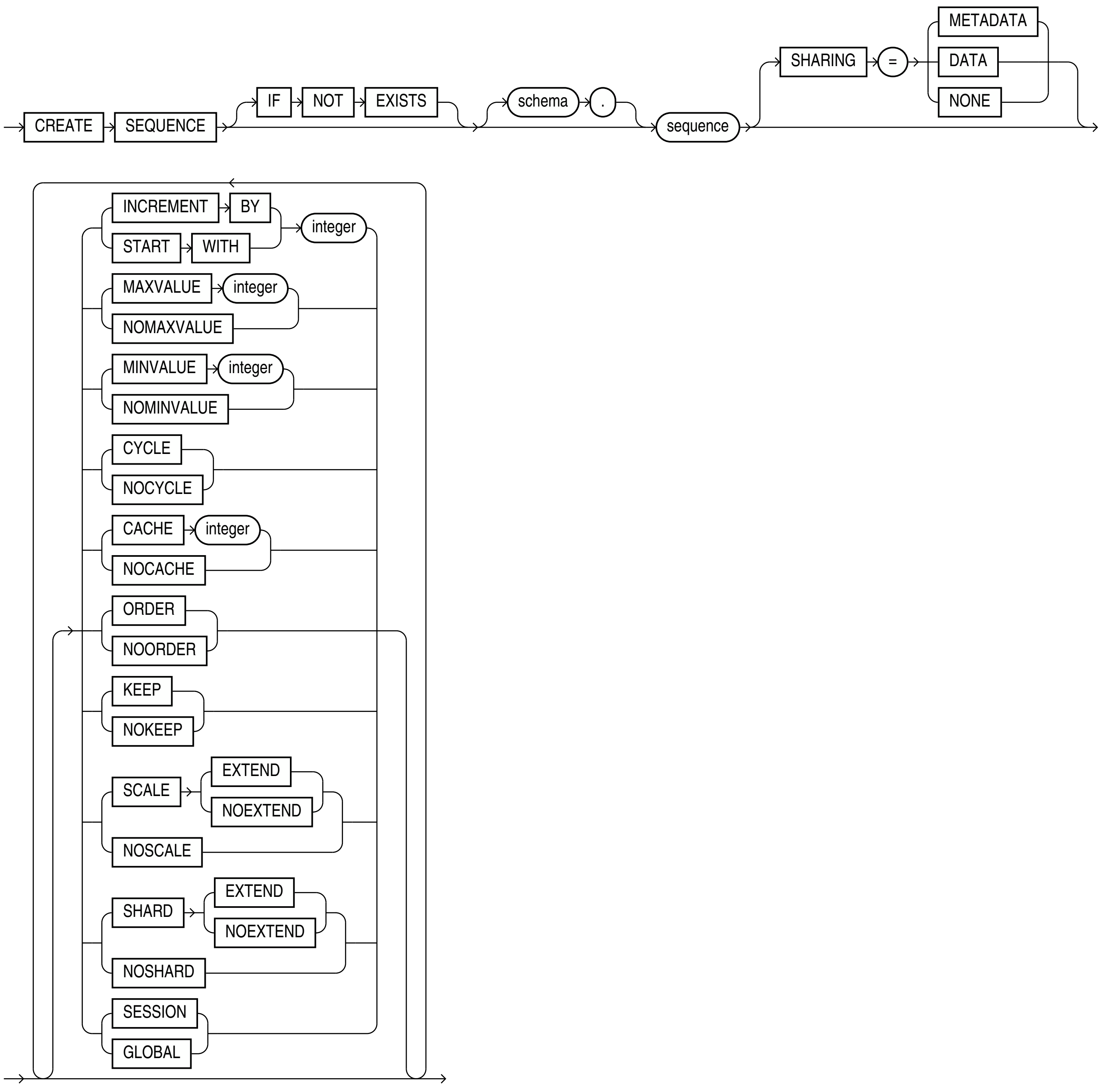
Prerequisites

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

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

Syntax

***create_sequence*::=**



Semantics

IF NOT EXISTS

Specifying `IF NOT EXISTS` has the following effects:

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

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

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

schema

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

sequence

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

If you specify none of the clauses `INCREMENT BY` through `GLOBAL`, then you create an ascending sequence that starts with 1 and increases by 1 with no upper limit. Specifying only `INCREMENT BY` -1 creates a descending sequence that starts with -1 and decreases with no lower limit.

- To create a sequence that increments without bound, for ascending sequences, omit the `MAXVALUE` parameter or specify `NOMAXVALUE`. For descending sequences, omit the `MINVALUE` parameter or specify the `NOMINVALUE`.
- To create a sequence that stops at a predefined limit, for an ascending sequence, specify a value for the `MAXVALUE` parameter. For a descending sequence, specify a value for the `MINVALUE` parameter. Also specify `NOCYCLE`. Any attempt to generate a sequence number once the sequence has reached its limit results in an error.
- To create a sequence that restarts after reaching a predefined limit, specify values for both the `MAXVALUE` and `MINVALUE` parameters. Also specify `CYCLE`.

SHARING

This clause applies only when creating a sequence in an application root. This type of sequence 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 sequence is shared, specify one of the following sharing attributes:

- `METADATA` - A metadata link shares the sequence’s metadata, but its data is unique to each container. This type of sequence is referred to as a **metadata-linked application common object**.
- `DATA` - A data link shares the sequence, and its data is the same for all containers in the application container. Its data is stored only in the application root. This type of sequence is referred to as a **data-linked application common object**.
- `NONE` - The sequence 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 sequence. If the `DEFAULT_SHARING` initialization parameter does not have a value, then the default is `METADATA`.

You cannot change the sharing attribute of a sequence 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

INCREMENT BY

Specify the interval between sequence numbers. This integer value can be any positive or negative integer, but it cannot be 0. This value can have 28 or fewer digits for an ascending sequence and 27 or fewer digits for a descending sequence. The absolute of this value must be less than the difference of `MAXVALUE` and `MINVALUE`. If this value is negative, then the sequence descends. If the value is positive, then the sequence ascends. If you omit this clause, then the interval defaults to 1.

START WITH

Specify the first sequence number to be generated. Use this clause to start an ascending sequence at a value greater than its minimum or to start a descending sequence at a value less than its maximum. For ascending sequences, the default value is the minimum value of the sequence. For descending sequences, the default value is the maximum value of the sequence. This integer value can have 28 or fewer digits for positive values and 27 or fewer digits for negative values.

Note:

This value is not necessarily the value to which an ascending or descending cycling sequence cycles after reaching its maximum or minimum value, respectively.

MAXVALUE

Specify the maximum value the sequence can generate. This integer value can have 28 or fewer digits for positive values and 27 or fewer digits for negative values. `MAXVALUE` must be equal to or greater than `START WITH` and must be greater than `MINVALUE`.

NOMAXVALUE

Specify `NOMAXVALUE` to indicate a maximum value of 10^{28}-1 for an ascending sequence or -1 for a descending sequence. This is the default.

MINVALUE

Specify the minimum value of the sequence. This integer value can have 28 or fewer digits for positive values and 27 or fewer digits for negative values. `MINVALUE` must be less than or equal to `START WITH` and must be less than `MAXVALUE`.

NOMINVALUE

Specify `NOMINVALUE` to indicate a minimum value of 1 for an ascending sequence or -(10^{27} -1) for a descending sequence. This is the default.

CYCLE

Specify `CYCLE` to indicate that the sequence continues to generate values after reaching either its maximum or minimum value. After an ascending sequence reaches its maximum value, it generates its minimum value. After a descending sequence reaches its minimum, it generates its maximum value.

NOCYCLE

Specify `NOCYCLE` to indicate that the sequence cannot generate more values after reaching its maximum or minimum value. This is the default.

CACHE

Specify how many values of the sequence the database preallocates and keeps in memory for faster access. This integer value can have 28 or fewer digits. The minimum value for this parameter is 2. For sequences that cycle, this value must be less than the number of values in the cycle. You cannot cache more values than will fit in a given cycle of sequence numbers. Therefore, the maximum value allowed for `CACHE` must be less than the value determined by the following formula:

```
CEIL ( (MAXVALUE - MINVALUE) / ABS (INCREMENT) )
```

If a system failure occurs, then all cached sequence values that have not been used in committed DML statements are lost. The potential number of lost values is equal to the value of the `CACHE` parameter.

> **Note:**
>
> Oracle recommends using the `CACHE` setting to enhance performance if you are using sequences in an Oracle Real Application Clusters environment.

NOCACHE

Specify `NOCACHE` to indicate that values of the sequence are not preallocated. If you omit both `CACHE` and `NOCACHE`, then the database caches 20 sequence numbers by default.

ORDER

Specify `ORDER` to guarantee that sequence numbers are generated in order of request. This clause is useful if you are using the sequence numbers as timestamps. Guaranteeing order is usually not important for sequences used to generate primary keys.

NOORDER

Specify `NOORDER` if you do not want to guarantee sequence numbers are generated in order of request. This is the default.

KEEP

Specify `KEEP` if you want `NEXTVAL` to retain its original value during replay for Application Continuity. This behavior will occur only if the user running the application is the owner of the schema containing the sequence. This clause is useful for providing bind variable consistency at replay after recoverable errors. Refer to *Oracle Database Development Guide* for more information on Application Continuity.

NOKEEP

Specify `NOKEEP` if you do not want `NEXTVAL` to retain its original value during replay for Application Continuity. This is the default.

> **Note:**
>
> The `KEEP` and `NOKEEP` clauses apply only to the owner of the schema containing the sequence. You can control whether `NEXTVAL` retains its original value for other users during replay for Application Continuity by granting or revoking the `KEEP SEQUENCE` object privilege on the sequence. Refer to Table 18-4 for more information on the `KEEP SEQUENCE` object privilege.

SCALE

Use `SCALE` to create a scalable sequence. A scalable sequence adds a 5 digit prefix to the sequence. The prefix is made up of a 2 digit instance offset concatenated to a 3 digit session offset as follows:

```
[(instance id % 100) ] || [session id % 1000]
```

The final sequence number is in the format `prefix || zero-padding || sequence`, where the amount of padding depends on the maximum width of the sequence.

Prior to Release 23, a scalable sequence would have a leading "1" as part of the instance offset:

```
SELECT mysequence.nextval FROM DUAL;

NEXTVAL
----------
101213001
```

In Release 23, this same scalable sequence will have the leading "1" of the instance offset removed:

```
SELECT mysequence.nextval FROM DUAL;

NEXTVAL
----------
1213001
```

> **Note:**
>
> Starting with Oracle Database Release 23 any newly created scalable sequences will have the leading "1" of the instance offset removed. Scalable sequences created prior to Release 23 will retain the leading '1' of the instance offset.

When you use `SCALE` it is highly recommended that you not use `ORDER` simultaneously on the sequence.

EXTEND

Specifying `EXTEND` with `SCALE` causes the sequence number to be left padded with zeros to its maximum length, then the prefix concatenated, so the final sequence number has 6 more digits than the `MAXVALUE` setting.

NOEXTEND

The default setting for the `SCALE` clause is `NOEXTEND`. With the `NOEXTEND` setting, the generated sequence values are at most as wide as the maximum number of digits in the sequence `MAXVALUE` setting.

`NOEXTEND` is the default setting for the `SCALE` clause. With the `NOEXTEND` setting, the generated sequence values are at most as wide as the maximum number of digits in the sequence `(maxvalue/minvalue)`. This setting is useful for integration with existing applications where sequences are used to populate fixed width columns.

NOSCALE

The default attribute for a sequence is `NOSCALE`, but you can also specify it explicitly to disable sequence scalability..

SHARD

For complete semantics on the `SHARD` clause please refer to the `SHARD` clause of the ALTER SEQUENCE statement.

SESSION

Specify `SESSION` to create a session sequence, which is a special type of sequence that is specifically designed to be used with global temporary tables that have session visibility. Unlike the existing regular sequences (referred to as "global" sequences for the sake of comparison), a session sequence returns a unique range of sequence numbers only within a session, but not across sessions. Another difference is that session sequences are not persistent. If a session goes away, so does the state of the session sequences that were accessed during the session.

Session sequences must be created by a read-write database but can be accessed on any read-write or read-only databases (either a regular database temporarily open read-only or a standby database).

The `CACHE`, `NOCACHE`, `ORDER`, or `NOORDER` clauses are ignored when specified with the `SESSION` clause.

See Also:

Oracle Data Guard Concepts and Administration for more information on session sequences

GLOBAL

Specify `GLOBAL` to create a global, or regular, sequence. This is the default.

Examples

Creating a Sequence: Example

The following statement creates the sequence `customers_seq` in the sample schema `oe`. This sequence could be used to provide customer ID numbers when rows are added to the `customers` table.

```
CREATE SEQUENCE customers_seq
 START WITH     1000
 INCREMENT BY   1
 NOCACHE
 NOCYCLE;
```

The first reference to `customers_seq.nextval` returns 1000. The second returns 1001. Each subsequent reference will return a value 1 greater than the previous reference.

CREATE SPFILE

Purpose

Use the `CREATE SPFILE` statement to create a server parameter file either from a traditional plain-text initialization parameter file or from the current system-wide settings. Server parameter files are binary files that exist only on the server and are called from client locations to start up the database.

Server parameter files let you make persistent changes to individual parameters. When you use a server parameter file, you can specify in an `ALTER SYSTEM SET` *`parameter`* statement that the new parameter value should be persistent. This means that the new value applies not only in the current instance, but also to any instances that are started up subsequently. Traditional plain-text parameter files do not let you make persistent changes to parameter values.

Server parameter files are located on the server, so they allow for automatic database tuning by Oracle Database and for backup by Recovery Manager (RMAN).

To use a server parameter file when starting up the database, you must create it using the `CREATE SPFILE` statement.

All instances in an Oracle Real Application Clusters environment must use the same server parameter file. However, when otherwise permitted, individual instances can have different settings of the same parameter within this one file. Instance-specific parameter definitions are specified as *`SID.parameter = value`*, where *`SID`* is the instance identifier.

The method of starting up the database with a server parameter file depends on whether you create a default or nondefault server parameter file. Refer to "Creating a Server Parameter File: Examples" for examples of how to use server parameter files.

Note on Creating Server Parameter Files in a CDB

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

- If the current container is the root, then the values that you set for initialization parameters in the root are used as default values for all other containers.
- If the current container is a PDB, then the database stores the PDB's initialization parameter values internally, rather than in a file. Therefore, you cannot specify an *`spfile_name`*. The values that you set for initialization parameters in the PDB are persistent and override any values set for those parameters in the root.

When PDB is in `MOUNT` state, any query on `V$parameter` (show parameter) shows the values from `ROOT` parameter table.

When PDB is in `OPEN` state, any query on `V$parameter` (show parameter) shows the values from PDB parameter table.

You can subsequently use the `ALTER SYSTEM` statement to modify initialization parameter values for the root or a PDB.

See Also:

- CREATE PFILE for information on creating a regular text parameter file from a binary server parameter file
- *Oracle Database Administrator's Guide* for information on traditional plain-text initialization parameter files and server parameter files
- *Oracle Real Application Clusters Administration and Deployment Guide* for information on using server parameter files in an Oracle Real Application Clusters environment

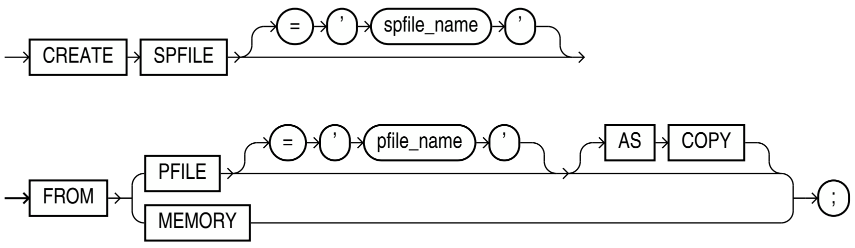
Prerequisites

You must have the `SYSBACKUP`, `SYSDBA`, `SYSDG`, or `SYSOPER` system privilege to execute this statement. You can execute this statement before or after instance startup. However, if you have already started an instance using *`spfile_name`*, you cannot specify the same *`spfile_name`* in this statement.

To create a server parameter file in a CDB, the current container must be the root and you must have the commonly granted `SYSBACKUP`, `SYSDBA`, `SYSDG`, or `SYSOPER` system privilege.

Syntax

***create_spfile*::=**



Semantics

spfile_name

This clause lets you specify a name for the server parameter file you are creating.

If you specify *`spfile_name`*, then Oracle Database creates a nondefault server parameter file.

- For *`spfile_name`*, you can specify a traditional filename, a file in an Oracle ACFS (Oracle Advanced Cluster File System) file system, or an Oracle Storage Management (Oracle ASM) filename.

- If you specify a traditional filename or a file in an Oracle ACFS file system, then *spfile_name* can include a path prefix. If you do not specify such a path prefix, then the database adds the path prefix for the default storage location, which is platform dependent.
- If you specify the Oracle ASM filename syntax, then the database creates the spfile in an Oracle ASM disk group.
- When using a nondefault server parameter file, you must specify the server parameter filename in the `STARTUP` command when you start up the database. The exception to this rule is as follows:
  - If the database is defined as a resource in Oracle Clusterware, the instance from which the command is issued is running, and you specify the *spfile_name*, specify the `FROM PFILE` clause, and omit the `AS COPY` clause, then this statement automatically updates the SPFILE in the database resource. In this case, you can start up the database without referring to the server parameter file by name. If the instance from which the command is issued is *not* running, then the SPFILE in the database resource must be updated manually using `srvctl modify database -d` *dbname* `-spfile` *spfile_path*.

If you omit *spfile_name*, then Oracle Database uses the platform-specific default server parameter filename. If such a file already exists on the server, then this statement overwrites it. When using a default server parameter file, you can start up the database without referring to the file by name.

Restriction on *spfile_name*

You cannot specify *spfile_name* when creating a server parameter file while connected to a PDB.

See Also:

- "Creating a Server Parameter File: Examples" for information on starting up the database with default and nondefault server parameter files
- *file_specification* for the syntax of traditional and Oracle ASM filenames and ALTER DISKGROUP for information on modifying the characteristics of an Oracle ASM file
- The appropriate operating-system-specific documentation for default parameter file names

pfile_name

Specify the traditional plain-text initialization parameter file from which you want to create a server parameter file. The traditional parameter file must reside on the server.

- If you specify *pfile_name* and the traditional parameter file does not reside in the default directory for parameter files on your operating system, then you must specify the full path.
- If you do not specify *pfile_name*, then Oracle Database looks in the default directory for parameter files on your operating system for the default parameter filename and uses that file. If that file does not exist in the expected directory, then the database returns an error.

> **Note:**
>
> In an Oracle Real Application Clusters environment, you must first combine all instance parameter files into one file before specifying that filename in this statement to create a server parameter file. For information on accomplishing this step, see *Oracle Real Application Clusters Administration and Deployment Guide*.

AS COPY

This clause applies only if the database is defined as a resource in Oracle Clusterware. By default, if you specify both the `spfile_name` and the `FROM PFILE` clause, then the `CREATE SPFILE` statement automatically updates the SPFILE in the database resource. You can specify `AS COPY` to prevent the database from updating the SPFILE in the database resource.

MEMORY

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

Examples

Creating a Server Parameter File: Examples

The following example creates a default server parameter file from a traditional plain-text parameter file named `t_init1.ora`:

```
CREATE SPFILE
   FROM PFILE = '$ORACLE_HOME/work/t_init1.ora';
```

> **Note:**
>
> Typically you will need to specify the full path and filename for parameter files on your operating system.

When you create a default server parameter file, you subsequently start up the database using that server parameter file by using the SQL*Plus command `STARTUP` without the `PFILE` parameter, as follows:

```
STARTUP
```

The following example creates a nondefault server parameter file `s_params.ora` from a traditional plain-text parameter file named `t_init1.ora`:

```
CREATE SPFILE = 's_params.ora'
   FROM PFILE = '$ORACLE_HOME/work/t_init1.ora';
```

When you create a nondefault server parameter file, you subsequently start up the database by first creating a traditional parameter file containing the following single line:

```
spfile = 's_params.ora'
```

The name of this parameter file must comply with the naming conventions of your operating system. You then use the single-line parameter file in the `STARTUP` command. The following example shows how to start up the database, assuming that the single-line parameter file is named `new_param.ora`:

```
STARTUP PFILE=new_param.ora
```

CREATE SYNONYM

Purpose

Use the `CREATE SYNONYM` statement to create a **synonym**, which is an alternative name for a table, view, sequence, operator, procedure, stored function, package, materialized view, Java class schema object, user-defined object type, or another synonym. A synonym places a dependency on its target object and becomes invalid if the target object is changed or dropped.

Synonyms provide both data independence and location transparency. Synonyms permit applications to function without modification regardless of which user owns the table or view and regardless of which database holds the table or view. However, synonyms are not a substitute for privileges on database objects. Appropriate privileges must be granted to a user before the user can use the synonym.

You can refer to synonyms in the following DML statements: `SELECT`, `INSERT`, `UPDATE`, `DELETE`, `FLASHBACK TABLE`, `EXPLAIN PLAN`, `LOCK TABLE`, `MERGE`, and `CALL` .

You can refer to synonyms in the following DDL statements: `AUDIT`, `NOAUDIT`, `GRANT`, `REVOKE`, and `COMMENT`.

See Also:

Oracle Database Concepts for general information on synonyms

Prerequisites

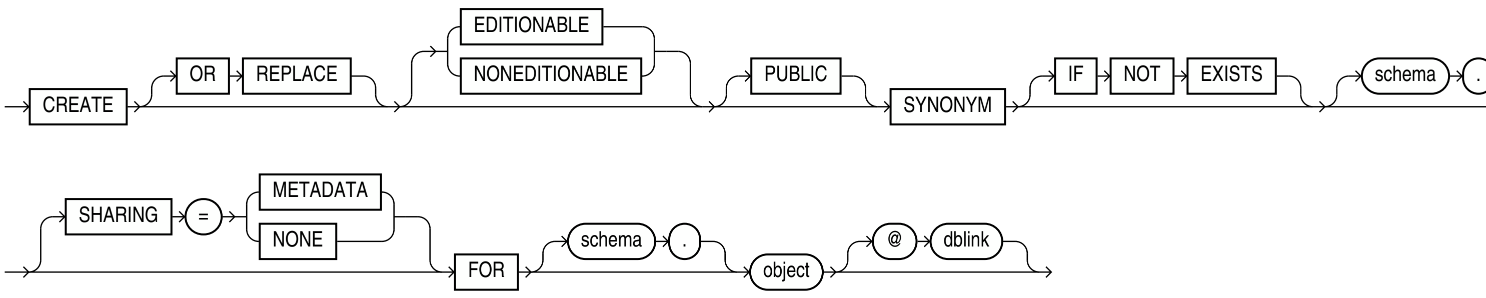
To create a private synonym in your own schema, you must have the `CREATE SYNONYM` system privilege.

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

To create a `PUBLIC` synonym, you must have the `CREATE PUBLIC SYNONYM` system privilege.

Syntax

***create_synonym*::=**



Semantics

OR REPLACE

Specify `OR REPLACE` to re-create the synonym if it already exists. Use this clause to change the definition of an existing synonym without first dropping it.

Restriction on Replacing a Synonym

You cannot use the `OR REPLACE` clause for a type synonym that has any dependent tables or dependent valid user-defined object types.

IF NOT EXISTS

Specifying `IF NOT EXISTS` has the following effects:

- If the synonym does not exist, a new synonym is created at the end of the statement.
- If the synonym exists, this is the synonym 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 synonym is an editioned or noneditioned object if editioning is enabled for the schema object type `SYNONYM` in *`schema`*. For private synonyms, the default is `EDITIONABLE`. For public synonyms, the default is `NONEDITIONABLE`. For information about editioned and noneditioned objects, see *Oracle Database Development Guide*.

PUBLIC

Specify `PUBLIC` to create a public synonym. Public synonyms are accessible to all users. However each user must have appropriate privileges on the underlying object in order to use the synonym.

When resolving references to an object, Oracle Database uses a public synonym only if the object is not prefaced by a schema and is not followed by a database link.

If you omit this clause, then the synonym is private. A private synonym name must be unique in its schema. A private synonym is accessible to users other than the owner only if those users have appropriate privileges on the underlying database object and specify the schema along with the synonym name.

Notes on Public Synonyms

The following notes apply to public synonyms:

- If you create a public synonym and it subsequently has dependent tables or dependent valid user-defined object types, then you cannot create another database object of the same name as the synonym in the same schema as the dependent objects.
- Take care not to create a public synonym with the same name as an existing schema. If you do so, then all PL/SQL units that use that name will be invalidated.

schema

Specify the schema to contain the synonym. If you omit `schema`, then Oracle Database creates the synonym in your own schema. You cannot specify a schema for the synonym if you have specified `PUBLIC`.

synonym

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

Note:

The maximum length of a synonym name is subject to the following rules:

- If the `COMPATIBLE` initialization parameter is set to a value of `12.2` or higher, then the maximum length of a synonym name is 128 bytes. The database will allow you to create and drop synonyms of length 129 to 4000 bytes. However, unless these longer synonym names represent a Java name they will not work in any other SQL command.
- If the `COMPATIBLE` initialization parameter is set to a value lower than `12.2`, then the maximum length of a synonym name is 30 bytes. The database will allow you to create and drop synonyms of length 31 to 128 bytes. However, unless these longer synonym names represent a Java name they will not work in any other SQL command.

The longer synonym names are transformed into obscure shorter strings for storage in the data dictionary.

See Also:

"CREATE SYNONYM: Examples" and "Oracle Database Resolution of Synonyms: Example"

SHARING

This clause applies only when creating a synonym in an application root. This type of synonym 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 synonym is shared, specify one of the following sharing attributes:

- `METADATA` - A metadata link shares the synonym’s metadata, but its data is unique to each container. This type of synonym is referred to as a **metadata-linked application common object**.
- `NONE` - The synonym 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 synonym. If the `DEFAULT_SHARING` initialization parameter does not have a value, then the default is `METADATA`.

You cannot change the sharing attribute of a synonym 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

FOR Clause

Specify the object for which the synonym is created. The schema object for which you are creating the synonym can be of the following types:

- Table or object table
- View or object view
- Sequence
- Stored procedure, function, or package
- Materialized view
- Java class schema object
- User-defined object type
- Synonym

The schema object need not currently exist and you need not have privileges to access the object.

Restriction on the FOR Clause

The schema object cannot be contained in a package.

schema

Specify the schema in which the object resides. If you do not qualify object with *`schema`*, then the database assumes that the schema object is in your own schema.

If you are creating a synonym for a procedure or function on a remote database, then you must specify *`schema`* in this `CREATE` statement. Alternatively, you can create a local public synonym on the database where the object resides. However, the database link must then be included in all subsequent calls to the procedure or function.

dblink

You can specify a complete or partial database link to create a synonym for a schema object on a remote database where the object is located. If you specify *`dblink`* and omit *`schema`*, then the synonym refers to an object in the schema specified by the database link. Oracle recommends that you specify the schema containing the object in the remote database.

If you omit *`dblink`*, then Oracle Database assumes the object is located on the local database.

Restriction on Database Links

You cannot specify *`dblink`* for a Java class synonym.

See Also:

- "References to Objects in Remote Databases " for more information on referring to database links
- CREATE DATABASE LINK for more information on creating database links

Examples

CREATE SYNONYM: Examples

To define the synonym `offices` for the table `locations` in the schema `hr`, issue the following statement:

```
CREATE SYNONYM offices
   FOR hr.locations;
```

To create a `PUBLIC` synonym for the `employees` table in the schema `hr` on the `remote` database, you could issue the following statement:

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

A synonym may have the same name as the underlying object, provided the underlying object is contained in another schema.

Oracle Database Resolution of Synonyms: Example

Oracle Database attempts to resolve references to objects at the schema level before resolving them at the `PUBLIC` synonym level. For example, the schemas `oe` and `sh` both contain tables named `customers`. In the next example, user `SYSTEM` creates a `PUBLIC` synonym named `customers` for `oe.customers`:

```
CREATE PUBLIC SYNONYM customers FOR oe.customers;
```

If the user `sh` then issues the following statement, then the database returns the count of rows from `sh.customers`:

```
SELECT COUNT(*) FROM customers;
```

To retrieve the count of rows from `oe.customers`, the user `sh` must preface `customers` with the schema name. (The user `sh` must have select permission on `oe.customers` as well.)

```
SELECT COUNT(*) FROM oe.customers;
```

If the user `hr`'s schema does not contain an object named `customers`, and if `hr` has select permission on `oe.customers`, then `hr` can access the `customers` table in `oe`'s schema by using the public synonym `customers`:

```
SELECT COUNT(*) FROM customers;
```

CREATE TABLE

Purpose

Use the `CREATE TABLE` statement to create one of the following types of tables:

- A **relational table** is the basic structure to hold user data.
- An **object table** that uses an object type for a column definition. An object table is explicitly defined to hold object instances of a particular type.
- A **JSON collection table** is a table that stores a collection of JSON documents (objects) in a JSON-type column while also guaranteeing a unique key per document.

You can also create an object type and then use it in a column when creating a relational table.

Tables are created with no data unless a subquery is specified. You can add rows to a table with the `INSERT` statement. After creating a table, you can define additional columns, partitions, and integrity constraints with the `ADD` clause of the `ALTER TABLE` statement. You can change the definition of an existing column or partition with the `MODIFY` clause of the `ALTER TABLE` statement.

See Also:

- *Oracle Database Administrator's Guide* and CREATE TYPE for more information about creating objects
- ALTER TABLE and DROP TABLE for information on modifying and dropping tables

Prerequisites

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

In addition to these table privileges, to create an object table or a relational table with an object type column, the owner of the table must have the `EXECUTE` object privilege in order to access all types referenced by the table, or you must have the `EXECUTE ANY TYPE` system privilege. These privileges must be granted explicitly and not acquired through a role.

Additionally, if the table owner intends to grant access to the table to other users, then the owner must have been granted the `EXECUTE` object privilege on the referenced types `WITH GRANT OPTION`, or have the `EXECUTE ANY TYPE` system privilege `WITH ADMIN OPTION`. Without these privileges, the table owner has insufficient privileges to grant access to the table to other users.

To enable a unique or primary key constraint, you must have the privileges necessary to create an index on the table. You need these privileges because Oracle Database creates an index on the columns of the unique or primary key in the schema containing the table.

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

To specify the *`zonemap_clause`*, you must have the permissions necessary to create a zone map. Refer to the "Prerequisites" section in the documentation on `CREATE MATERIALIZED ZONEMAP`.

To create an **external table**, you must have the required read and write operating system privileges on the appropriate operating system directories. You must have the `READ` object privilege on the database directory object corresponding to the operating system directory in which the external data resides. You must also have the `WRITE` object privilege on the database directory in which the files will reside if you specify a log file or bad file in the `opaque_format_spec` or if you unload data into an external table from a database table by specifying the `AS subquery` clause.

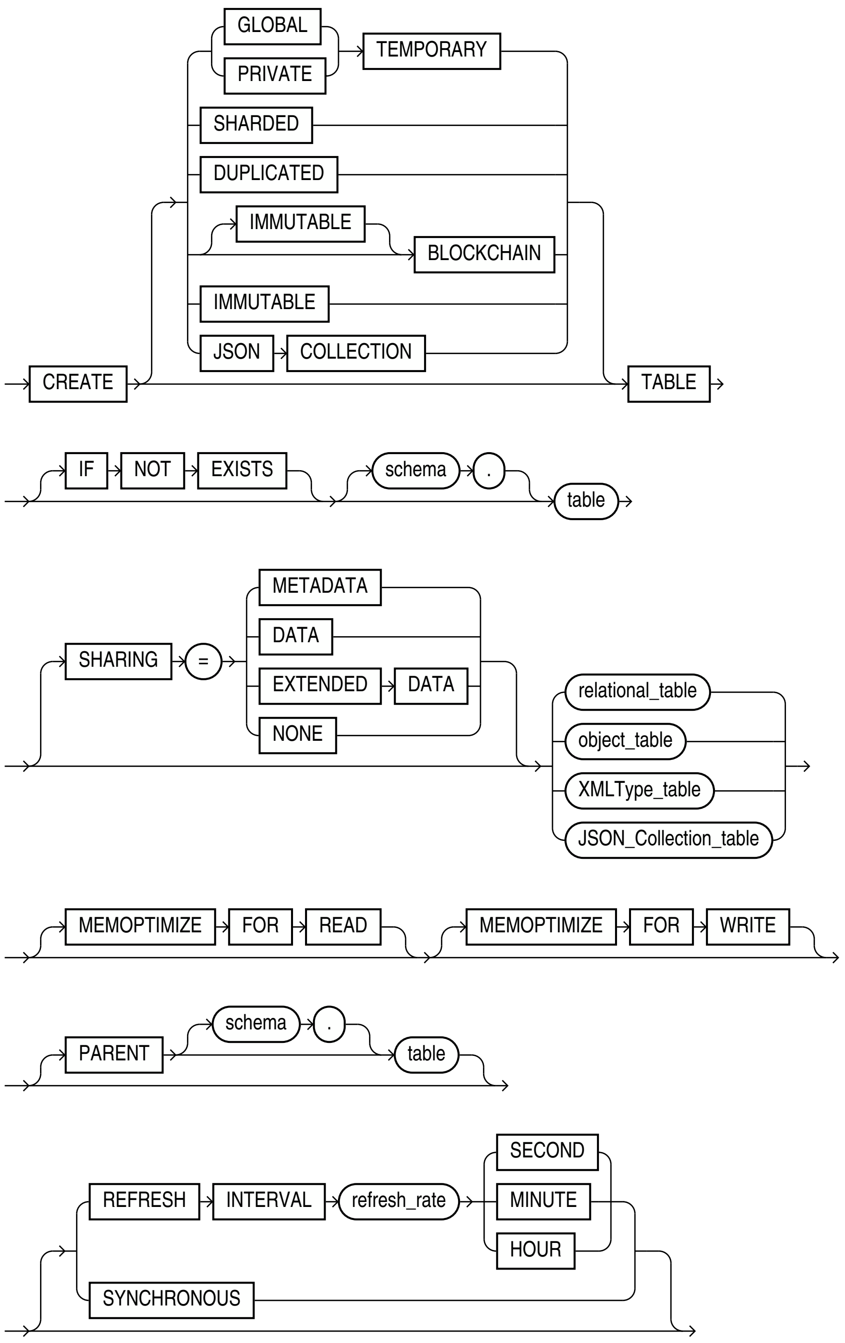
To create an **XMLType table** in a different database schema from your own, you must have not only privilege `CREATE ANY TABLE` but also privilege `CREATE ANY INDEX`. This is because a unique index is created on column `OBJECT_ID` when you create the table. Column `OBJECT_ID` stores a system-generated object identifier.

See Also:

- CREATE INDEX
- *Oracle Database Administrator's Guide* for more information about the privileges required to create tables using types

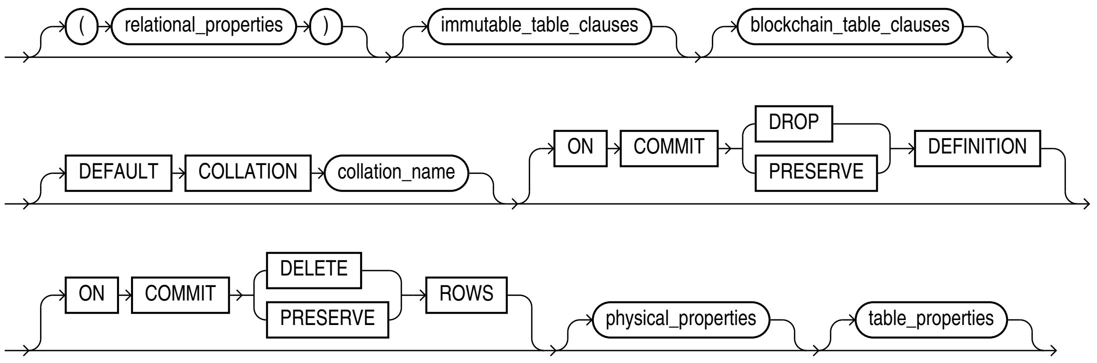
Syntax

***create_table*::=**



(*relational_table*::=, *object_table*::=, *XMLType_table*::=, *JSON_Collection_table*::=)

***relational_table*::=**

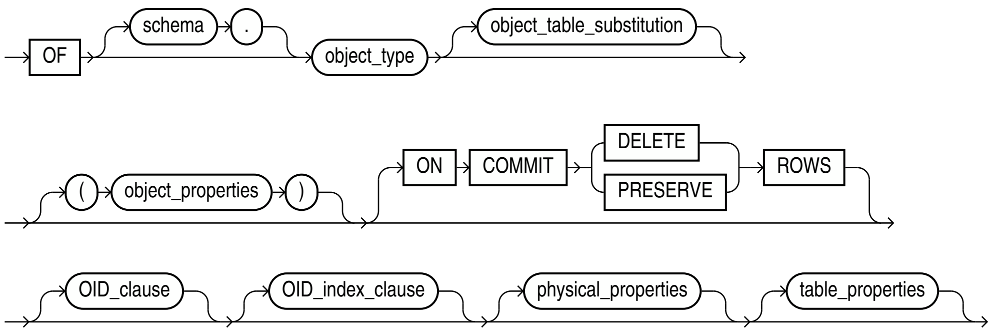


Note:

Each of the clauses following the table name is optional for any given relational table. However, for every table you must at least specify either column names and data types using the `relational_properties` clause or an `AS subquery` clause using the `table_properties` clause.

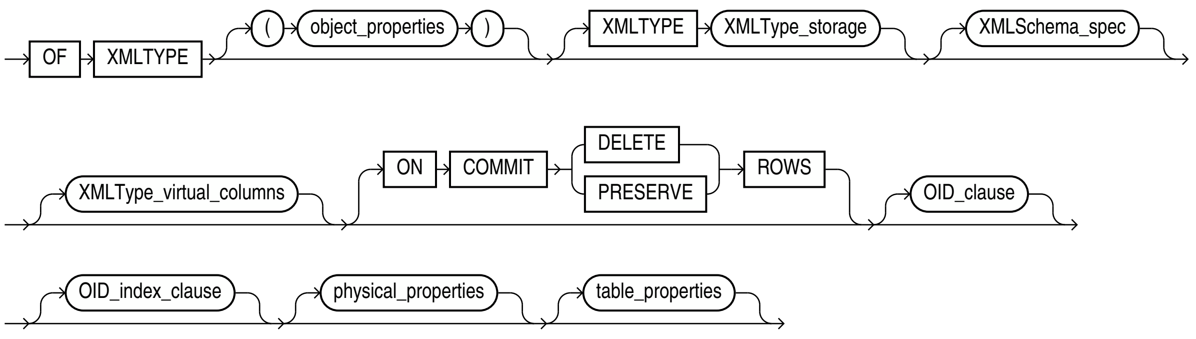
(*relational_properties*::=, immutable_table_clauses ,blockchain_table_clauses::= ,*physical_properties*::=, *table_properties*::=)

object_table::=



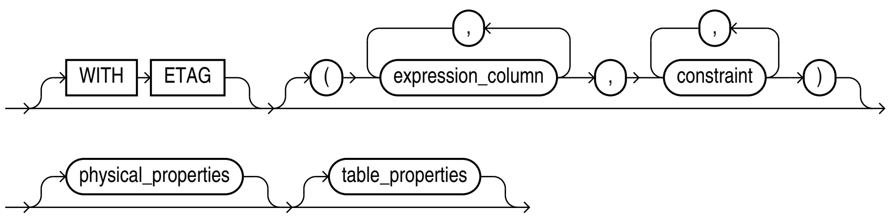
(*object_table_substitution*::=, *object_properties*::=, *oid_clause*::=, *oid_index_clause*::=, *physical_properties*::=, *table_properties*::=)

***XMLType_table*::=**

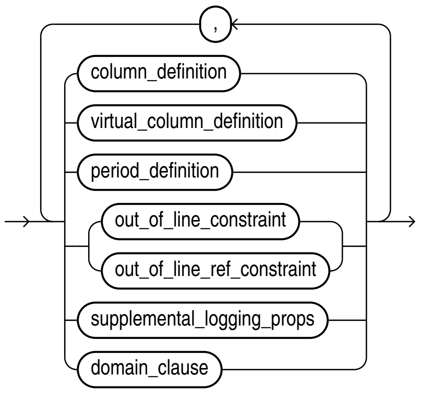


(*XMLType_storage*::=, *XMLSchema_spec*::=, *XMLType_virtual_columns*::=, *oid_clause*::=, *oid_index_clause*::=, *physical_properties*::=, *table_properties*::=)

***JSON_Collection_table*::=**



***relational_properties*::=**

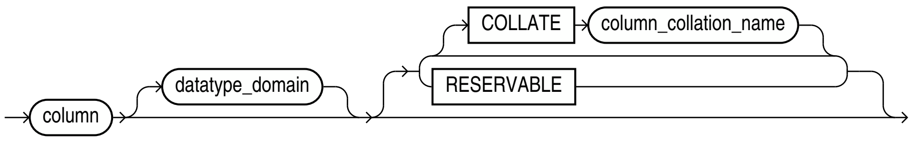

,
column_definition
virtual_column_definition
period_definition
out_of_line_constraint
out_of_line_ref_constraint
supplemental_logging_props
domain_clause

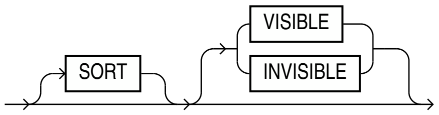
Note:

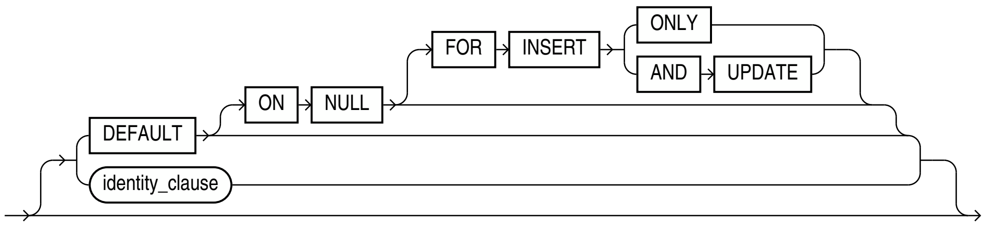
You can specify these clauses in any order with the following exception: You must specify at least one `column_definition` or `virtual_column_definition` before you specify `period_definition`. You can specify `period_definition` only once.

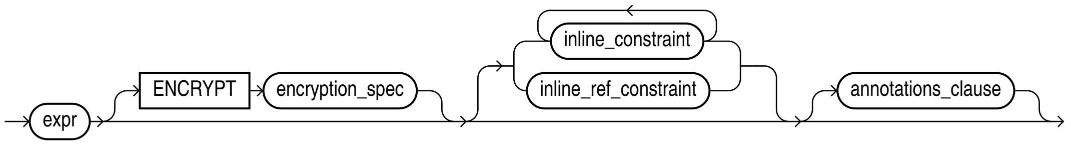
(*column_definition*::=, *virtual_column_definition*::=, *period_definition*::=, *out_of_line_constraint*::=, *out_of_line_ref_constraint*::=, *supplemental_logging_props*::=, domain_clause::=)

***column_definition*::=**



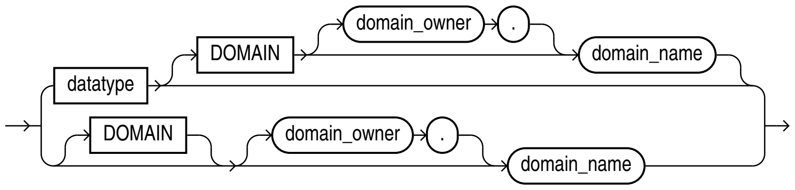






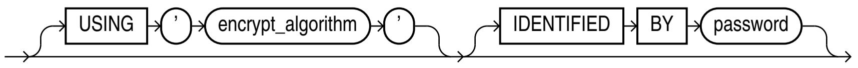
(*datatype_domain*::=,*identity_clause*::=, *inline_constraint*::=, *inline_ref_constraint*::=, *annotations_clause*::=)

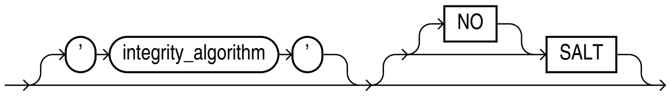
***datatype_domain*::=**



(*datatype*::=)

***encryption_spec*::=**

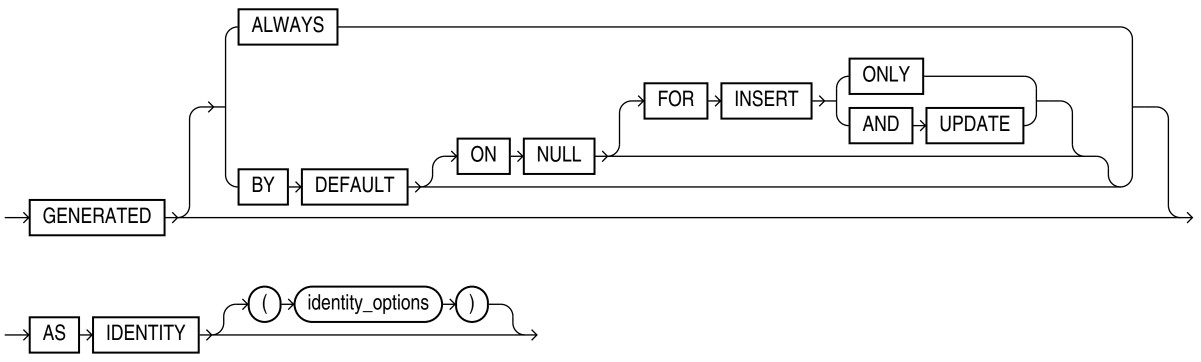




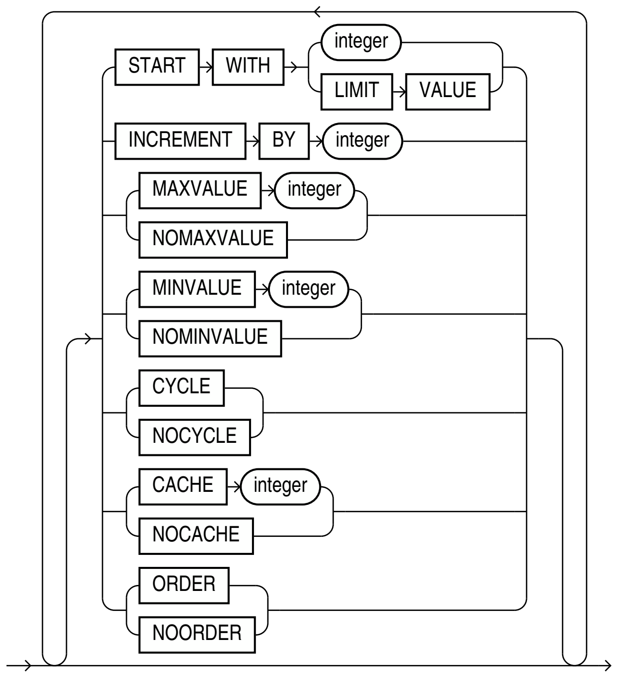
***annotations_clause*::=**

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

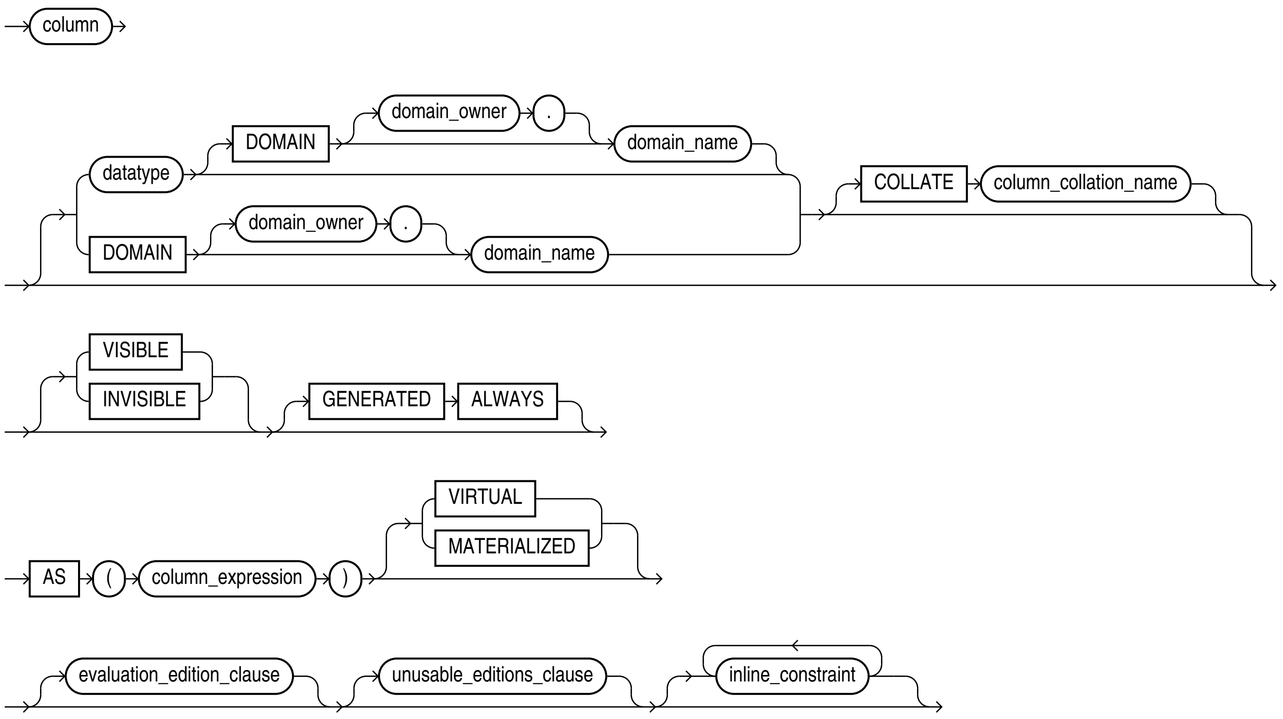
***identity_clause*::=**



***identity_options*::=**

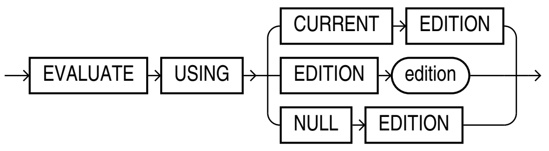


***virtual_column_definition*::=**

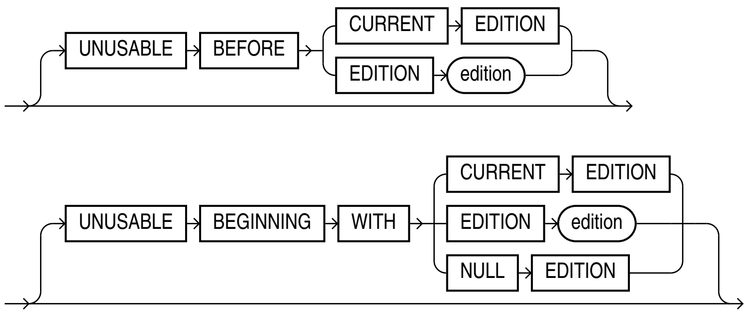


(*datatype*::=,*evaluation_edition_clause*::=, *unusable_editions_clause*::=, *constraint*::=)

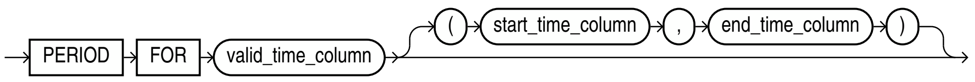
***evaluation_edition_clause*::=**



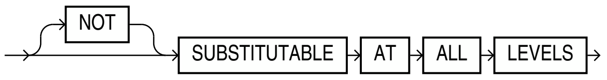
***unusable_editions_clause*::=**



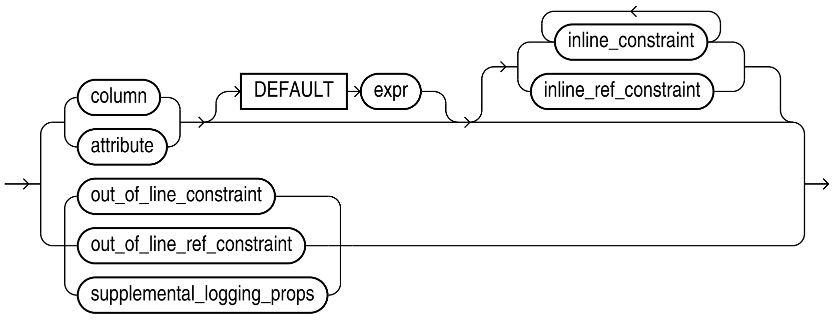
***period_definition*::=**



***object_table_substitution*::=**

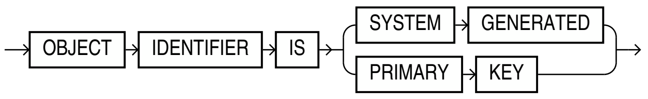


***object_properties*::=**

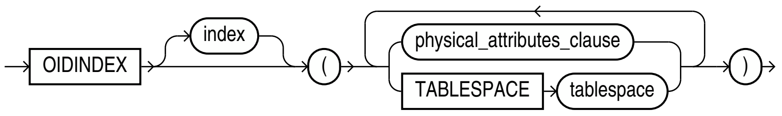


(*constraint*::=, #unique_98/unique_98_Connect_42_I2126822)

***oid_clause*::=**

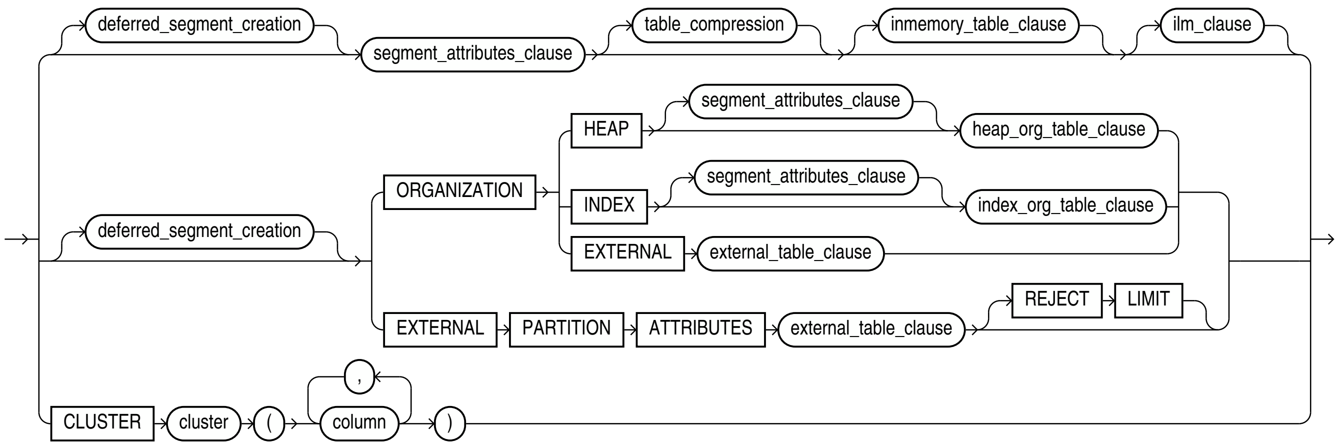


***oid_index_clause*::=**



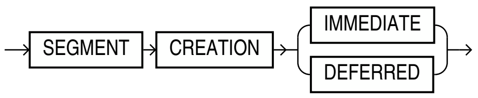
(*physical_attributes_clause*::=)

***physical_properties*::=**

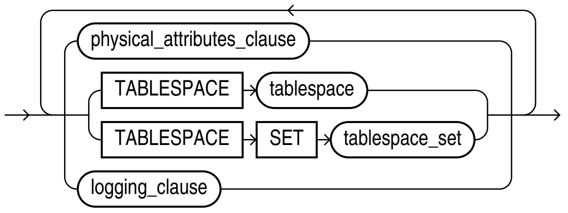


(*deferred_segment_creation*::=, *segment_attributes_clause*::=, *table_compression*::=, *inmemory_table_clause*::=, *ilm_clause*::=, *heap_org_table_clause*::=, *index_org_table_clause*::=, *external_table_clause*::=)

***deferred_segment_creation*::=**

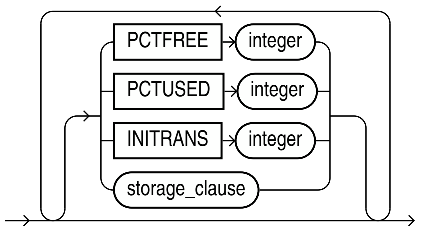


***segment_attributes_clause*::=**



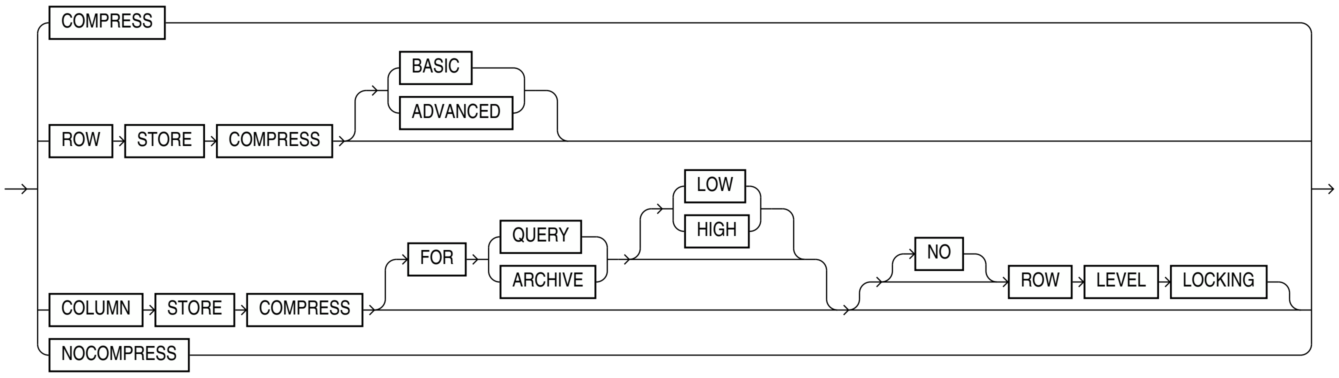
(*physical_attributes_clause*::=, *logging_clause*::=)

***physical_attributes_clause*::=**

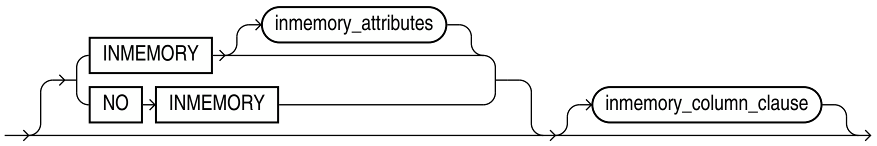


(*storage_clause::=*)

table_compression::=

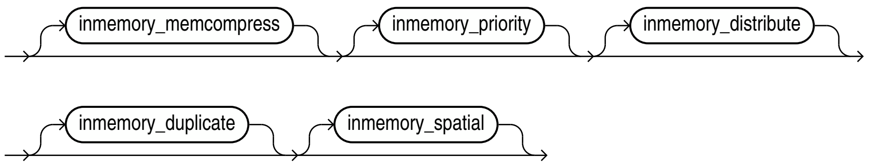


inmemory_table_clause::=



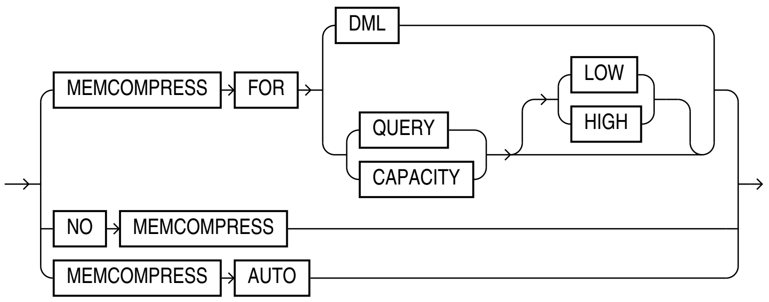
(*inmemory_attributes::=*, *inmemory_column_clause::=*)

inmemory_attributes::=

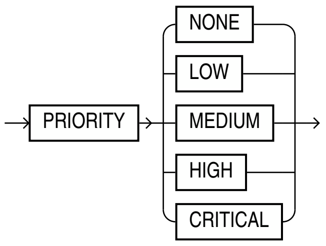


(*inmemory_memcompress::=*, *inmemory_priority::=*, *inmemory_distribute::=*, *inmemory_duplicate::=*)

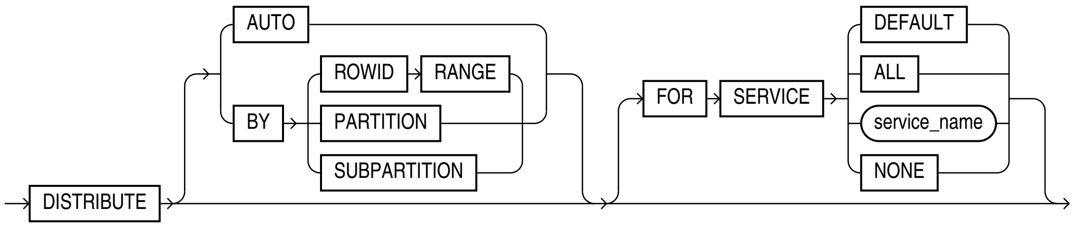
inmemory_memcompress::=



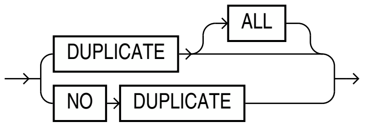
***inmemory_priority*::=**



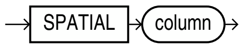
***inmemory_distribute*::=**



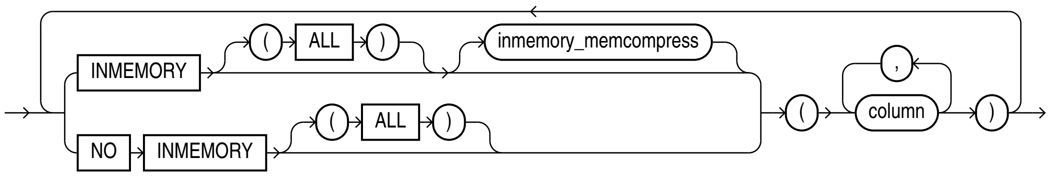
***inmemory_duplicate*::=**



***inmemory_spatial*::=**

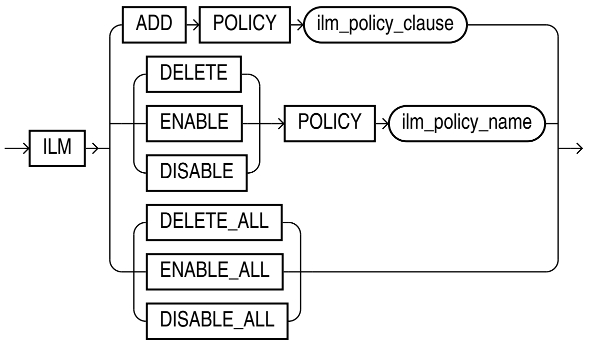


***inmemory_column_clause*::=**

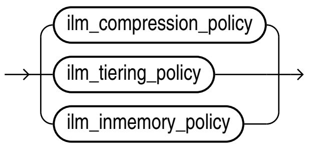


(*inmemory_memcompress*::=)

***ilm_clause*::=**

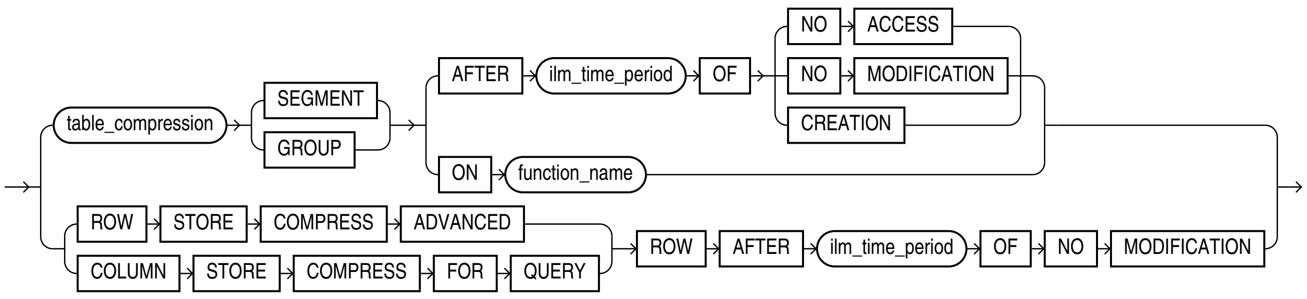


***ilm_policy_clause*::=**



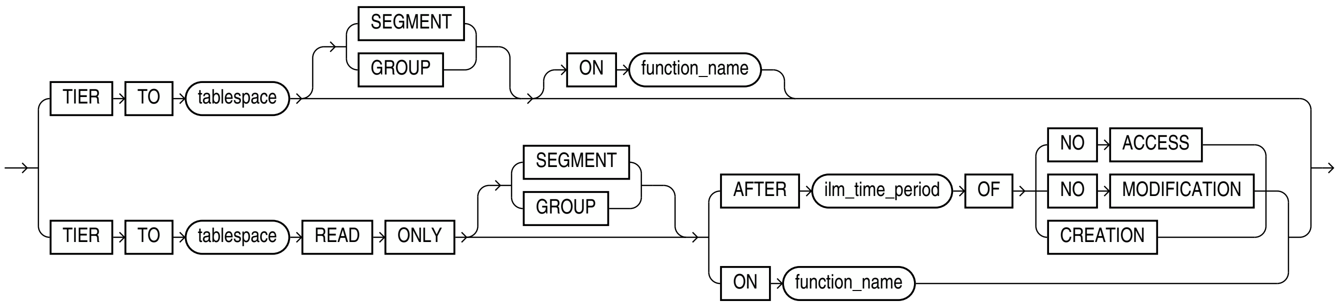
(*ilm_compression_policy::=*, *ilm_tiering_policy::=*, *ilm_inmemory_policy::=*)

***ilm_compression_policy*::=**



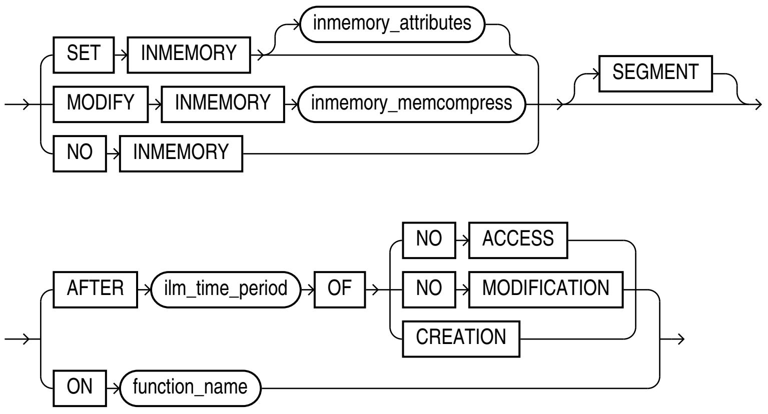
(*table_compression::=*, *ilm_time_period::=*)

***ilm_tiering_policy*::=**

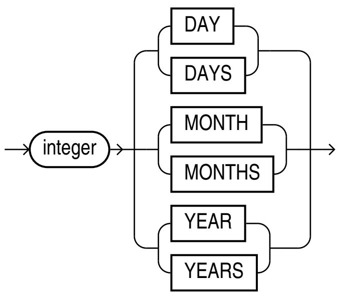


(*ilm_time_period*::=)

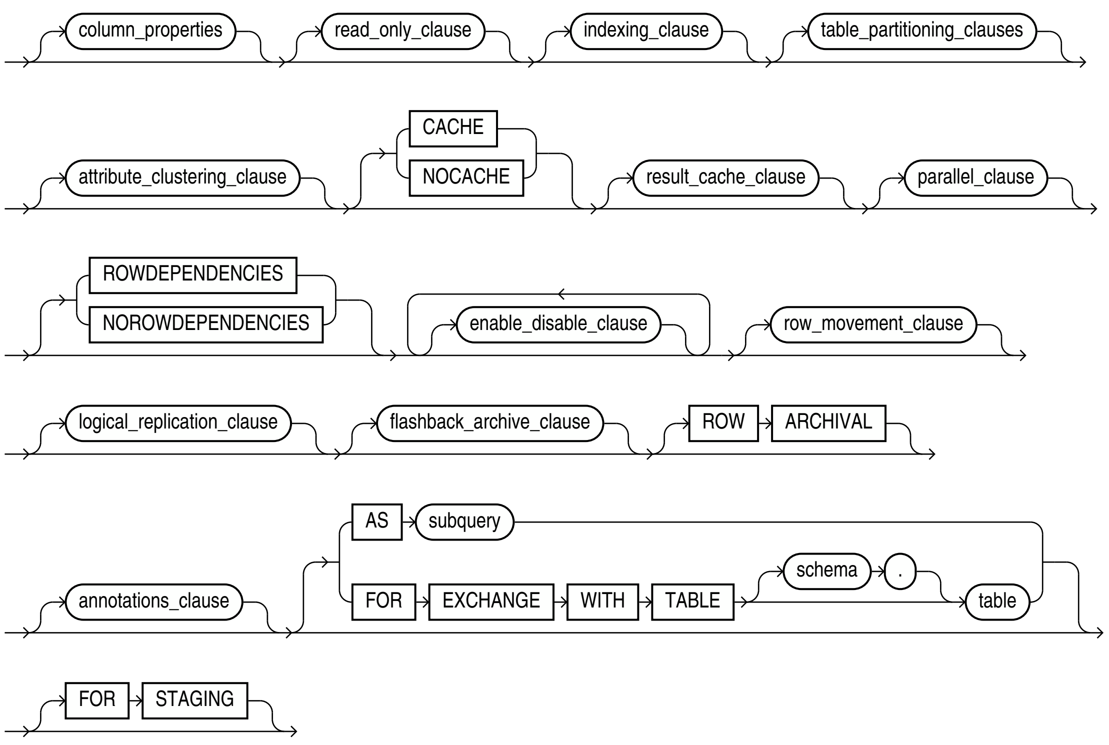
***ilm_inmemory_policy*::=**



***ilm_time_period*::=**

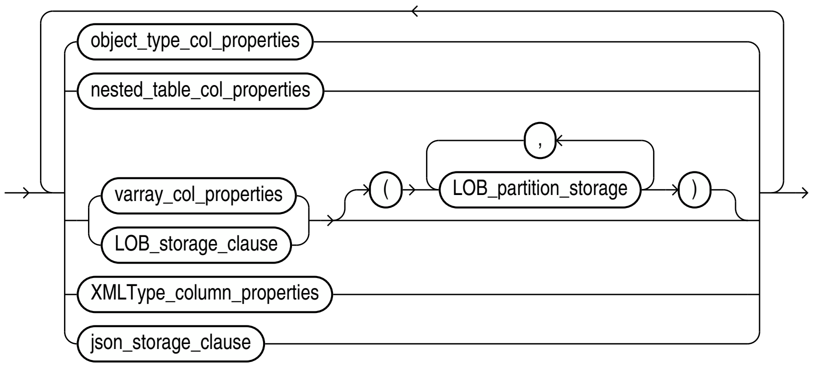


***table_properties*::=**



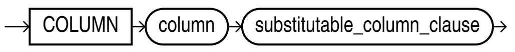
(*column_properties*::=, *read_only_clause*::=, *indexing_clause*::=, *table_partitioning_clauses*::=, *attribute_clustering_clause*::=, *parallel_clause*::=, *enable_disable_clause*::=, *row_movement_clause*::=, *logical_replication_clause*::=, *flashback_archive_clause*::= , *subquery*::=)

***column_properties*::=**

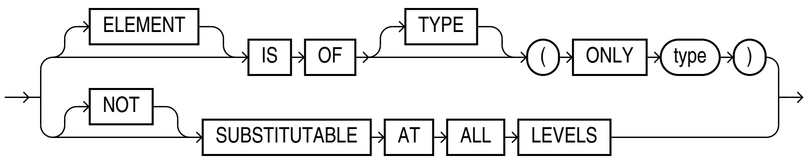


(*object_type_col_properties*::=, *nested_table_col_properties*::=, *varray_col_properties*::=, *LOB_storage_clause*::=, *LOB_partition_storage*::=, *XMLType_column_properties*::=,json_storage_clause::=)

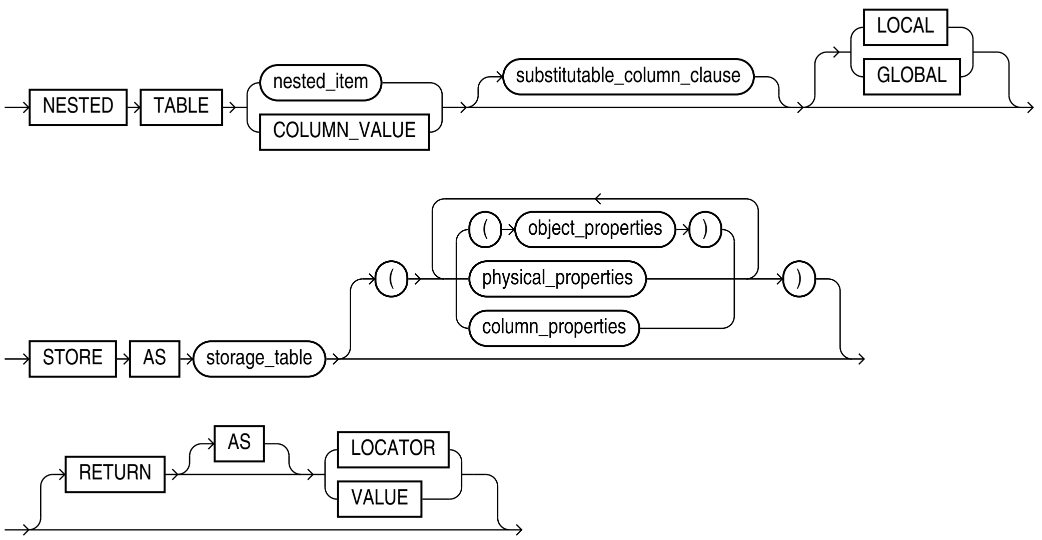
***object_type_col_properties*::=**



***substitutable_column_clause*::=**

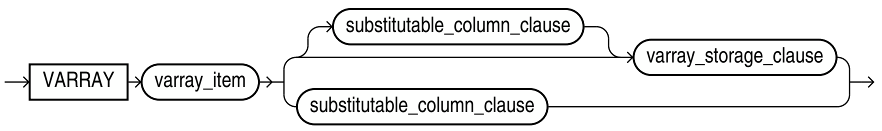


***nested_table_col_properties*::=**



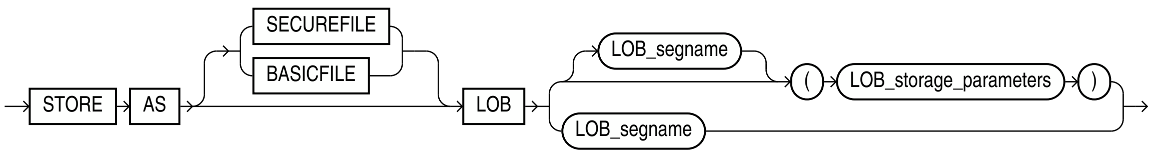
(*substitutable_column_clause*::=, *object_properties*::=, *physical_properties*::=, *column_properties*::=)

***varray_col_properties*::=**



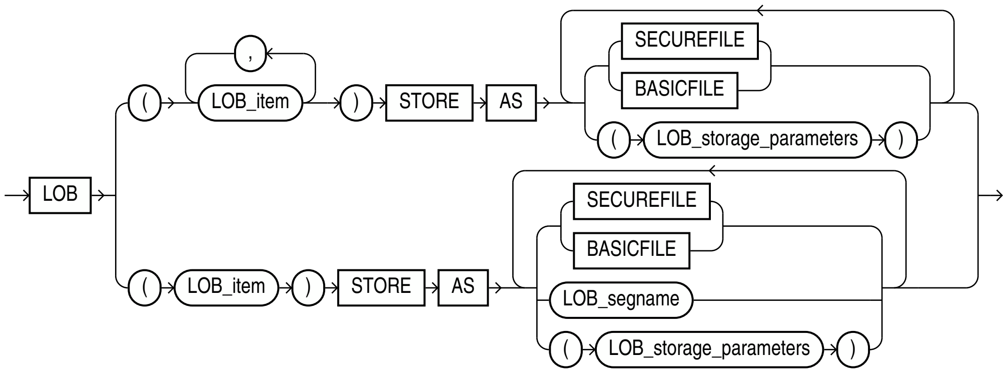
(*substitutable_column_clause*::=, *varray_storage_clause*::=)

***varray_storage_clause*::=**



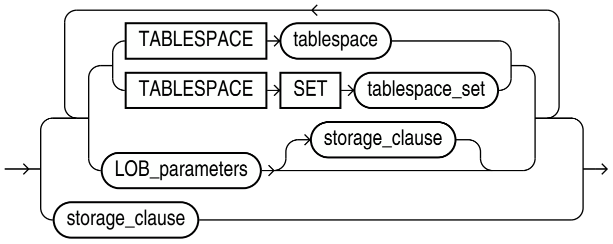
(*LOB_parameters*::=)

***LOB_storage_clause*::=**



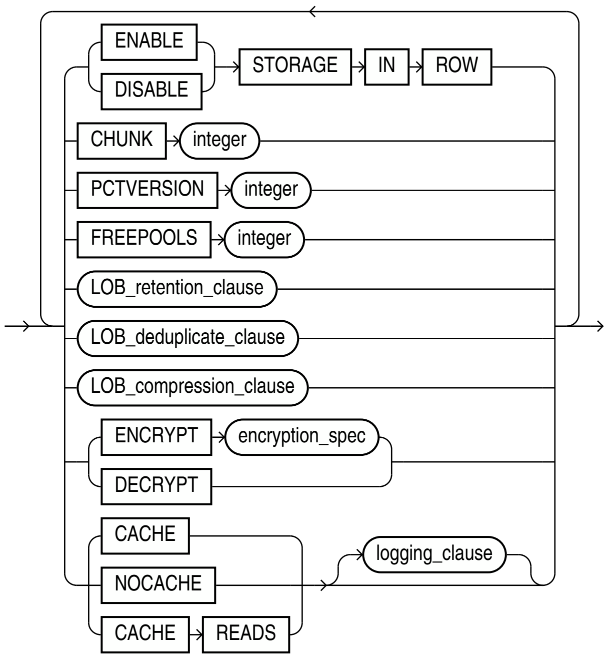
(*LOB_storage_parameters*::=)

***LOB_storage_parameters*::=**



(*LOB_parameters*::=, *storage_clause*::=)

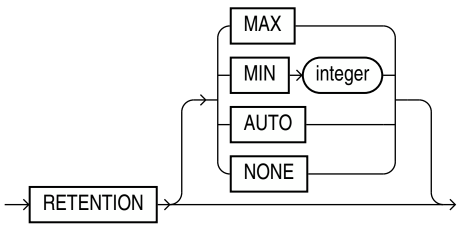
***LOB_parameters*::=**



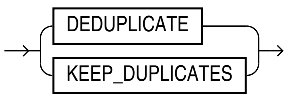
(*LOB_deduplicate_clause*::=, *LOB_compression_clause*::=, *logging_clause*::=)

> **Note:**
>
> Several of the LOB parameters are no longer needed if you are using SecureFiles for LOB storage. Refer to *LOB_storage_parameters* for more information.

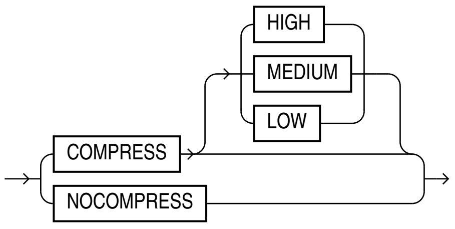
***LOB_retention_clause*::=**



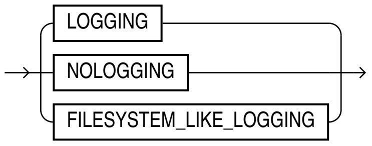
***LOB_deduplicate_clause*::=**



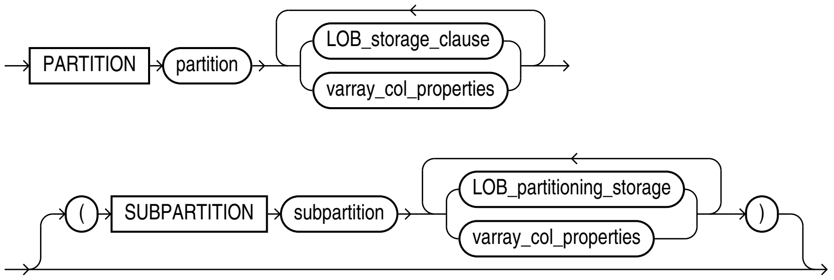
***LOB_compression_clause*::=**



***logging_clause*::=**

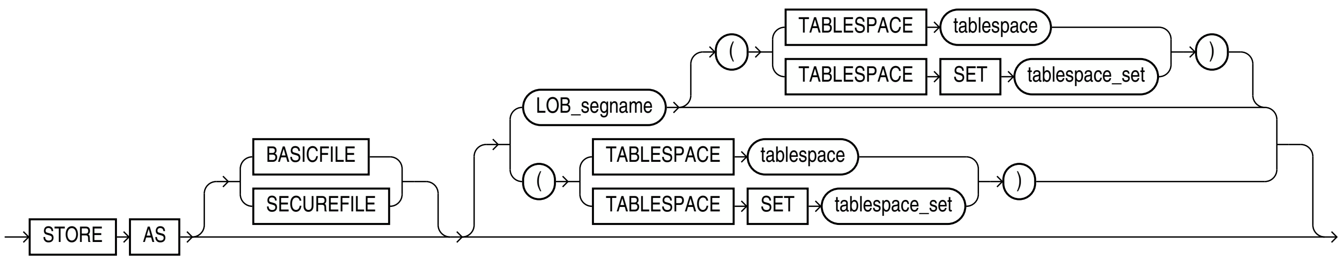


***LOB_partition_storage*::=**

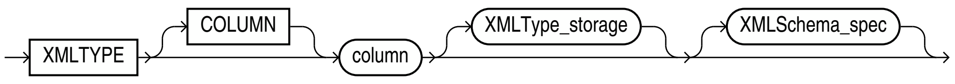


(*LOB_storage_clause*::=, *varray_col_properties*::=, *LOB_partitioning_storage*::=)

***LOB_partitioning_storage*::=**

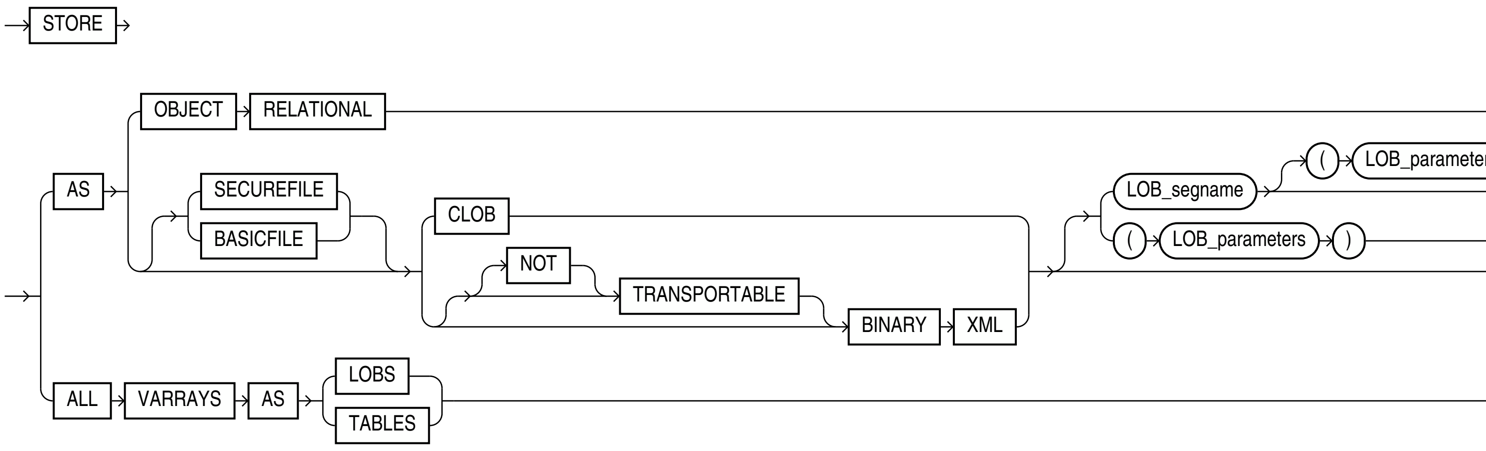


***XMLType_column_properties*::=**



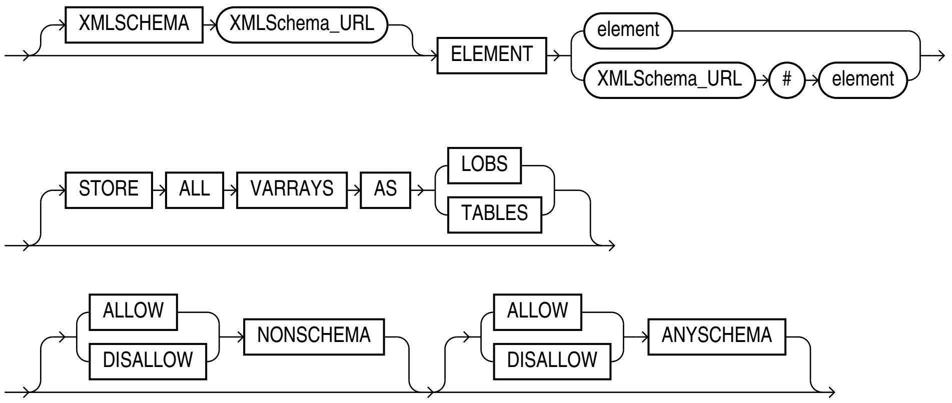
(*XMLType_storage::=*, *XMLSchema_spec::=*)

***XMLType_storage*::=**

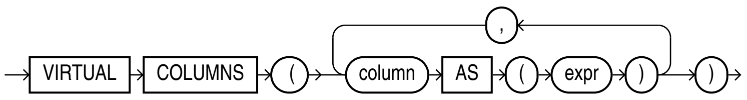


(*LOB_parameters::=*)

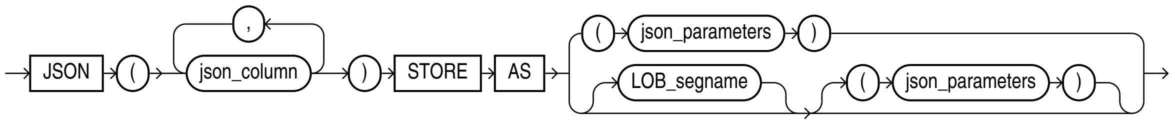
***XMLSchema_spec*::=**



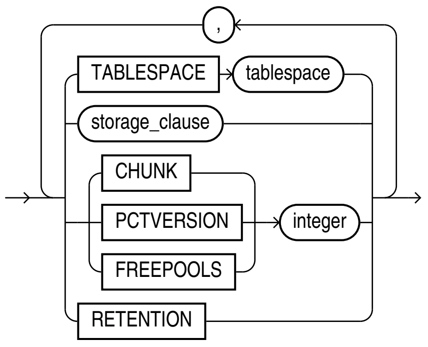
***XMLType_virtual_columns*::=**



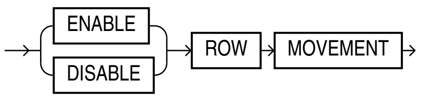
***JSON_storage_clause* ::=**



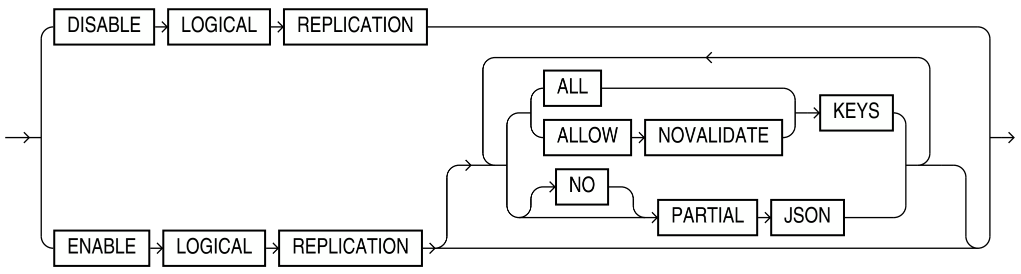
***JSON_parameters* ::=**



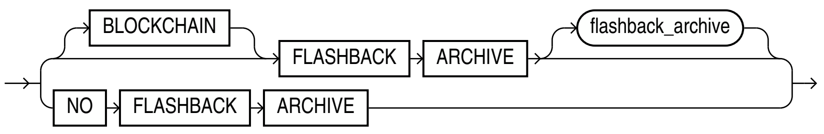
***row_movement_clause*::=**



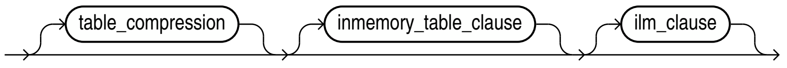
***logical_replication_clause*::=**



***flashback_archive_clause*::=**

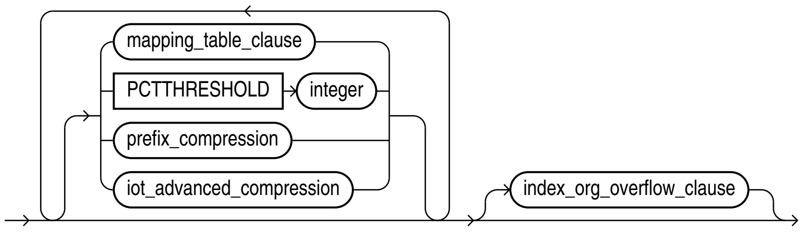


***heap_org_table_clause*::=**



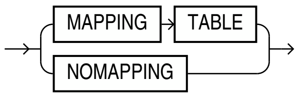
(*table_compression*::=, *inmemory_table_clause*::=, *ilm_clause*::=)

***index_org_table_clause*::=**

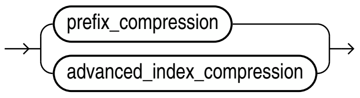


(*mapping_table_clauses*::=, *prefix_compression*::=, *index_org_overflow_clause*::=)

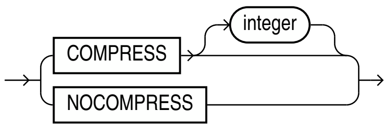
***mapping_table_clauses*::=**



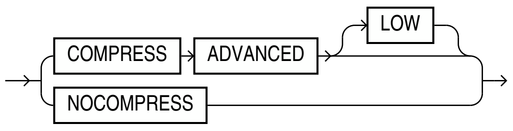
***index_compression*::=**



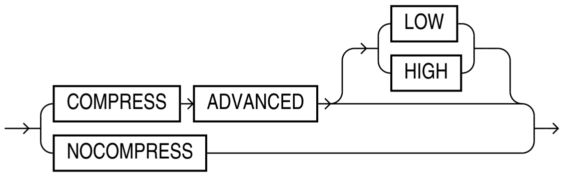
***prefix_compression*::=**



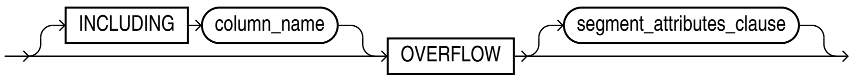
***iot_advanced_compression*::=**



***advanced_index_compression*::=**

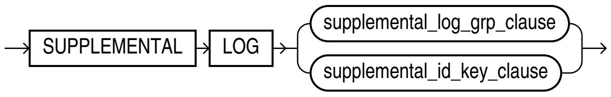


***index_org_overflow_clause*::=**

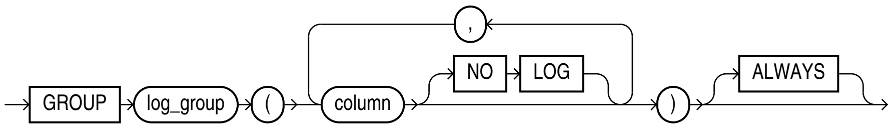


(*segment_attributes_clause*::=)

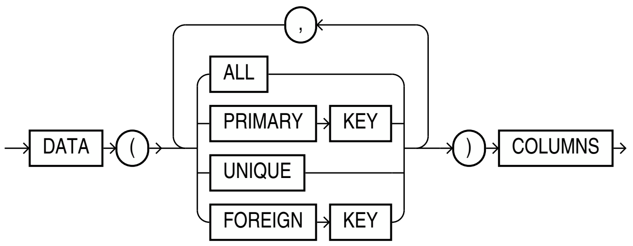
***supplemental_logging_props*::=**



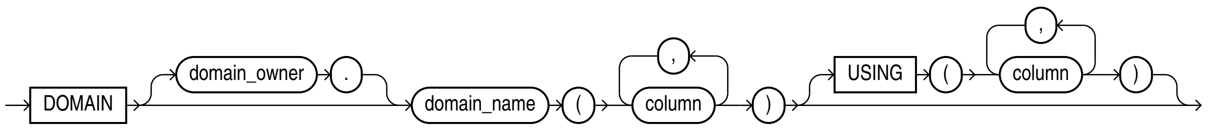
***supplemental_log_grp_clause*::=**



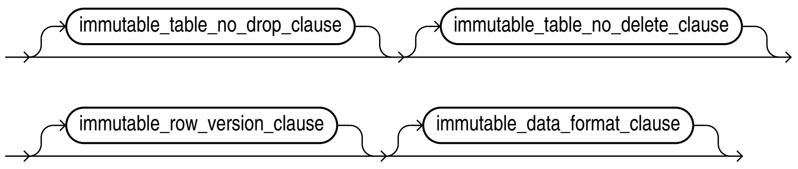
***supplemental_id_key_clause*::=**



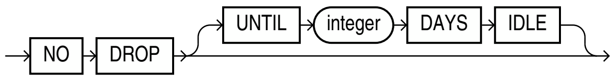
***domain_clause*::=**



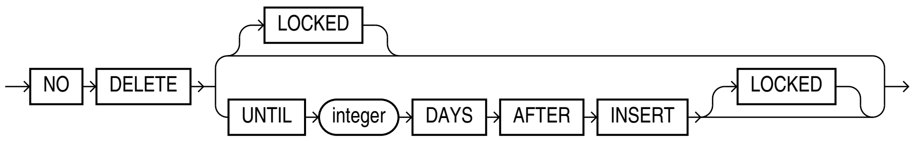
***immutable_table_clauses*::=**


immutable_table_no_drop_clause
immutable_table_no_delete_clause
immutable_row_version_clause
immutable_data_format_clause

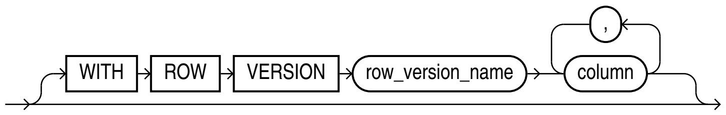
***immutable_table_no_drop_clause*::=**



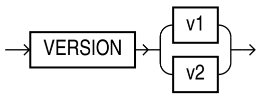
***immutable_table_no_delete_clause*::=**



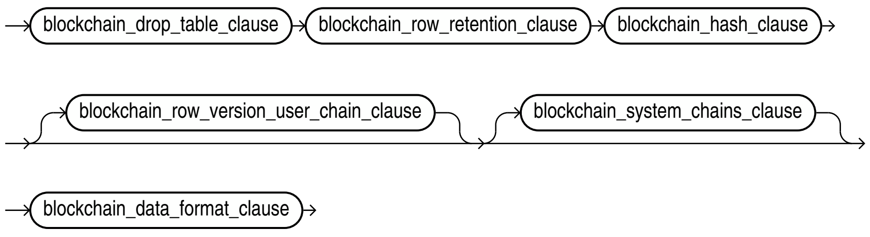
***immutable_row_version_clause*::=**



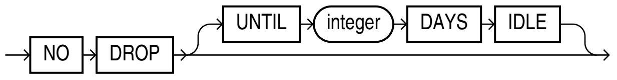
***immutable_data_format_clause*::=**



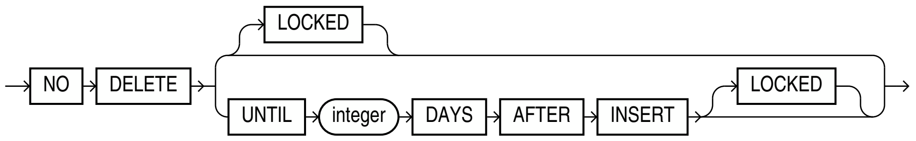
***blockchain_table_clauses*::=**


blockchain_drop_table_clause
blockchain_row_retention_clause
blockchain_hash_clause
blockchain_row_version_user_chain_clause
blockchain_system_chains_clause
blockchain_data_format_clause

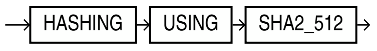
blockchain_drop_table_clause::=



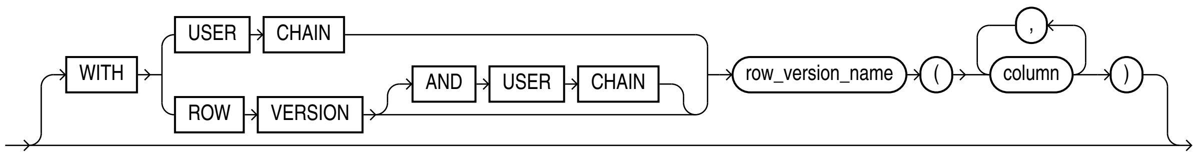
blockchain_row_retention_clause::=



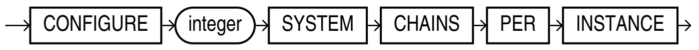
blockchain_hash_clause::=



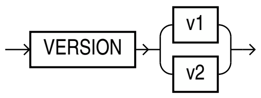
blockchain_row_version_user_chain_clause::=



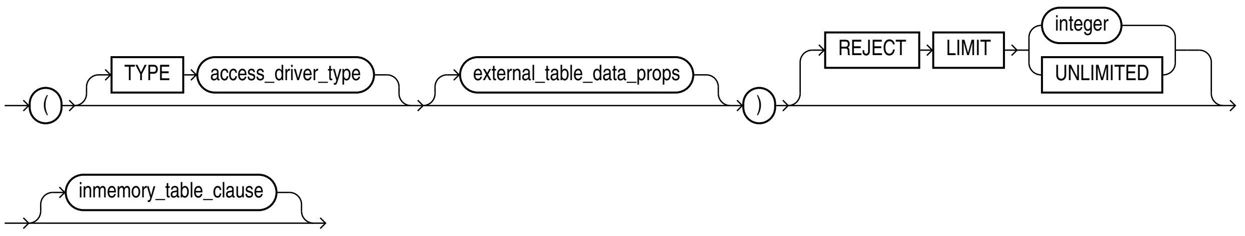
blockchain_system_chains_clause::=



blockchain_data_format_clause::=

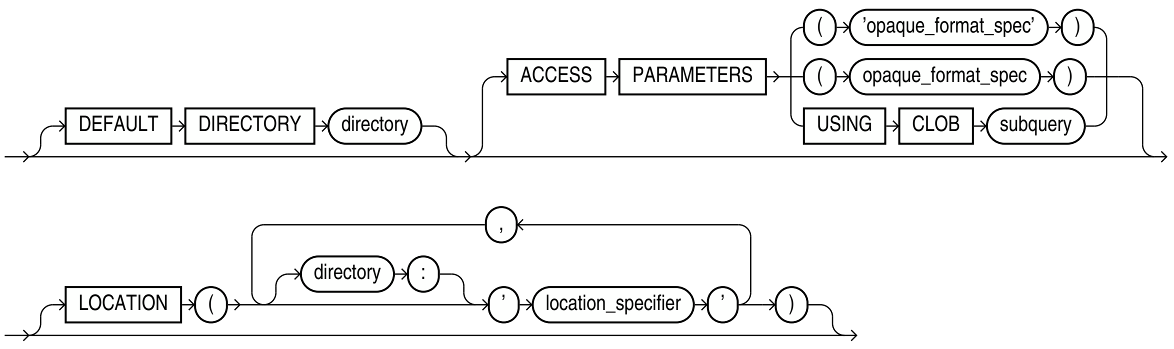


external_table_clause::=



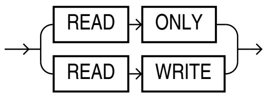
(*external_table_data_props*::=)

***external_table_data_props*::=**

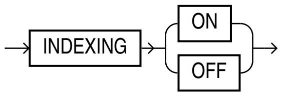


(`opaque_format_spec`: This clause specifies the access parameters for the `ORACLE_LOADER`, `ORACLE_DATAPUMP`, `ORACLE_HDFS`, and `ORACLE_HIVE` access drivers. See *Oracle Database Utilities* for descriptions of these parameters.)

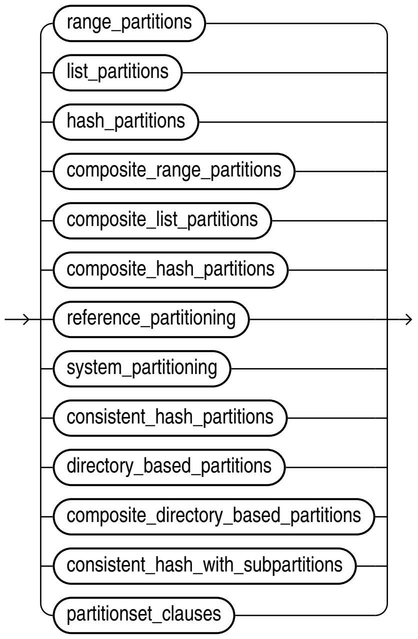
***read_only_clause*::=**



***indexing_clause*::=**

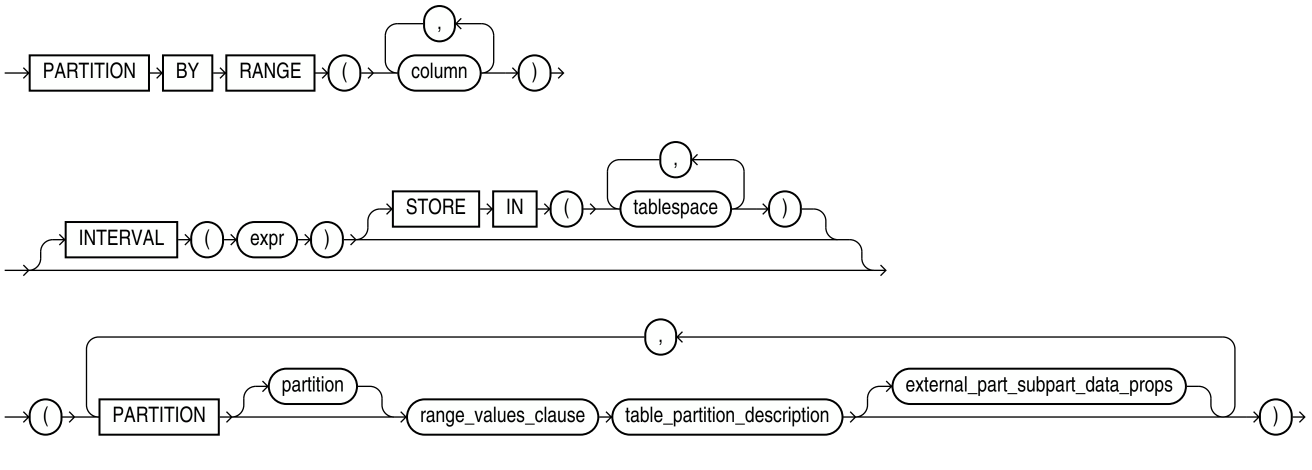


table_partitioning_clauses::=


range_partitions
list_partitions
hash_partitions
composite_range_partitions
composite_list_partitions
composite_hash_partitions
reference_partitioning
system_partitioning
consistent_hash_partitions
directory_based_partitions
composite_directory_based_partitions
consistent_hash_with_subpartitions
partitionset_clauses

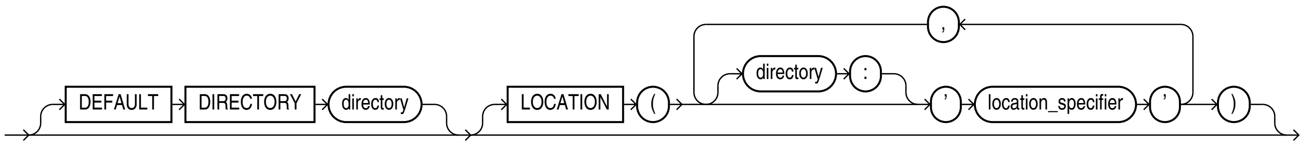
(*range_partitions*::=, *list_partitions*::=, *hash_partitions*::=, *composite_range_partitions*::=, *composite_list_partitions*::= *composite_hash_partitions*::=, *reference_partitioning*::=, *system_partitioning*::=, *consistent_hash_partitions*::=, *directory_based_partitions*::=,*composite_directory_based_partitions*::=,*consistent_hash_with_subpartitions*::=, *partitionset_clauses*::=)

range_partitions::=

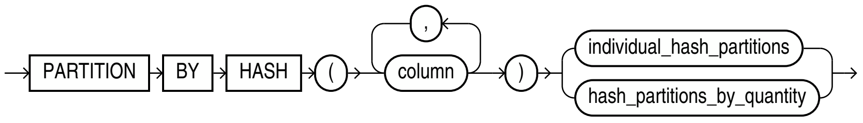


(*range_values_clause*::=, *table_partition_description*::=, *external_part_subpart_data_props*::=)

external_part_subpart_data_props::=

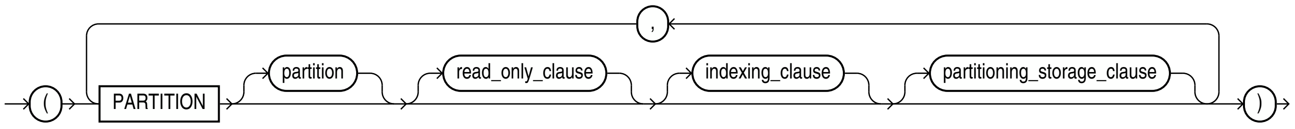


hash_partitions::=



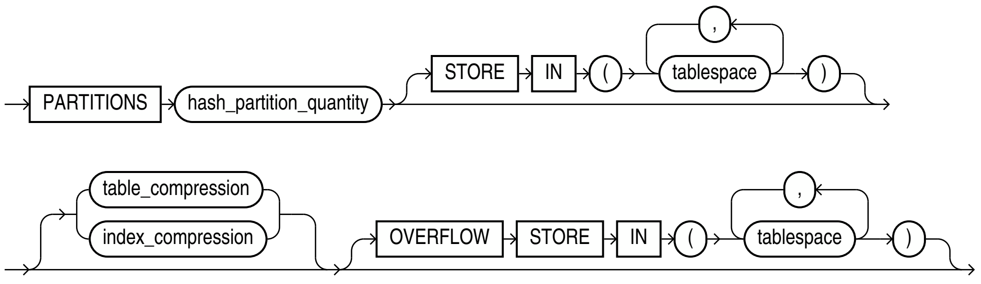
(*individual_hash_partitions*::=, *hash_partitions_by_quantity*::=)

individual_hash_partitions::=



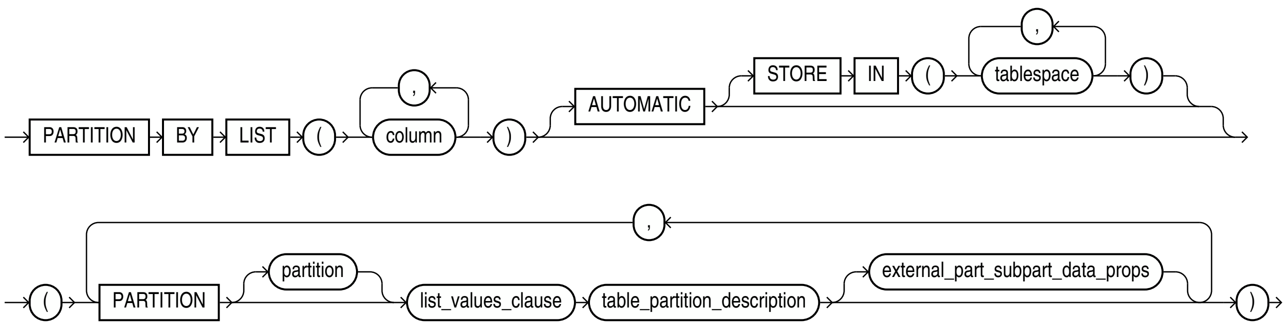
(*read_only_clause*::=, *indexing_clause*::=, *partitioning_storage_clause*::=)

hash_partitions_by_quantity::=



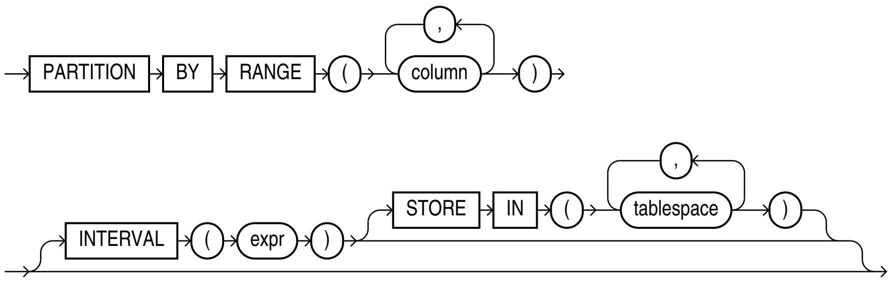
(*table_compression*::=, *index_compression*::=)

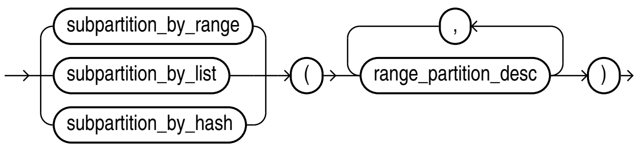
list_partitions::=



(*list_values_clause*::=, *table_partition_description*::=, *external_part_subpart_data_props*::=)

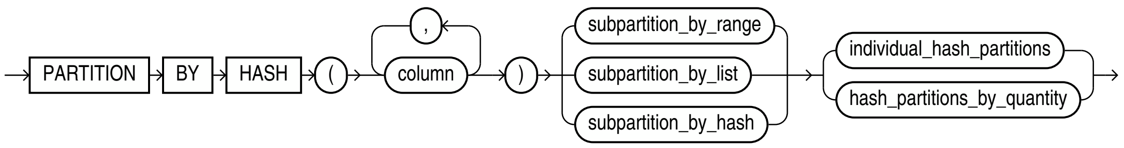
***composite_range_partitions*::=**





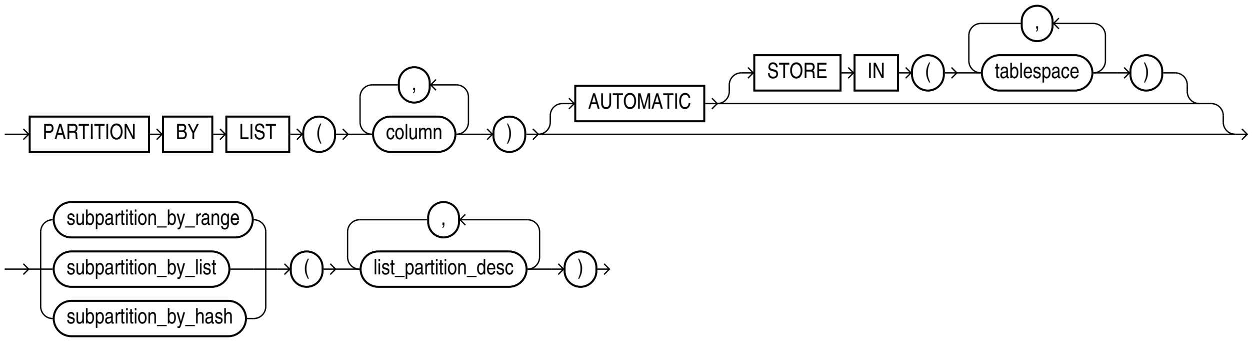
(*subpartition_by_range*::=. *subpartition_by_list*::=, *subpartition_by_hash*::=, *range_partition_desc*::=)

***composite_hash_partitions*::=**



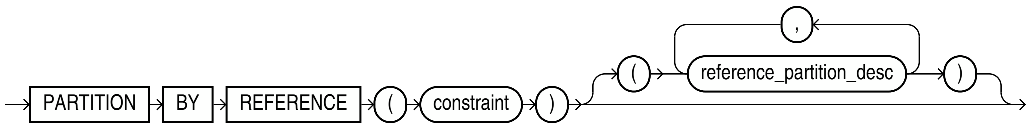
(*subpartition_by_range*::=, *subpartition_by_list*::=, *subpartition_by_hash*::=, *individual_hash_partitions*::=, *hash_partitions_by_quantity*::=)

***composite_list_partitions*::=**



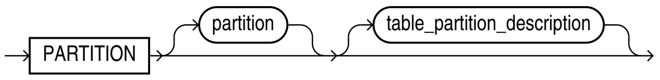
(*subpartition_by_range*::=. *subpartition_by_list*::=, *subpartition_by_hash*::=, *list_partition_desc*::=)

***reference_partitioning*::=**



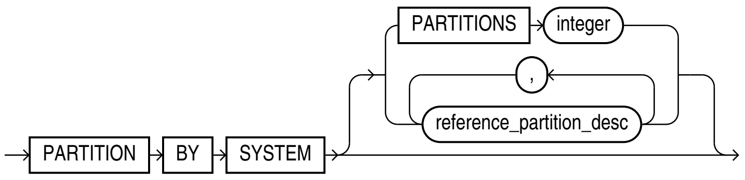
(*constraint*::=, *reference_partition_desc*::=)

***reference_partition_desc*::=**



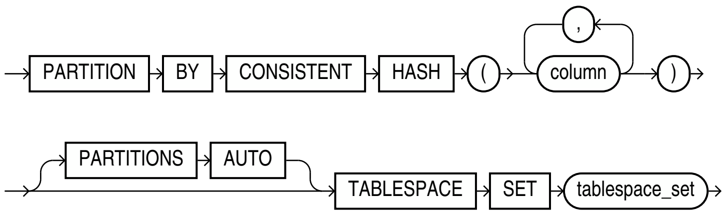
(*table_partition_description*::=)

***system_partitioning*::=**

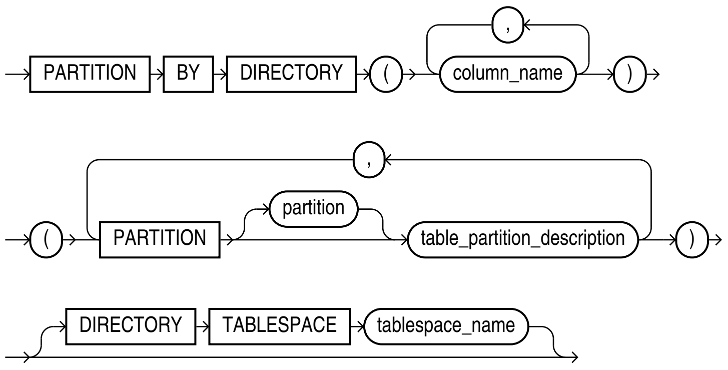


(*reference_partition_desc*::=)

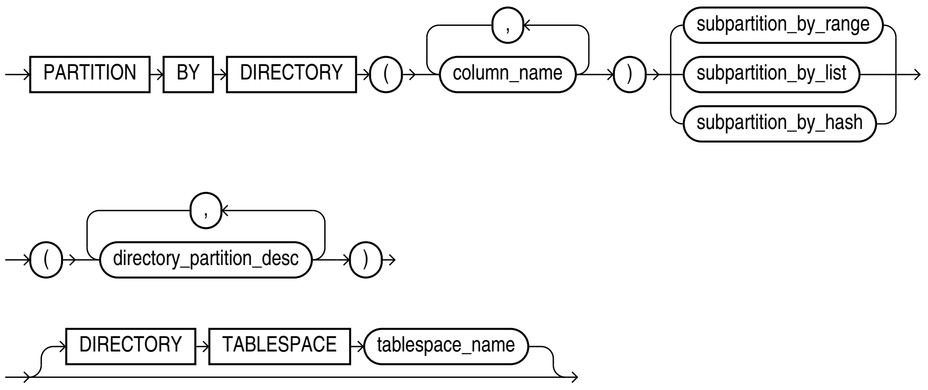
***consistent_hash_partitions*::=**



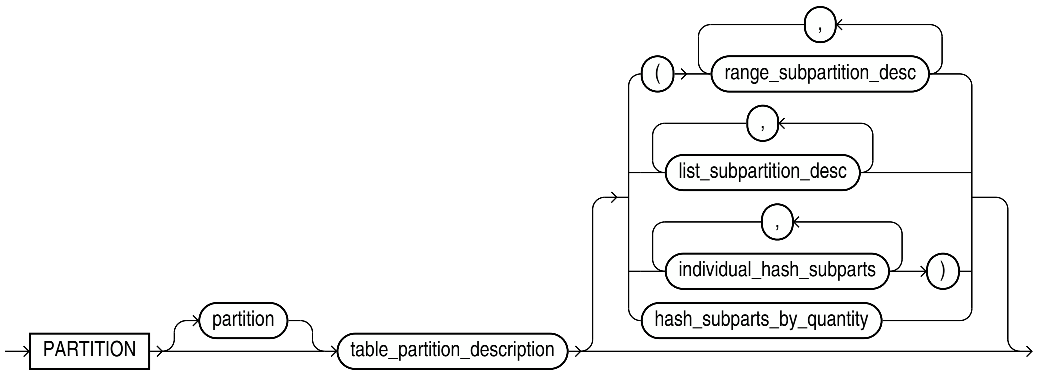
***directory_based_partitions*::=**



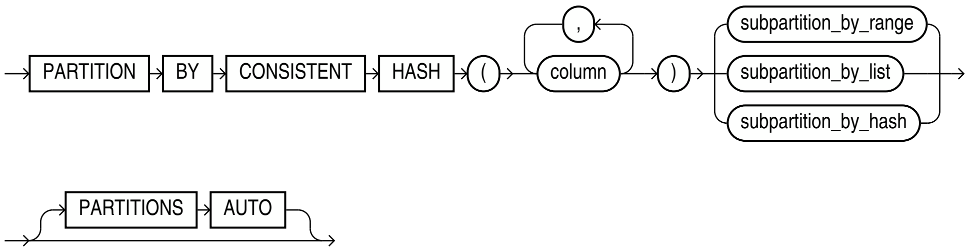
***composite_directory_based_partitions*::=**



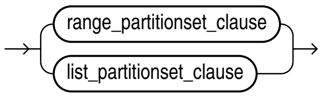
***directory_partition_desc*::=**



***consistent_hash_with_subpartitions*::=**

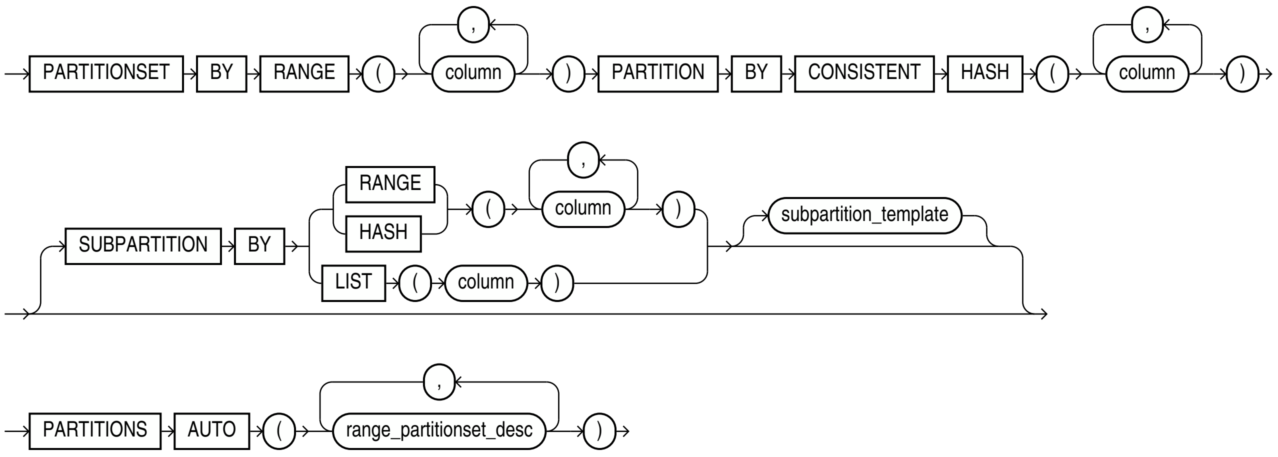


***partitionset_clauses*::=**

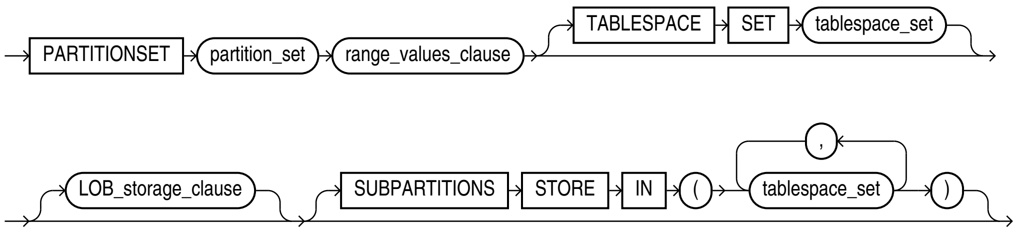


(*range_partitionset_clause*::=, *list_partitionset_clause*::=

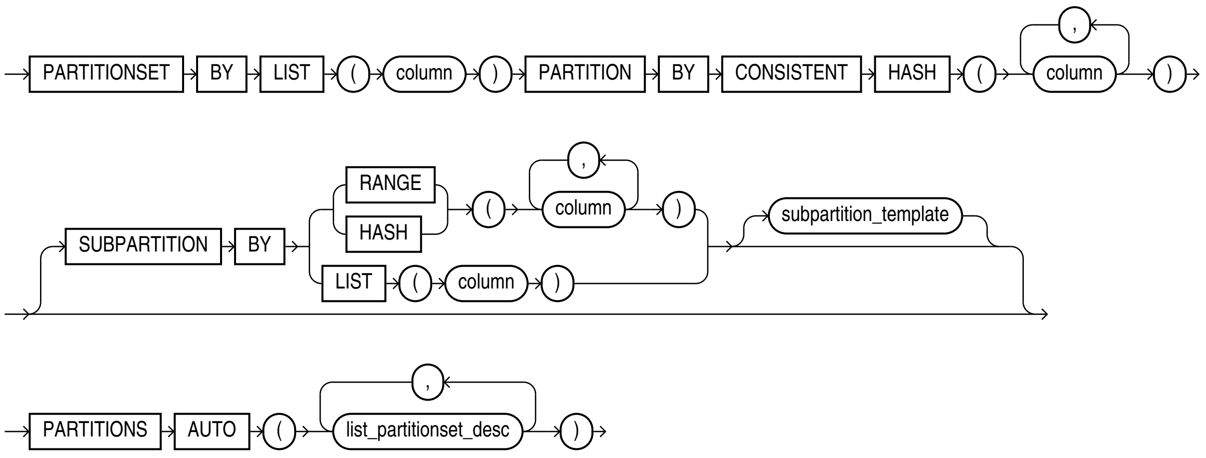
***range_partitionset_clause*::=**



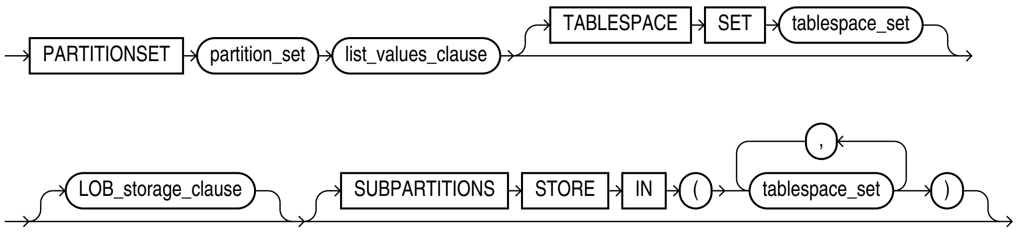
***range_partitionset_desc*::=**



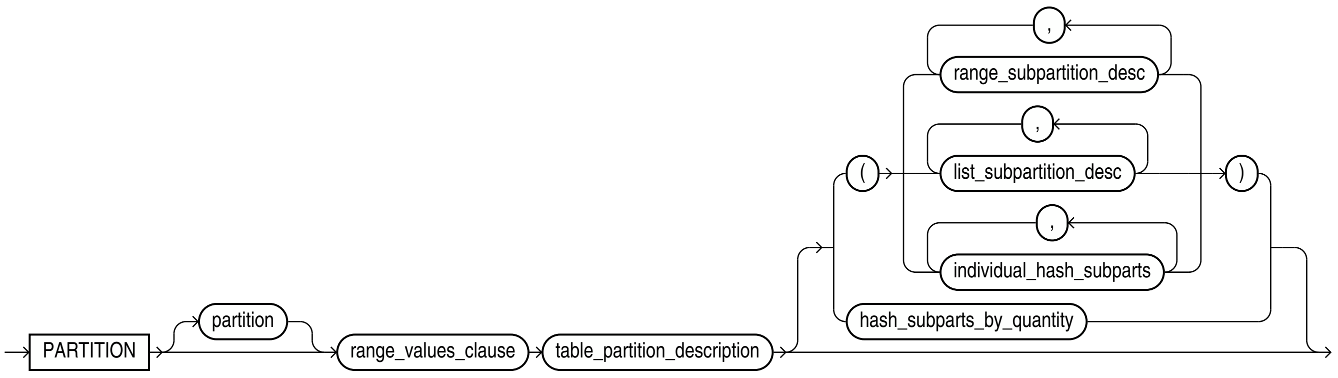
***list_partitionset_clause*::=**



***list_partitionset_desc*::=**

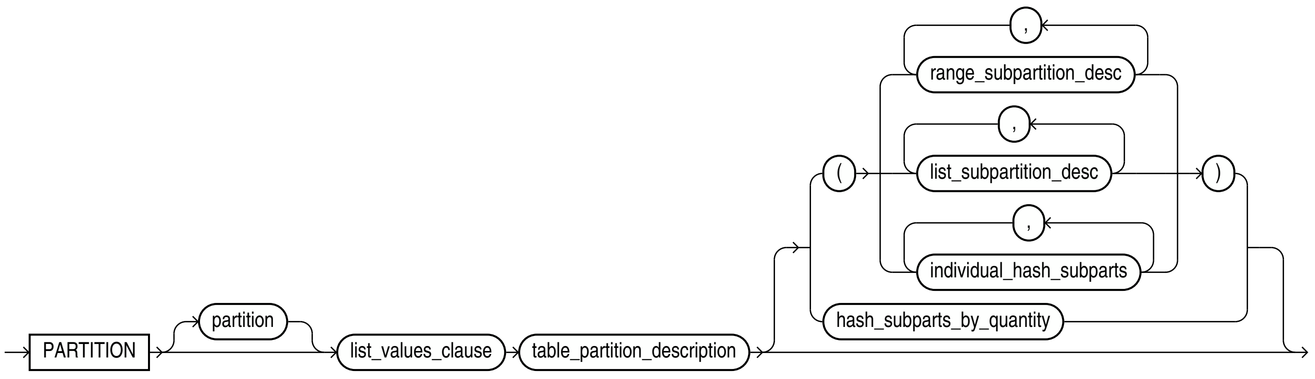


***range_partition_desc*::=**



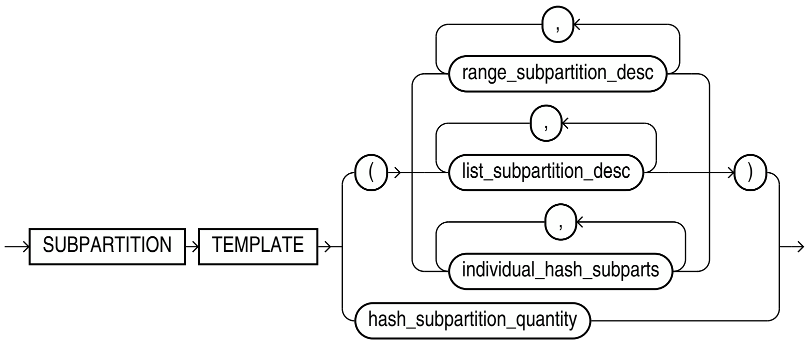
(*range_values_clause*::=, *table_partition_description*::=, *range_subpartition_desc*::=, *list_subpartition_desc*::=, *individual_hash_subparts*::=, *hash_subparts_by_quantity*::=)

***list_partition_desc*::=**



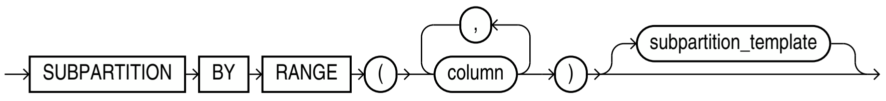
(*list_values_clause*::=, *table_partition_description*::=, *range_subpartition_desc*::=, *list_subpartition_desc*::=, *individual_hash_subparts*::=, *hash_subparts_by_quantity*::=)

***subpartition_template*::=**



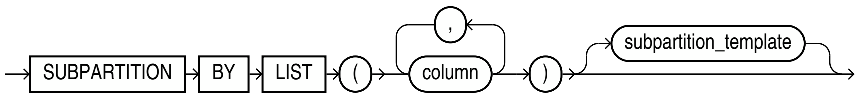
(*range_subpartition_desc*::=, *list_subpartition_desc*::=, *individual_hash_subparts*::=)

***subpartition_by_range*::=**



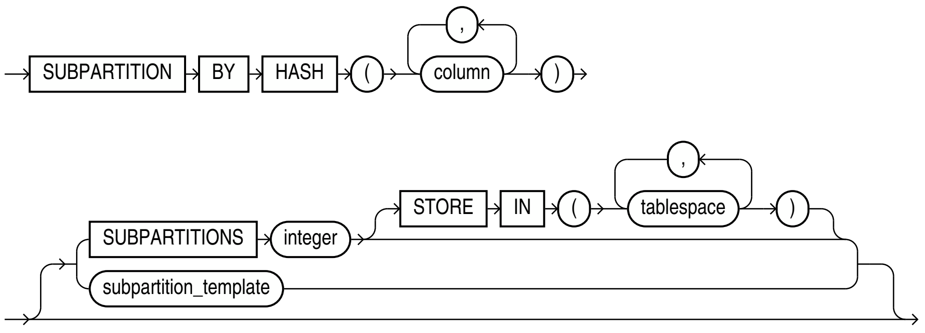
(*subpartition_template*::=)

***subpartition_by_list*::=**



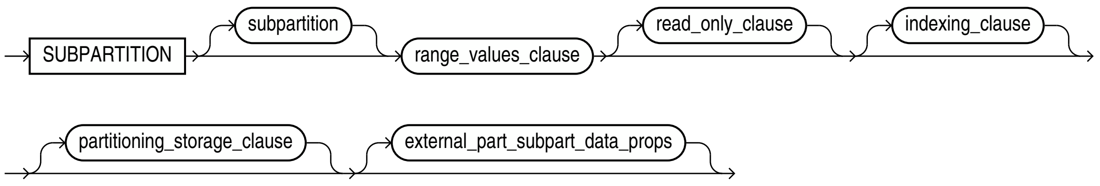
(*subpartition_template*::=)

***subpartition_by_hash*::=**



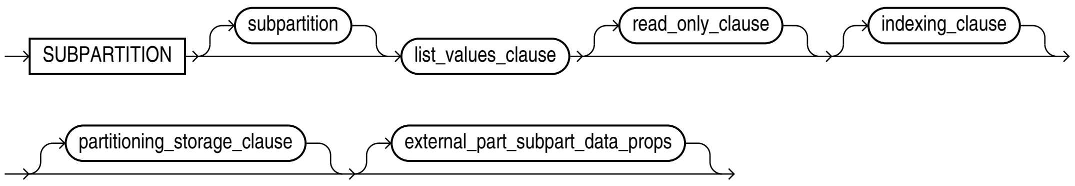
(*subpartition_template*::=)

***range_subpartition_desc*::=**



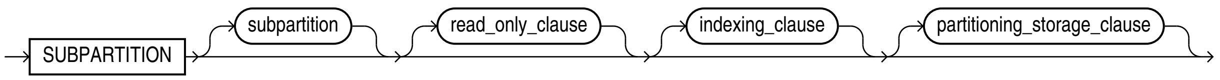
(*range_values_clause*::=, *read_only_clause*::=, *indexing_clause*::=, *partitioning_storage_clause*::=, *external_part_subpart_data_props*::=)

***list_subpartition_desc*::=**



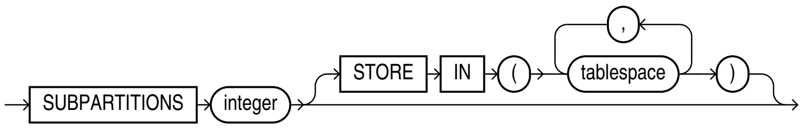
(*list_values_clause*::=, *read_only_clause*::=, *indexing_clause*::=, *partitioning_storage_clause*::=, *external_part_subpart_data_props*::=)

***individual_hash_subparts*::=**

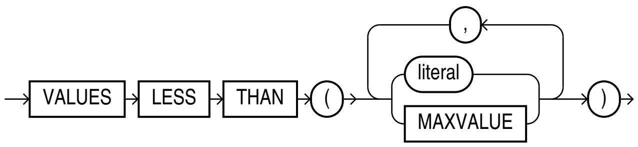


(*read_only_clause*::=, *indexing_clause*::=, *partitioning_storage_clause*::=)

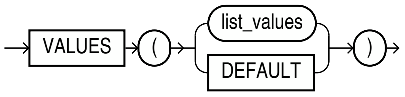
***hash_subparts_by_quantity*::=**



***range_values_clause*::=**

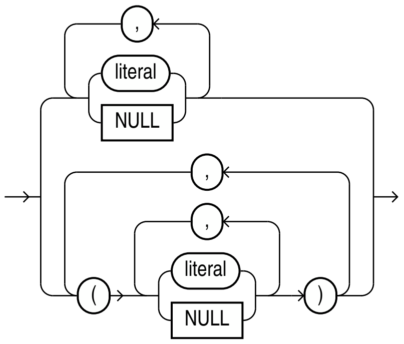


***list_values_clause*::=**

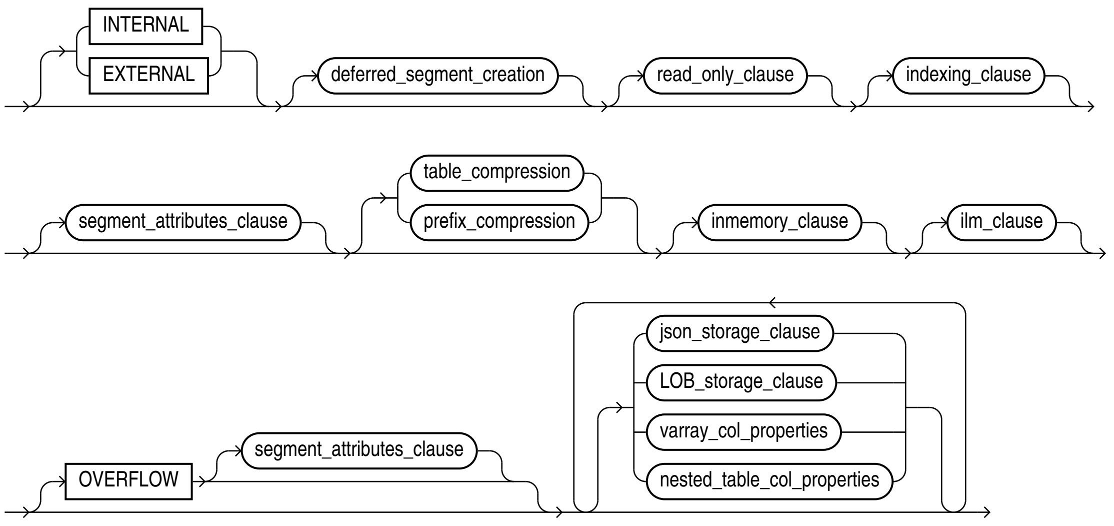


(*list_values*::=)

***list_values*::=**

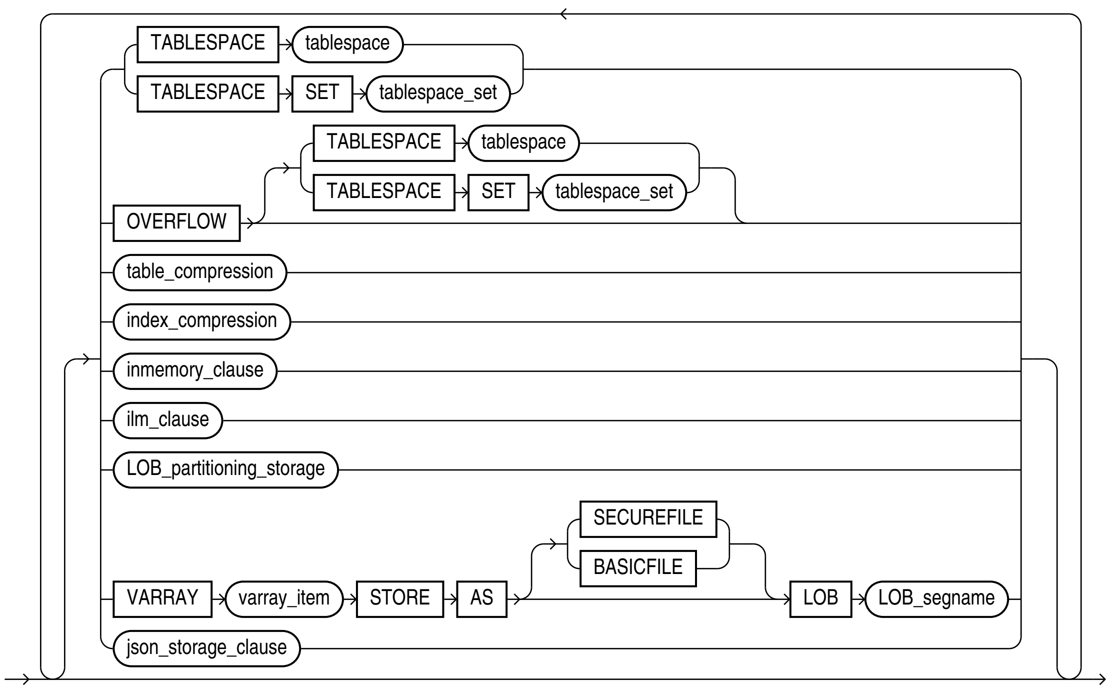


***table_partition_description*::=**



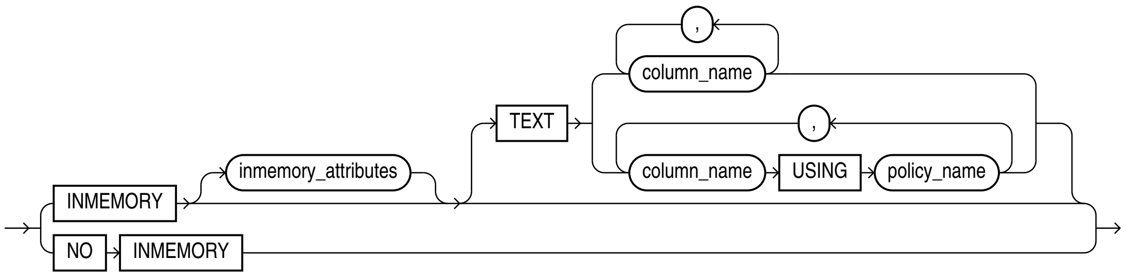
(*deferred_segment_creation*::=, *read_only_clause*::=, *indexing_clause*::=, *segment_attributes_clause*::=, *table_compression*::=, *prefix_compression*::=, *inmemory_clause*::=, *segment_attributes_clause*::=, *LOB_storage_clause*::=, *varray_col_properties*::=, *nested_table_col_properties*::=)

***partitioning_storage_clause*::=**



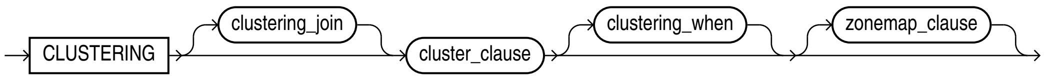
(*table_compression*::=, *index_compression*::=, *inmemory_clause*::=, *LOB_partitioning_storage*::=)

***inmemory_clause*::=**



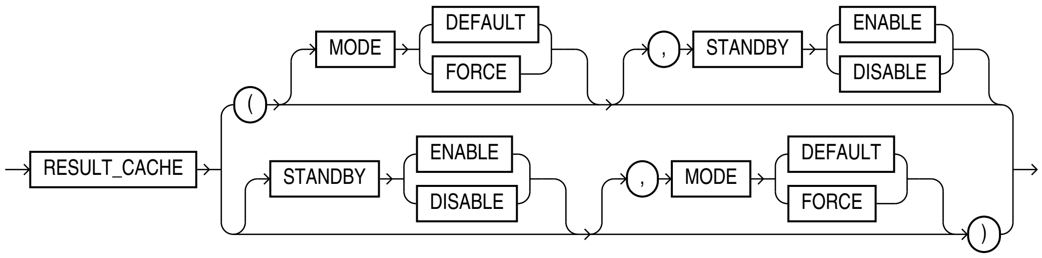
(*inmemory_memcompress*::=, *inmemory_attributes*::=)

***attribute_clustering_clause*::=**

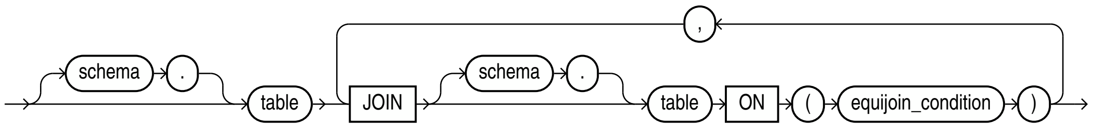


(*clustering_join*::=, *cluster_clause*::=, *clustering_when*::=, *zonemap_clause*::=)

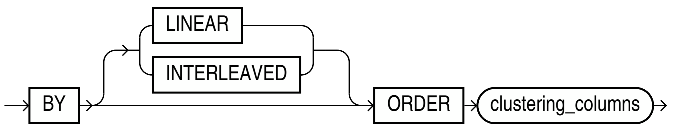
result_cache_clause



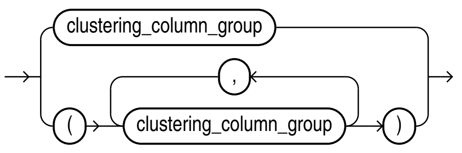
***clustering_join*::=**



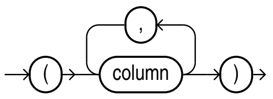
***cluster_clause*::=**



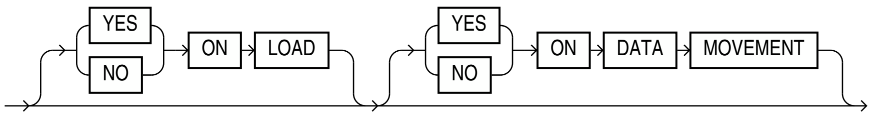
clustering_columns::=



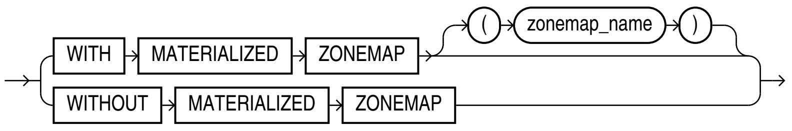
clustering_column_group::=



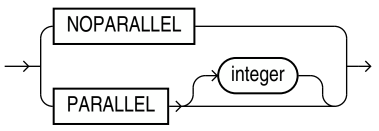
clustering_when::=



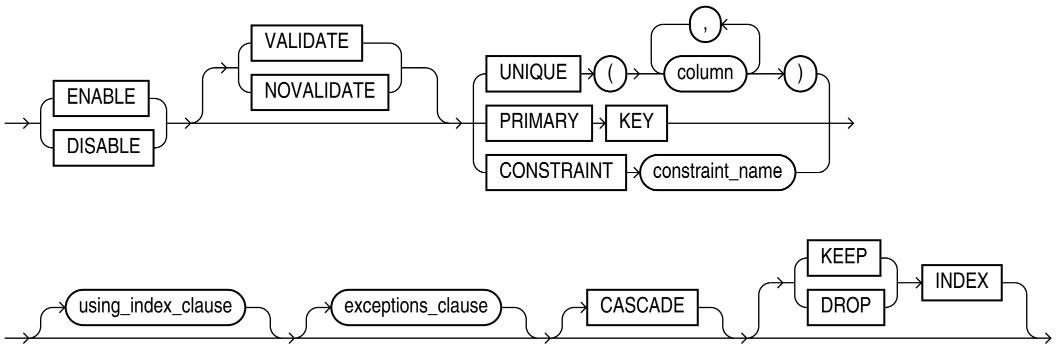
zonemap_clause::=



parallel_clause::=

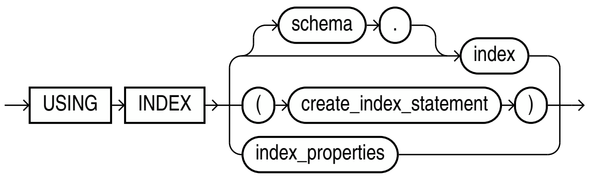


enable_disable_clause::=



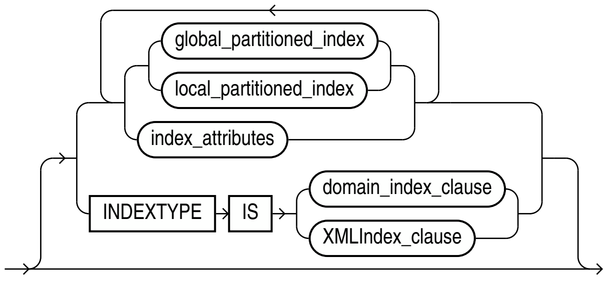
(*using_index_clause::=*, `exceptions_clause` not supported in `CREATE TABLE` statements)

***using_index_clause*::=**



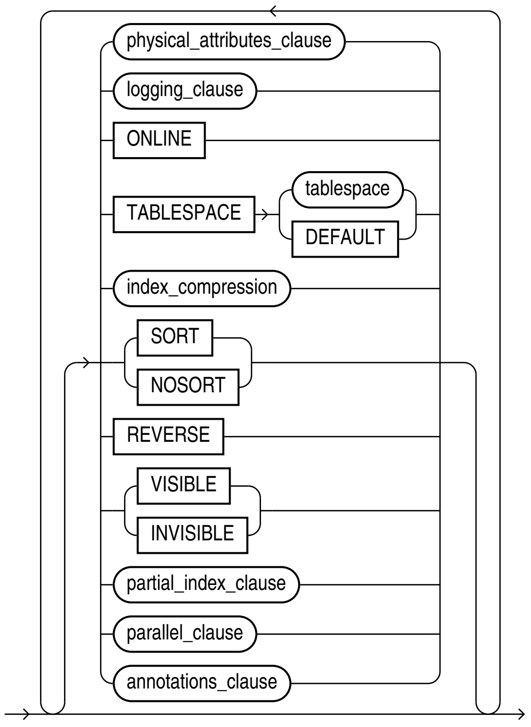
(*create_index::=*, *index_properties::=*)

***index_properties*::=**



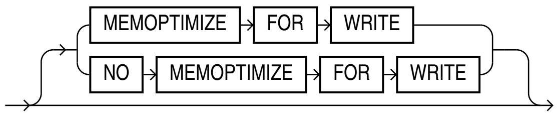
(*global_partitioned_index::=*, *local_partitioned_index::=*—part of `CREATE INDEX`, *index_attributes::=*, `domain_index_clause` and `XMLIndex_clause`: not supported in `using_index_clause`)

***index_attributes*::=**



(*physical_attributes_clause*::=, *logging_clause*::=, *index_compression*::=, `partial_index_clause` and `parallel_clause`: not supported in `using_index_clause`)

memoptimize_write_clause



Semantics

GLOBAL TEMPORARY

Specify `GLOBAL TEMPORARY` to create a temporary table, whose **definition** is visible to all sessions with appropriate privileges. The **data** in a temporary table is visible only to the session that inserts the data into the table.

When you first create a temporary table, its metadata is stored in the data dictionary, but no space is allocated for table data. Space is allocated for the table segment at the time of the first DML operation on the table. The temporary table definition persists in the same way as the definitions of regular tables, but the table segment and any data the table contains are either **session-specific** or **transaction-specific** data. You specify whether the table segment and data are session- or transaction-specific with the ON COMMIT clause.

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 with an `INSERT` operation on the table. A session becomes unbound to a temporary table with a `TRUNCATE` statement or at session termination, or, for a transaction-specific temporary table, by issuing a `COMMIT` or `ROLLBACK` statement.

PRIVATE TEMPORARY

Specify `PRIVATE TEMPORARY` to create a private temporary table.

A private temporary table differs from a temporary table in that its **definition** and **data** are visible ***only*** within the session that created it.

Use the `ON COMMIT` clause to define the scope of a private temporary table: either **transaction** or **session**.

The `ON COMMIT` clause used with the keywords `DROP DEFINITION` creates a transaction-specific table whose data **and** definition are dropped when the transaction commits. This is the default behavior.

The `ON COMMIT` clause used with keywords `PRESERVE DEFINITION` creates a session-specific table whose definition is preserved when the transaction commits.

See here for usage details of the ON COMMIT clause.

Three DDL statements are supported for private temporary tables: `CREATE`, `DROP`, and `TRUNCATE`.

Restrictions on Temporary Tables

Temporary tables are subject to the following restrictions:

- Temporary tables cannot be partitioned, clustered, or index organized.

- You cannot specify any foreign key constraints on temporary tables.
- Temporary tables cannot contain columns of nested table.
- You cannot specify the following clauses of the *`LOB_storage_clause`*: `TABLESPACE`, *`storage_clause`*, or *`logging_clause`*.
- Parallel `UPDATE`, `DELETE` and `MERGE` are not supported for temporary tables.
- The only part of the *`segment_attributes_clause`* you can specify for a temporary table is `TABLESPACE`, which allows you to specify a single temporary tablespace.
- Distributed transactions are not supported for temporary tables.
- A temporary table cannot contain `INVISIBLE` columns.

Restrictions on Private Temporary Tables

In addition to the general limitations of temporary tables, private temporary tables are subject to the following restrictions:

You must be a user other than `SYS` to create private temporary tables.

You cannot specify the following constraints on private temporary tables that are permitted on global temporary tables:

- `PRIMARY KEY` constraint
- `UNIQUE` constraint
- `CHECK` constraint
- `NOT NULL` constraint
- The name of private temporary tables must *always* be prefixed with whatever is defined with the `init.ora` parameter `PRIVATE_TEMP_TABLE_PREFIX`. The default is `ORA$PTT_`.
- You cannot create indexes, materialized views, or zone maps on private temporary tables.
- You cannot define column with default values.
- You cannot reference private temporary tables in any permanent object, e.g. views or triggers.
- Private temporary tables are not visible through database links.
- You cannot associate table columns of private temporary tables with a domain using `CREATE TABLE`. Doing so results in the following error: `Cannot associate a private temporary table with a domain.`

See Also:

Oracle Database Concepts for information on temporary tables and "Creating a Table: Temporary Table Example"

SHARDED

Specify `SHARDED` to create a sharded table.

This clause is valid only if you are using Oracle Sharding, which is a data tier architecture in which data is horizontally partitioned across independent databases. Each database in such configuration is called a shard. All of the shards together make up a single logical database, which is referred to as a sharded database (SDB). Horizontal partitioning involves splitting a table across shards so that each shard contains the table with the same columns but a different subset of rows. A table split up in this manner is called a sharded table.

When you create a sharded table, you must specify a tablespace set in which to create the table. There is no default tablespace set for sharded tables. See CREATE TABLESPACE SET for more information.

Oracle Sharding is based on the Oracle Partitioning feature. Therefore, a sharded table must be a partitioned or composite-partitioned table. When creating a sharded table, you must specify one of the `table_partitioning_clauses`. See *table_partitioning_clauses* for the full semantics of these clauses.

Restrictions on Sharded Tables

The following restrictions apply to sharded tables:

- In system-managed sharding you can create multiple root tables (and therefore table families) without throwing `ORA-02530` , when the `CREATE SHARDED TABLE` statement does not contain a `PARTITION BY REFERENCE` or `PARENT` clause and there is already a root table in existence.
- A sharded table cannot be a temporary table or an index-organized table.
- A sharded table cannot contain a nested table column or an identity column.
- You cannot specify a tablespace for a sharded system or a composite sharded table with the `TABLESPACE`clause, because system or composite sharded tables require tablespace sets.
- You cannot create tablespace sets in a user-defined sharding environment.
- A sharded tablespace is required for sharded tables. Normal tablespaces are not supported.
- You cannot specify the same tablespace for multiple partitions of the sharded table. This rule applies to subpartitions also. The same tablespace cannot be specified for subpartitions belonging to different partitions of a sharded table.
- You must specify a tablespace per partition of non-reference partitioned sharded tables.
- For user defined sharding the partition method must be range or list. Autolist and Interval partitioning is not supported.
- The list partition method can only have one partitioning column.
- Default partitions are not supported in list partitioned tables.
- NULL partitions are not supported in list partitioned tables.
- A primary key constraint defined on a sharded table must contain the sharding columns. A foreign key constraint on a column of a sharded table referencing a duplicated table column is not supported.
- System partitioning and interval-range partitioning are not supported for sharded tables.
- You cannot specify a virtual (expression) column in a sharded table in the `PARTITION BY` or `PARTITIONSET BY` clauses.
- `XMLType` columns for sharded tables are defaulted to `TRANSPORTABLE BINARY XML`, which is the only storage type allowed. Any other storage clause for `XMLType` will throw an error.

See Also:

- *Using Oracle Sharding*
- *Oracle Database Administrator's Guide*

DUPLICATED

This clause is valid only if you are using Oracle Sharding. Specify `DUPLICATED` to create a duplicated table, which is duplicated on all shards. It can be a nonpartitioned table or partitioned table.

Duplicated tables are not tied to any table family.

Restrictions on Duplicated Tables

The following restrictions apply to duplicated tables:

- A duplicated table cannot contain a `LONG` column.
- The maximum number of non-primary key columns in a duplicated table is 999.
- `XMLType` columns for duplicated tables are defaulted to `TRANSPORTABLE BINARY XML`, which is the only storage type allowed. Any other storage clause for `XMLType` will throw an error.
- A duplicated table cannot be a temporary table.
- A duplicated table cannot be a reference-partitioned table or a system-partitioned table.
- You cannot specify `NOLOGGING` or `PARALLEL` for a duplicated table.
- You cannot enable a duplicated table for the In-Memory Column Store.

IMMUTABLE

Specify the `IMMUTABLE` keyword to create an append-only table that protects data from unauthorized modification by insiders.

You can create a blockchain table that emphasizes its immutability by using the keywords `IMMUTABLE BLOCKCHAIN` in `CREATE TABLE`.

You must specify the mandatory *`immutable_table_clauses`* when you create an immutable table using the `CREATE IMMUTABLE TABLE` statement.

Prerequistes

- The `COMPATIBLE` initialization parameter must be set to 19.11.0.0 or higher.
- The `CREATE TABLE` system privilege is required to create immutable tables in your own schema. The `CREATE ANY TABLE` system privilege is required to create immutable tables in another user's schema.
- The `NO DROP` and `NO DELETE` clauses are mandatory.

BLOCKCHAIN

Specify the `BLOCKCHAIN` keyword to create a blockchain table.

You must specify the mandatory *`blockchain_table_clauses`* when you create a blockchain table using the `CREATE BLOCKCHAIN TABLE` statement.

When you create a blockchain table, an entry is created in the dictionary table `blockchain_table$` owned by `SYS` .

Restrictions

The following `CREATE TABLE` clauses are disallowed with the creation of blockchain tables:

- `ORGANIZATION INDEX`
- `ORGANIZATION EXTERNAL`
- `NESTED TABLE`

See Also:

- *Managing Immutable Tables*
- *Managing Blockchain Tables*

IF NOT EXISTS

Specifying `IF NOT EXISTS` has the following effects:

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

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

schema

Specify the schema to contain the table. If you omit *`schema`*, then the database creates the table in your own schema.

table

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

See Also:

Oracle Database Administrator's Guide for more on sharded tables.

SHARING

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

- `METADATA` - A metadata link shares the table's metadata, but its data is unique to each container. This type of table is referred to as a **metadata-linked application common object**.
- `DATA` - A data link shares the table, and its data is the same for all containers in the application container. Its data is stored only in the application root. This type of table is referred to as a **data-linked application common object**.
- `EXTENDED DATA` - An extended data link shares the table, and its data in the application root is the same for all containers in the application container. However, each application PDB in the application container can store data that is unique to the application PDB. For this type of table, data is stored in the application root and, optionally, in each application PDB. This type of table is referred to as an **extended data-linked application common object**.
- `NONE` - The table 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 table. If the `DEFAULT_SHARING` initialization parameter does not have a value, then the default is `METADATA`.

When creating a relational table, you can specify `METADATA`, `DATA`, `EXTENDED DATA`, or `NONE`.

When creating an object table or an `XMLTYPE` table, you can specify only `METADATA` or `NONE`.

You cannot change the sharing attribute of a table 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

relational_table

This clause lets you create a relational table.

relational_properties

The relational properties describe the components of a relational table.

column_definition

The *`column_definition`* lets you define the characteristics of the column.

Specifying *column_definition* with AS *subquery*

If you specify the `AS` *`subquery`* clause, and each column returned by *`subquery`* has a column name or is an expression with a specified column alias, then you can omit the *`column_definition`* clause. In this case, the names of the columns of table are the same as the names of the columns returned by *`subquery`*. The exception is creating an index-organized table, for which you must specify the *`column_definition`* clause, because you must designate a primary key column. Regardless of the table type, if you specify the *`column_definition`* clause and the `AS` *`subquery`* clause, then you must omit *`datatype`* from the *`column_definition`* clause.

column

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

If you also specify `AS` *`subquery`*, then you can omit *`column`* and *`datatype`* unless you are creating an index-organized table. If you specify `AS` *`subquery`* when creating an index-organized table, then you must specify *`column`*, and you must omit *`datatype`*.

The absolute maximum number of columns in a table is 1000, if the `MAX_COLUMNS` initialization parameter = `STANDARD`, or 4096 columns if `MAX_COLUMNS` = `EXTENDED`. See *Oracle Database Reference* for more on the `MAX_COLUMNS` initialization parameter.

When you create an object table or a relational table with columns of object, nested table, varray, or REF type, Oracle Database maps the columns of the user-defined types to relational columns, in effect creating hidden columns that count toward the 1000-column limit. A relational column that stores a user-defined type attribute inherits the collation property of the attribute. In Oracle Database 12*c* Release 2 (12.2), user-defined types are created using the pseudo-collation property `USING_NLS_COMP` and their corresponding relational columns inherit this property.

datatype_domain

datatype

Specify the data type of a column in *`datatype`*.

In general, you must specify *`datatype`*. However, the following exceptions apply:

- You must omit *`datatype`* if you specify the `AS` *`subquery`* clause.
- You can also omit *`datatype`* if the statement designates the column as part of a foreign key in a referential integrity constraint. Oracle Database automatically assigns to the column the data type of the corresponding column of the referenced key of the referential integrity constraint.

Restrictions on Table Column Data Types

- Do not create a table with `LONG` columns. Use LOB columns (`CLOB`, `NCLOB`, `BLOB`) instead. `LONG` columns are supported only for backward compatibility.
- 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 `LONG` columns and on Oracle-supplied data types

You can specify a user-defined data type as non-persistable when creating or altering the data type. Instances of non-persistable types cannot persist on disk. See CREATE TYPE for more on user-defined data types declared as non-persistable types.

domain_clause

Use this clause to associate columns with a domain. You can associate non-strict domains with a table column with a compatibile type with any limit. For example, a number(10) domain can be assigned to columns with number(9) or number(11). For strict domains you can only associate their columns with table columns with a compatible type and identical type limits. For example, a number(10) domain column can only be assigned to numeric columns with precision 10, like decimal(10) or numeric(10).

The position of columns in this clause is the same as those in the domain. The number of columns listed must be the same as in *domain_name*.

Each column can belong to at most one domain. If a column is in two or more domains, the statement will error.

If you specify the data type, you must use the DOMAIN keyword. You can omit the DOMAIN keyword, if you omit the data type and just use the domain owner or domain name.

USING

The USING clause defines the discriminant columns in flexible domains. This clause is mandatory when associating columns with a flexible domain.

The columns in the DOMAIN and USING clauses must be different.

RESERVABLE

Reservable columns provide for lock-free reservations. Lock-free reservations allow other concurrent transactions updating the reservable columns to proceed without being blocked. Lock-free reservations hold locks on hot data for short intervals of time and only when the value is modified during the commit of the transaction.

Specify RESERVABLE on a column to make it reservable on columns with numeric data type.

Guidelines and Restrictions for Reservable Columns

- The schema definition of user tables declares the reservable columns with the RESERVABLE keyword.
- A reservable column can be specified only on columns of the following numeric data types: NUMBER, INTEGER, and FLOAT.
- A reservable column cannot be a Primary Key or an identity column (or virtual (expression) column) because the reservable column is an aggregate type.
- A user table can have at most ten reservable columns.
- User tables that have reservable columns must have a Primary Key.
- Indexes are not supported on reservable columns.
- Composite reservable columns are not allowed. Reservable columns can be included only in CHECK constraints expression.
- The CHECK constraint can be at the column-level or table-level. User-defined operational constraints are used for reservable columns to ensure application correctness.
- Partitioning cannot be made on reservable columns. Transactions with pending reservations must finalize before you can drop the reservable column or mark the column as UNUSED.

See Also:

- *Using Lock-Free Reservation* of the *Database Development Guide*.
- Columns section in *Tables and Table Clusters* of *Database Concepts*.

COLLATE

The COLLATE clause lets you specify a data-bound collation for the column.

For *column_collation_name*, specify a valid named collation or pseudo-collation. For columns of data type CLOB or NCLOB, the only allowed value for *column_collation_name* is the pseudo-collation USING_NLS_COMP.

If you omit this clause, then the column is assigned:

- the pseudo-collation USING_NLS_COMP, if the column has the data type CLOB or NCLOB, or
- the collation of the corresponding parent key column, if the column belongs to a foreign key, or
- the default collation for the table as it stands at the time the column is created.

Refer to the DEFAULT COLLATION clause for more information on the default collation for a table.

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.

SORT

The SORT keyword is valid only if you are creating this table as part of a hash cluster and only for columns that are also cluster columns.

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 operations on the clustered data.

See Also:

- "CLUSTER Clause" for information on creating a cluster table
- *Managing Hash Clusters*

VISIBLE | INVISIBLE

Use this clause to specify whether *column* is VISIBLE or INVISIBLE. The default is VISIBLE.

INVISIBLE columns are user-specified hidden columns. To display or assign a value to an INVISIBLE column, you must specify its name explicitly. For example:

- The SELECT * syntax will not display an INVISIBLE column. However, if you include an INVISIBLE column in the select list of a SELECT statement, then the column will be displayed.
- You cannot implicitly specify a value for an INVISIBLE column in the VALUES clause of an INSERT statement. You must specify the INVISIBLE column in the column list.
- You must explicitly specify an INVISIBLE column in Oracle Call Interface (OCI) describes.
- You can configure SQL*Plus to allow INVISIBLE column information to be viewed with the DESCRIBE command. Refer to *SQL*Plus User's Guide and Reference* for more information.

Notes on VISIBLE and INVISIBLE Columns

The following notes apply to VISIBLE and INVISIBLE columns:

- An INVISIBLE column can be used as a partitioning key when specified as part of CREATE TABLE.
- You can specify INVISIBLE columns in a *column_expression*.
- A virtual (expression) column can be an INVISIBLE column.
- PL/SQL %ROWTYPE attributes do not show INVISIBLE columns.
- The COLUMN_ID column of the ALL_, DBA_, and USER_TAB_COLUMNS data dictionary views determines the order in which a SELECT * query returns columns for a table, view, or materialized view. The value of COLUMN_ID is NULL for INVISIBLE columns. When you make an invisible column visible, it will be assigned the next highest available COLUMN_ID value. When you make a visible column invisible, its COLUMN_ID value is set to NULL and COLUMN_ID is decremented by 1 for any columns with a higher COLUMN_ID.

Restrictions on VISIBLE and INVISIBLE Columns

The following restrictions apply to VISIBLE and INVISIBLE columns:

- INVISIBLE columns are not supported in external tables, cluster tables, or temporary tables.
- You cannot make a system-generated hidden column visible.

Note:

To determine whether a column is a system-generated hidden column, query the HIDDEN_COLUMN and USER_GENERATED columns of the ALL_, DBA_, and USER_TAB_COLS data dictionary views. Refer to *Oracle Database Reference* for more information.

DEFAULT

The DEFAULT clause lets you specify a value to be assigned to the column if a subsequent INSERT statement omits a value for the column. The data type of the expression must match the data type specified for the column. The column must also be large enough to hold this expression.

The DEFAULT expression can include any SQL function as long as the function does not return a literal argument, a column reference, or a nested function invocation.

The DEFAULT expression can include the sequence pseudocolumns CURRVAL and NEXTVAL, as long as the sequence exists and you have the privileges necessary to access it. Users who perform subsequent inserts that use the DEFAULT expression must have the INSERT privilege on the table and the SELECT privilege on the sequence. If the sequence is later dropped, then subsequent INSERT statements where the DEFAULT expression is used will result in an error. If you do not fully qualify the sequence by specifying the sequence owner, for example, SCOTT.SEQ1, then Oracle Database will default the sequence owner to be the user who issues the CREATE TABLE statement. For example, if user MARY creates SCOTT.TABLE and refers to a sequence that is not fully qualified, such as SEQ2, then the column will use sequence MARY.SEQ2. Synonyms on sequences undergo a full name resolution and are stored as the fully qualified sequence in the data dictionary; this is true for public and private synonyms. For example, if user BETH adds a column referring to public or private synonym SYN1 and the synonym refers to PETER.SEQ7, then the column will store PETER.SEQ7 as the default.

Restrictions on Default Column Values

Default column values are subject to the following restrictions:

- A DEFAULT expression cannot contain references to PL/SQL functions or to other columns, the pseudocolumns LEVEL, PRIOR, and ROWNUM, or date constants that are not fully specified.
- The expression can be of any form except a scalar subquery expression.

See Also:

"About SQL Expressions " for the syntax of *expr*

ON NULL

If you specify the ON NULL clause, then Oracle Database assigns the DEFAULT column value when a subsequent INSERT statement attempts to assign a value that evaluates to NULL.

When you specify ON NULL, the NOT NULL constraint and NOT DEFERRABLE constraint state are implicitly specified. If you specify an inline constraint that conflicts with NOT NULL and NOT DEFERRABLE, then an error is raised.

If you specify DEFAULT ON NULL FOR INSERT AND UPDATE, the DEFAULT ON NULL semantics applies for INSERT, including the insert branch of merge and multi-table insert and UPDATE, including the update branch of merge.

If you specify DEFAULT ON NULL FOR INSERT ONLY, it is equivalent to DEFAULT ON NULL. It means that DEFAULT ON NULL semantics will apply for INSERT including the insert branch of merge and multi-table insert only.

In before-row DML triggers, *:new.column-name* shows the defaulted value, and you can override the default value in the trigger. If a column is defined as DEFAULT ON NULL FOR INSERT AND UPDATE and the trigger updates the value to NULL, then DEFAULT ON NULL semantics will not apply – i.e. NULL will not be converted to the column default value.

In the following trigger, column c2 has DEFAULT ON NULL semantics. When the trigger is executed, an error is raised since it sets c2 to NULL:

```
create or replace trigger t1_t
before insert or update on t1 for each row
begin
  :new.c2 := NULL;
end;
```

Restriction on the ON NULL Clause

You cannot specify this clause for an object type column or a REF column.

See Also:

"Creating a Table with a DEFAULT ON NULL Column Value: Example"

annotations_clause

The `annotation_name` is an identifier that can have up to 4000 characters. If the annotation name is a reserved word it must be provided in double quotes. When a double quoted identifier is used, the identifier can also contain whitespace characters. However, identifiers that contain only whitespace characters are not accepted.

For examples see Add Annotations at Table Creation: Example

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

identity_clause

Use this clause to specify an identity column. The identity column will be assigned an increasing or decreasing integer value from a sequence generator for each subsequent `INSERT` statement. You can use the `identity_options` clause to configure the sequence generator.

To create an identity column in a schema other than your own, you must have the `CREATE ANY TABLE`, `CREATE ANY SEQUENCE`, and `SELECT ANY SEQUENCE` system privileges.

ALWAYS

If you specify `ALWAYS`, then Oracle Database always uses the sequence generator to assign a value to the column. If you attempt to explicitly assign a value to the column using `INSERT` or `UPDATE`, then an error will be returned. This is the default.

BY DEFAULT

If you specify `BY DEFAULT`, then Oracle Database uses the sequence generator to assign a value to the column by default, but you can also explicitly assign a specified value to the column. If you specify `ON NULL`, then Oracle Database uses the sequence generator to assign a value to the column when a subsequent `INSERT` statement attempts to assign a value that evaluates to NULL. See `column_definition` ON NULL for full semantics.

identity_options

Use the `identity_options` clause to configure the sequence generator. The `identity_options` clause has the same parameters as the `CREATE SEQUENCE` statement. Refer to CREATE SEQUENCE for a full description of these parameters and characteristics. The exception is `START WITH LIMIT VALUE`, which is specific to `identity_options` and can only be used with `ALTER TABLE MODIFY`. Refer to identity_options for more information.

Note:

When you create an identity column, Oracle recommends that you specify the `CACHE` clause with a value higher than the default of 20 to enhance performance.

Restrictions on Identity Columns

Identity columns are subject to the following restrictions:

- You can specify only one identity column per table.
- If you specify `identity_clause`, then you must specify a numeric data type for `datatype` in the `column_definition` clause. You cannot specify a user-defined data type.

- If you specify `identity_clause`, then you cannot specify the `DEFAULT` clause in the `column_definition` clause.
- When you specify `identity_clause`, the `NOT NULL` constraint and `NOT DEFERRABLE` constraint state are implicitly specified. If you specify an inline constraint that conflicts with `NOT NULL` and `NOT DEFERRABLE`, then an error is raised.
- If an identity column is encrypted, then the encryption algorithm may be inferred. Oracle recommends that you use a strong encryption algorithm on identity columns.
- `CREATE TABLE AS SELECT` will not inherit the identity property on a column.

See Also:

"Creating a Table with an Identity Column: Examples"

encryption_spec

Starting with Oracle Database 23ai, the Transparent Data Encryption (TDE) decryption libraries for the GOST and SEED algorithms are deprecated, and encryption to GOST and SEED are desupported.

GOST 28147-89 has been deprecated by the Russian government, and SEED has been deprecated by the South Korean government. If you need South Korean government-approved TDE cryptography, then use ARIA instead. If you are using GOST 28147-89, then you must decrypt and encrypt with another supported TDE algorithm. The decryption algorithms for GOST 28147-89 and SEED are included in Oracle Database 23ai, but are deprecated, and the GOST encryption algorithm is desupported with Oracle Database 23ai. If you are using GOST or SEED for TDE encryption, then Oracle recommends that you decrypt and encrypt with another algorithm before upgrading to Oracle Database 23ai. However, with the exception of the HP Itanium platform, the GOST and SEED decryption libraries are available with Oracle Database 23ai, so you can also decrypt after upgrading.

The `ENCRYPT` clause lets you use the Transparent Data Encryption (TDE) feature to encrypt the column you are defining. You can encrypt columns of type `CHAR`, `NCHAR`, `VARCHAR2`, `NVARCHAR2`, `NUMBER`, `DATE`, LOB, and `RAW`. The data does not appear in its encrypted form to authorized users, such as the user who encrypts the column.

Note:

Column encryption requires that a system administrator with appropriate privileges has initialized the security module, opened a keystore, and set an encryption key. Refer to *Transparent Data Encryption* for general information about column encryption and to *security_clauses* for related `ALTER SYSTEM` statements.

USING '*encrypt_algorithm*'

Use this clause to specify the name of the algorithm to be used. Valid algorithms are `AES256`, `AES192`, `AES128` and `3DES168`. If the `COMPATIBLE` initialization parameter is set to `12.2` or higher, then the following algorithms are also valid: `ARIA128`, `ARIA192`, `ARIA256`, `GOST256`, and `SEED128`. If you omit this clause, then the database uses `AES192`. If you encrypt more than one column in the same table, and if you specify the `USING` clause for one of the columns, then you must specify the same encryption algorithm for all the encrypted columns.

IDENTIFIED BY *password*

If you specify this clause, then the database derives the column key from the specified password.

'*integrity_algorithm*'

Use this clause to specify the integrity algorithm to be used. Valid integrity algorithms are `SHA-1` and `NOMAC`.

- If you specify `SHA-1`, then TDE uses the Secure Hash Algorithm (SHA-1) and adds a 20-byte Message Authentication Code (MAC) to each encrypted value for integrity checking. This is the default.
- If you specify `NOMAC`, then TDE does not add a MAC and does not perform the integrity check. This saves 20 bytes of disk space per encrypted value. Refer to *Transparent Data Encryption* for more information on using `NOMAC` to save disk space and improve performance.

All encrypted columns in a table must use the same integrity algorithm. If you already have a table column using the `SHA-1` algorithm, then you cannot use the `NOMAC` parameter to encrypt another column in the same table. Refer to the REKEY *encryption_spec* clause of `ALTER TABLE` to learn how to change the integrity algorithm used by all encrypted columns in a table.

SALT | NO SALT

Specify `SALT` to instruct the database to append a random string, called "salt," to the clear text of the column before encrypting it. This is the default.

Specify `NO SALT` to prevent the database from appending salt to the clear text of the column before encrypting it.

The following considerations apply when specifying `SALT` or `NO SALT` for encrypted columns:

- If you want to use the column as an index key, then you must specify `NO SALT`. Refer to *Transparent Data Encryption* for a description of "salt" in this context.
- If you specify table compression for the table, then the database does not compress the data in encrypted columns with `SALT`.

You cannot specify `SALT` or `NO SALT` for LOB encryption.

Restrictions on *encryption_spec*

The following restrictions apply to column encryption:

- Transparent Data Encryption is not supported by the traditional import and export utilities or by transportable-tablespace-based export. Use the Data Pump import and export utilities with encrypted columns instead.
- To encrypt a column in an external table, the table must use `ORACLE_DATAPUMP` as its access type.
- You cannot encrypt a column in tables owned by `SYS`.
- You cannot encrypt a foreign key column.

See Also:

Transparent Data Encryption for more information about Transparent Data Encryption

virtual_column_definition

Use *virtual_column_definition* to create a virtual column, also known as an expression column. Expression columns can be virtual, meaning not stored on disk, or materialized, meaning stored on disk. Depending on whether such a column is virtual or materialized, the database derives the values on demand at access time (virtual) or by computing the values and storing them on disk at DML time (materialized). Such columns can be used in queries, DML, and DDL statements. They can be indexed, and you can collect statistics on them. Thus, they can be treated just like other columns. Exceptions and restrictions are listed below in "Notes on Virtual (Expression) Columns" and "Restrictions on Virtual (Expression) Columns".

column

For *column*, specify the name of the virtual (expression) column.

datatype

You can optionally specify the data type of the virtual (expression) column. If you omit *datatype*, then the database determines the data type of the column based on the data type of the underlying expressions. All Oracle scalar data types and XMLType are supported.

COLLATE

The COLLATE clause lets you specify a data-bound collation for the virtual (expression) column. For *column_collation_name*, specify a valid named collation or pseudo-collation. If you omit this clause, then the column is assigned the default collation for the table as it stands at the time the column is created, unless the column belongs to a foreign key, in which case it inherits the collation from the corresponding column of the parent key. Refer to the DEFAULT COLLATION clause for more information on the default collation for a table.

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.

VISIBLE | INVISIBLE

Use this clause to specify whether the virtual (expression) column is VISIBLE or INVISIBLE. The default is VISIBLE. For complete information, refer to "VISIBLE | INVISIBLE".

GENERATED ALWAYS

The optional keywords GENERATED ALWAYS are provided for semantic clarity.

column_expression

The AS *column_expression* clause determines the content of the column. Refer to "Column Expressions " for more information on *column_expression*.

VIRTUAL

Use VIRTUAL to create an expression column that is virtual.

MATERIALIZED

Use MATERIALIZED to store the computed value of an expression column on disk. Rather than computing the values at access time, the computation will take place at DML time.

An alternate way to specify a materialized expression column is to use the keyword STORED instead of MATERIALIZED.

evaluation_edition_clause

You must specify this clause if *column_expression* refers to an editioned PL/SQL function. Use this clause to specify the edition that is searched during name resolution of the editioned PL/SQL function—the evaluation edition.

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

If you omit the *evaluation_edition_clause*, then editioned objects are invisible during name resolution and an error will result. If the evaluation edition is dropped, then a subsequent query on the virtual (expression) column will result in an error.

The database does not maintain dependencies on the functions referenced by a virtual (expression) column. Therefore, if a virtual (expression) column refers to a noneditioned function, and the function becomes editioned, then the following operations may raise an error:

- Querying the virtual (expression) column
- Updating a row that includes the virtual (expression) column
- Firing a trigger that accesses the virtual (expression) column

See Also:

Oracle Database Development Guide for more information on specifying the evaluation edition for a virtual (expression) column

unusable_editions_clause

This clause lets you specify that the virtual (expression) column expression is unusable for evaluating queries in one or more editions. The remaining editions form a range of editions in which it is safe for the optimizer to use the virtual (expression) column expression to evaluate queries.

For example, suppose you define a function-based index on the virtual (expression) column. The optimizer can use the function-based index to evaluate queries that contain the virtual (expression) column expression in their `WHERE` clause. If a query is compiled in an edition that is in the usable range of editions for the virtual (expression) column, then the optimizer will consider using the index to evaluate the query. If a query is compiled in an edition outside the usable range of editions for the virtual (expression) column, then the optimizer will not consider using the index.

See Also:

Oracle Database Concepts for more information on optimization with function-based indexes

UNUSABLE BEFORE Clause

This clause lets you specify that the virtual (expression) column expression is unusable for evaluating queries in the ancestors of an edition.

- If you specify `CURRENT EDITION`, then the virtual (expression) column expression is unusable in the ancestors of the edition in which this DDL statement is executed.
- If you specify `EDITION` *`edition`*, then the virtual (expression) column expression is unusable in the ancestors of the specified *`edition`*.

UNUSABLE BEGINNING WITH Clause

This clause lets you specify that the virtual (expression) column expression is unusable for evaluating queries in an edition and its descendants.

- If you specify `CURRENT EDITION`, then the virtual (expression) column expression is unusable in the edition in which this DDL statement is executed and its descendants.
- If you specify `EDITION` *`edition`*, then the virtual (expression) column expression is unusable in the specified *`edition`* and its descendants.
- Specifying `NULL EDITION` is equivalent to omitting the `UNUSABLE BEGINNING WITH` clause.

If an edition specified in this clause is subsequently dropped, there is no effect on the virtual (expression) column.

Notes on Virtual (Expression) Columns

- If *`column_expression`* refers to a column on which column-level security is implemented, then the virtual (expression) column does not inherit the security rules of the base column. In such a case, you must ensure that data in the virtual (expression) column is protected, either by duplicating a column-level security policy on the virtual (expression) column or by applying a function that implicitly masks the data. For example, it is common for credit card numbers to be protected by a column-level security policy, while still allowing call center employees to view the last four digits of the credit card number for validation purposes. In such a case, you could define the virtual (expression) column to take a substring of the last four digits of the credit card number.
- A table index defined on a virtual (expression) column is equivalent to a function-based index on the table.
- You cannot directly update a virtual (expression) column. Thus, you cannot specify a virtual (expression) column in the `SET` clause of an `UPDATE` statement. However, you can specify a virtual (expression) column in the `WHERE` clause of an `UPDATE` statement. Likewise, you can specify a virtual (expression) column in the `WHERE` clause of a `DELETE` statement to delete rows from a table based on the derived value of the virtual (expression) column.
- A query that specifies in its `FROM` clause a table containing a virtual (expression) column is eligible for result caching. Refer to "RESULT_CACHE Hint " for more information on result caching.
- The *`column_expression`* can refer to a PL/SQL function if the function is explicitly designated `DETERMINISTIC` during its creation. However, if the function is subsequently replaced, definitions dependent on the virtual (expression) column are not invalidated. In such a case, if the table contains data, queries that reference the virtual (expression) column may return incorrect results if the virtual (expression) column is used in the definition of constraints, indexes, or materialized views or for result caching. Therefore, in order to replace the deterministic PL/SQL function for a virtual (expression) column.
  - Disable and re-enable any constraints on the virtual (expression) column.
  - Rebuild any indexes on the virtual (expression)column.
  - Fully refresh materialized views accessing the virtual (expression) column.
  - Flush the result cache if cached queries have accessed the virtual (expression) column.

  - Regather statistics on the table.
- A virtual (expression) column can be an `INVISIBLE` column. The *`column_expression`* can contain `INVISIBLE` columns.

Restrictions on Virtual (Expression) Columns

- You can create virtual (expression) columns only in relational heap tables. virtual (expression) columns are not supported for index-organized, external, object, cluster, or temporary tables.
- The *`column_expression`* in the `AS` clause has the following restrictions:
  - It cannot refer to another virtual (expression) column by name.
  - Any columns referenced in *`column_expression`* must be defined on the same table.
  - It can refer to a deterministic user-defined function, but if it does, then you cannot use the virtual (expression) column as a partitioning key column.
  - The output of *`column_expression`* must be a scalar value.

> **See Also:**
>
> "Column Expressions " for additional information and restrictions on *`column_expression`*

- The virtual (expression) column cannot be an Oracle supplied data type, a user-defined type, or LOB or `LONG RAW`.
- You cannot specify a call to a PL/SQL function in the defining expression for a virtual (expression) column that you want to use as a partitioning column.

> **See Also:**
>
> "Adding a Virtual Table Column: Example" and *Oracle Database Administrator's Guide* for examples of creating tables with virtual (expression) columns

period_definition

Use the *`period_definition`* clause to create a valid time dimension for *`table`*.

This clause implements Temporal Validity support for *`table`*. If you specify this clause, then one column in *`table`*, the start time column, contains a start date or timestamp, and another column in *`table`*, the end time column, contains an end date or timestamp. These two columns define a valid time dimension for *`table`*—that is, a period of time for which each row is considered valid. You can use Oracle Flashback Query to retrieve rows from *`table`* based on whether they are considered valid as of a specified time, before a specified time, or during a specified time period.

You can specify at most one valid time dimension when you create a table. You can subsequently add additional valid time dimensions to a table with the *add_period_clause* of `ALTER TABLE`.

valid_time_column

Specify the name of the valid time dimension. The name must satisfy the requirements listed in "Database Object Naming Rules ". Oracle Database creates an `INVISIBLE` virtual (expression) column with this name of data type `NUMBER` in *`table`*.

start_time_column **and** ***end_time_column***

You can optionally specify these clauses as follows:

- Use *`start_time_column`* to specify the name of the start time column, which contains the start date or timestamp.
- Use *`end_time_column`* to specify the name of the end time column, which contains the end date or timestamp.

The names you specify for *`start_time_column`* and *`end_time_column`* must satisfy the requirements listed in "Database Object Naming Rules ".

If you specify these clauses, then you must define *`start_time_column`* and *`end_time_column`* in the *`column_definition`* clause of `CREATE TABLE`. Each column must be of a datetime data type (`DATE`, `TIMESTAMP`, `TIMESTAMP WITH TIME ZONE`, or `TIMESTAMP WITH LOCAL TIME ZONE`) and can be `VISIBLE` or `INVISIBLE`.

If you do not specify these clauses, then Oracle Database creates a start time column named *`valid_time_column`*`_START`, and an end time column named *`valid_time_column`*`_END`. These columns are of data type `TIMESTAMP WITH TIME ZONE` and are `INVISIBLE`.

You can insert and update values in the start time column and end time column as you would any column, with the following considerations:

- If the value of the start time column is NULL, then the row is considered valid for all time values that occur before, but not including, the value of the end time column.
- If the value of the end time column is NULL, then the row is considered valid for all time values that occur on or after the value of the start time column.
- If the value of neither column is NULL, then the value of the start time column must be earlier than the value of the end time column. The row is considered valid for all time values that occur on or after the value of the start time column, and up to, but not including, the value of the end time column.
- If the value of both columns is NULL, then the row is considered valid for all time values.

Restrictions on Valid Time Dimension Columns

The following restrictions apply to valid time dimension columns:

- The *`valid_time_column`* is for internal use only. You cannot perform DDL or DML operations on it with one exception: You can drop the column by using the *`drop_period_clause`* of `ALTER TABLE`.
- You can drop the start time column and end time column only by using the *`drop_period_clause`* of `ALTER TABLE`.
- If the start time column and end time column are automatically created by Oracle Database, then they are `INVISIBLE` and you cannot subsequently make them `VISIBLE`.

See Also:

- *Oracle Database Development Guide* for more information on Temporal Validity
- `SELECT` *flashback_query_clause* for more information on Oracle Flashback Query
- `ALTER TABLE` *add_period_clause* and *drop_period_clause* for information how to add and drop a valid time dimension

Constraint Clauses

Use these clauses to create constraints on the table columns. You must specify a `PRIMARY KEY` constraint for an index-organized table, and it cannot be `DEFERRABLE`. Refer to *constraint* for syntax and description of these constraints as well as examples.

inline_ref_constraint* and *out_of_line_ref_constraint

These clauses let you describe a column of type `REF`. The only difference between these clauses is that you specify *`out_of_line_ref_constraint`* from the table level, so you must identify the `REF` column or attribute you are defining. Specify *`inline_ref_constraint`* as part of the definition of the `REF` column or attribute.

See Also:

"REF Constraint Examples"

inline_constraint

Use the *`inline_constraint`* to define an integrity constraint as part of the column definition.

You can create `UNIQUE`, `PRIMARY KEY`, and `REFERENCES` constraints on scalar attributes of object type columns. You can also create `NOT NULL` constraints on object type columns and `CHECK` constraints that reference object type columns or any attribute of an object type column.

out_of_line_constraint

Use the *`out_of_line_constraint`* syntax to define an integrity constraint as part of the table definition.

supplemental_logging_props

The *`supplemental_logging_props`* clause lets you instruct the database to put additional data into the log stream to support log-based tools.

supplemental_log_grp_clause

Use this clause to create a named log group.

- The `NO LOG` clause lets you omit from the redo log one or more columns that would otherwise be included in the redo for the named log group. You must specify at least one fixed-length column without `NO LOG` in the named log group.
- If you specify `ALWAYS`, then during an update, the database includes in the redo all columns in the log group. This is called an **unconditional log group** (sometimes called an "always log group"), because Oracle Database supplementally logs all the columns in the log group

when the associated row is modified. If you omit `ALWAYS`, then the database supplementally logs all the columns in the log group only if any column in the log group is modified. This is called a **conditional log group**.

You can query the appropriate `USER_`, `ALL_`, or `DBA_LOG_GROUP_COLUMNS` data dictionary view to determine whether any supplemental logging has already been specified.

supplemental_id_key_clause

Use this clause to specify that all or a combination of the primary key, unique key, and foreign key columns should be supplementally logged. Oracle Database will generate either an **unconditional log group** or a **conditional log group**. With an unconditional log group, the database supplementally logs all the columns in the log group when the associated row is modified. With a conditional log group, the database supplementally logs all the columns in the log group only if any column in the log group is modified.

- If you specify `ALL COLUMNS`, then the database includes in the redo log all the fixed-length maximum size columns of that row. Such a redo log is a system-generated unconditional log group.
- If you specify `PRIMARY KEY COLUMNS`, then for all tables with a primary key, the database places into the redo log all columns of the primary key whenever an update is performed. Oracle Database evaluates which columns to supplementally log as follows:
  - First the database chooses columns of the primary key constraint, if the constraint is validated or marked `RELY` and is not marked as `DISABLED` or `INITIALLY DEFERRED`.
  - If no primary key columns exist, then the database looks for the smallest `UNIQUE` index with at least one `NOT NULL` column and uses the columns in that index.
  - If no such index exists, then the database supplementally logs all scalar columns of the table.
- If you specify `UNIQUE COLUMNS`, then for all tables with a unique key or a bitmap index, if any of the unique key or bitmap index columns are modified, the database places into the redo log all other columns belonging to the unique key or bitmap index. Such a log group is a system-generated conditional log group.
- If you specify `FOREIGN KEY COLUMNS`, then for all tables with a foreign key, if any foreign key columns are modified, the database places into the redo log all other columns belonging to the foreign key. Such a redo log is a system-generated conditional log group.

If you specify this clause multiple times, then the database creates a separate log group for each specification. You can query the appropriate `USER_`, `ALL_`, or `DBA_LOG_GROUPS` data dictionary view to determine whether any supplemental logging data has already been specified.

immutable_table_clauses

You must specify this clause when you create an immutable table.

If you do not specify the `VERSION` using *`immutable_data_format_clause`*, a `V1` immutable table is created by default.

Example: Create an Immutable Table

The following example creates an immutable table named *`trade_ledger`* in your user schema. The immutable table can be dropped only after 40 days of inactivity. Rows cannot be deleted until 100 days after they have been inserted.

```
CREATE IMMUTABLE TABLE trade_ledger (tr_id NUMBER, user_name VARCHAR2(40), tr_value
NUMBER)

       NO DROP UNTIL 40 DAYS IDLE

       NO DELETE UNTIL 100 DAYS AFTER INSERT;
```

blockchain_table_clauses

When you create a blockchain table, you must specify the *`blockchain_table_clauses`* :

- *`blockchain_drop_table_clause`*
- *`blockchain_row_retention_clause`*
- *`blockchain_hash_clause`*
- *`blockchain_data_format_clause`*

blockchain_drop_table_clause

```
NO DROP [ UNTIL integer DAYS IDLE ]
```

Use `integer` to specify the number of days that the blockchain table must be idle (i.e. have no rows inserted). The minimum idle retention period is 0 days, but the recommended idle retention period is 16 days.

You can specify this clause in two ways:

- `NO DROP` means that the blockchain table cannot be dropped.
- `NO DROP UNTIL integer DAYS IDLE` means that the blockchain table cannot be dropped, if the newest row is less than `integer` of days old.

blockchain_row_retention_clause

```
NO DELETE [ LOCKED ]
  | NO DELETE UNTIL integer DAYS AFTER INSERT [LOCKED]
```

- `integer` specifies the idle retention period for inserted rows before they can be deleted. The minimum idle retention period for inserted rows is 16 days.
- If you specify `LOCKED`, then you cannot change the retention period using `ALTER TABLE`.
- If you do not specify `LOCKED` in the clause `UNTIL number DAYS AFTER INSERT`, then you can change the retention period using `ALTER TABLE`, but only to a value higher than the previous retention period.
- If you specify `NO DELETE LOCKED`, then you cannot delete any rows from this table. But you can drop the entire table if the table is inactive for more than the number of days specified in the *`blockchain_drop_table_clause`*.

blockchain_hash_clause

```
HASHING USING sha2_512
```

You must specify this clause last after *`blockchain_drop_table_clause`* and *`blockchain_row_retention_clause`* when you create a blockchain table.

You cannot specify this clause to modify a blockchain table using the `ALTER TABLE` statement.

blockchain_row_version_user_chain_clause

WITH ROW VERSION

This clause is optional and can only be specified on `VERSION V2` blockchain tables. You can specify at most three user-defined columns with the clause. The name of the row version sequence identified by *`row_version_name`* is used to verify the user chain. The types of the columns are restricted to `NUMBER`, `CHAR`, `VARCHAR2`, and `RAW`. When the clause is specified, rows with identical values in the specified user-defined columns are sequenced using the Oracle managed hidden column `ORABCTAB_ROW_VERSION$`.

Example: Row Versions

```
CREATE BLOCKCHAIN TABLE bank_ledger (bank VARCHAR2(128), account_no NUMBER, deposit_date
DATE, deposit_amount NUMBER)
  NO DROP UNTIL 31 DAYS IDLE
  NO DELETE LOCKED
  HASHING USING SHA2_512 WITH ROW VERSION ACCOUNT_NO (bank, account_no) VERSION V2;
```

AND USER CHAIN

The optional `AND USER CHAIN` clause can only be specified on `VERSION V2` blockchain tables as part of the `WITH ROW VERSION` clause. It extends the functionality offered by the row versions and links these rows in a separate (user) blockchain. At most three user-defined columns can be specified to define a user chain. The name of the user chain is that specified in the `WITH ROW VERSION` clause. The sequencing of the rows in the user chain is accomplished using column `ORABCTAB_ROW_VERSION$`. The crypto hash is maintained in column `ORABCTAB_USER_CHAIN_HASH$`. Note, that `ORABCTAB_USER_CHAIN_HASH$` is not signed in `V2` blockchain tables. Only the system crypto hash can be signed (with signature stored in `ORABCTAB_SIGNATURE$ or ORABCTAB_DELEGATE_SIGNATURE$` column).

Example: AND USER CHAIN

```
CREATE BLOCKCHAIN TABLE bank_ledger (bank VARCHAR2(128), account_no NUMBER, deposit_date
DATE, deposit_amount NUMBER)
         NO DROP UNTIL 31 DAYS IDLE
         NO DELETE LOCKED
         HASHING USING SHA2_512 WITH ROW VERSION AND USER CHAIN bank_accounts (bank,
account_no) VERSION V2;
```

blockchain_system_chains_clause

Specify this clause to override the default of 32 system chains per instance. The number of system chains configured by this clause must be between 1 and 1024.

blockchain_data_format_clause

You must specify the version when you create a blockchain table, either `V1` or `V2`. Version `V2` creates additional Oracle managed hidden columns than `V1`.

You cannot use `ALTER TABLE` to convert version from `V1` to `V2` and vice versa.

Example: HASHING and VERSION

```
CREATE BLOCKCHAIN TABLE bank_ledger (bank VARCHAR2(128), deposit_date DATE,
deposit_amount NUMBER)
  NO DROP UNTIL 31 DAYS IDLE
  NO DELETE LOCKED
  HASHING USING SHA2_512 VERSION V2;
```

DEFAULT COLLATION

This clause lets you specify the default collation for the table. The default collation is assigned to columns of the table that are of a character data type and are created with this statement or subsequently added to the table with an `ALTER TABLE` statement. For *`collation_name`*, specify a valid named collation or pseudo-collation.

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

You can override the table's default collation and assign a data-bound collation to a particular column by specifying the `COLLATE` clause in the *`column_definition`* or *`virtual_column_definition`* clause of `CREATE TABLE` or `ALTER TABLE`, or the *`modify_col_properties`* or *`modify_virtcol_properties`* clause of `ALTER TABLE`.

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

Restriction on Collation for CLOB and NCLOB Columns

If a column has the data type of `CLOB` or `NCLOB`, then its specified collation must be `USING_NLS_COMP`. The collation of `CLOB` and `NCLOB` columns is always `USING_NLS_COMP` and is not affected by the default collation for the table.

See Also:

Oracle Database Globalization Support Guide for full information on default collations and data-bound collations

ON COMMIT

The `ON COMMIT` clause is relevant only if you are creating a global temporary table. This clause specifies whether the data in the temporary table persists for the duration of a transaction or a session.

DELETE ROWS

Specify `DELETE ROWS` for a transaction-specific temporary table. This is the default. Oracle Database will truncate the table (delete all its rows) after each commit.

PRESERVE ROWS

Specify `PRESERVE ROWS` for a session-specific temporary table. Oracle Database will truncate the table (delete all its rows) when you terminate the session.

You can define the scope of a private temporary table using `ON COMMIT`. Use `DROP DEFINITION` to define a transaction-specific table and `PRESERVE DEFINITION` to define a session-specific table .

DROP DEFINITION

Specify `DROP DEFINITION` to create a private temporary table whose content and definition are dropped when the transaction commits. The creation of a transaction-specific private temporary table does not issue an implicit commit, but can be issued within an ongoing transaction. The scope of this private temporary table is limited to the ongoing transaction. The scope of this private temporary table is limited to the transaction. This is the default.

PRESERVE DEFINITION

Specify `PRESERVE DEFINITION` to create a private temporary table whose definition is preserved when the transaction commits. The creation of a session-specific private temporary table issues an implicit commit. The scope of this private temporary table is extended to the session.

physical_properties

The physical properties relate to the treatment of extents and segments and to the storage characteristics of the table.

INTERNAL | EXTERNAL
Use the keyword `INTERNAL` to indicate an internal partition. This is the default. Use the keyword `EXTERNAL` to indicate an external partition.

deferred_segment_creation

Use this clause to determine when the database should create the segment(s) for this table:

- `SEGMENT CREATION DEFERRED`: This clause defers creation of the table segment — as well as segments for any LOB columns of the table, any indexes created implicitly as part of table creation, and any indexes subsequently explicitly created on the table — until the first row of data is inserted into the table. At that time, the segments for the table, LOB columns and indexes, and explicitly created indexes are all materialized and inherit any storage properties specified in this `CREATE TABLE` statement or, in the case of explicitly created indexes, the `CREATE INDEX` statement. These segments are created regardless whether the initial insert operation is uncommitted or rolled back. This is the default value.

  Caution:

  When creating many tables with deferred segment creation, ensure that you allocate enough space for your database so that when the first rows are inserted, there is enough space for all the new segments.

- `SEGMENT CREATION IMMEDIATE`: The table segment is created as part of this `CREATE TABLE` statement.

Immediate segment creation is useful, for example, if your application depends upon the object appearing in the `DBA_`, `USER_`, and `ALL_SEGMENTS` data dictionary views, because the object will not appear in those views until the segment is created. This clause overrides the setting of the `DEFERRED_SEGMENT_CREATION` initialization parameter.

To determine whether a segment has been created for an existing table or its LOB columns or indexes, query the `SEGMENT_CREATED` column of `USER_TABLES`, `USER_INDEXES`, or `USER_LOBS`.

Notes on Tables Without Segments

The following rules apply to a table whose segment has not yet been materialized:

- If you create this table with `CREATE TABLE` ... `AS` *`subquery`*, then if the source table has no rows, segment creation of the new table is deferred. If the source table has rows, then segment creation of the new table is not deferred.

- If you specify `ALTER TABLE` ... `ALLOCATE EXTENT` before the segment is materialized, then the segment is materialized and then an extent is allocated. However the `ALLOCATE EXTENT` clause in a DDL statement on any indexes of the table will return an error.
- In a DDL statement on the table or its LOB columns or indexes, any specification of `DEALLOCATE UNUSED` is silently ignored.
- `ONLINE` operations on indexes of a table or table partition without a segment will silently be disabled; that is, they will proceed `OFFLINE`.
- If any of the following DDL statements are executed on a table with one or more LOB columns, then the resulting partition(s) or subpartition(s) will be materialized:
  - `ALTER TABLE SPLIT [SUB]PARTITION`
  - `ALTER TABLE MERGE [SUB]PARTITIONS`
  - `ALTER TABLE ADD [SUB]PARTITION` (hash partitions only)
  - `ALTER TABLE COALESCE [SUB]PARTITION` (hash partitions only)

Restrictions on Deferred Segment Creation

This clause is subject to the following restrictions:

- You cannot defer segment creation for the following types of tables: clustered tables, global temporary tables, session-specific temporary tables, internal tables, external tables, and tables owned by `SYS`, `SYSTEM`, `PUBLIC`, `OUTLN`, or `XDB`.
- Deferred segment creation is not supported in dictionary-managed tablespaces.
- Deferred segment creation is not supported in the `SYSTEM` tablespace.
- Serializable transactions do not work with deferred segment creation. Trying to insert data into an empty table with no segment created causes an error.

See Also:

Oracle Database Concepts for general information on segment allocation and *Oracle Database Reference* for more information about the `DEFERRED_SEGMENT_CREATION` initialization parameter

segment_attributes_clause

The *`segment_attributes_clause`* lets you specify physical attributes and tablespace storage for the table.

physical_attributes_clause

The *`physical_attributes_clause`* lets you specify the value of the `PCTFREE`, `PCTUSED`, and `INITRANS` parameters and the storage characteristics of the table.

- For a nonpartitioned table, each parameter and storage characteristic you specify determines the actual physical attribute of the segment associated with the table.
- For partitioned tables, the value you specify for the parameter or storage characteristic is the default physical attribute of the segments associated with all partitions specified in this `CREATE` statement (and in subsequent `ALTER TABLE` ... `ADD PARTITION` statements), unless you explicitly override that value in the `PARTITION` clause of the statement that creates the partition.

If you omit this clause, then Oracle Database sets PCTFREE to 10, PCTUSED to 40, and INITRANS to 1.

See Also:

- *physical_attributes_clause* and storage_clause for a description of these clauses
- "Creating a Table: Storage Example"

TABLESPACE

Specify the tablespace in which Oracle Database creates the table, object table OIDINDEX, partition, LOB data segment, LOB index segment, or index-organized table overflow data segment. If you omit TABLESPACE, then the database creates that item in the default tablespace of the owner of the schema containing the table.

For a heap-organized table with one or more LOB columns, if you omit the TABLESPACE clause for LOB storage, then the database creates the LOB data and index segments in the tablespace where the table is created.

For an index-organized table with one or more LOB columns, if you omit TABLESPACE, then the LOB data and index segments are created in the tablespace in which the primary key index segment of the index-organized table is created.

For nonpartitioned tables, the value specified for TABLESPACE is the actual physical attribute of the segment associated with the table. For partitioned tables, the value specified for TABLESPACE is the default physical attribute of the segments associated with all partitions specified in the CREATE statement and on subsequent ALTER TABLE ... ADD PARTITION statements, unless you specify TABLESPACE in the PARTITION description.

See Also:

CREATE TABLESPACE for more information on tablespaces

TABLESPACE SET

This clause is valid only when creating a sharded table by specifying the SHARDED keyword of CREATE TABLE. Use this clause to specify the tablespace set in which Oracle Database creates the table.

You can only associate a tablespace set with one table family when you use the CREATE SHARDED TABLE statement. If you try to use a tablespace set with more than one table family, an error will be thrown .

logging_clause

Specify whether the creation of the table and of any indexes required because of constraints, partition, or LOB storage characteristics will be logged in the redo log file (LOGGING) or not (NOLOGGING).The logging attribute of the table is independent of that of its indexes.

This attribute also specifies whether subsequent direct loader (SQL*Loader) and direct-path INSERT operations against the table, partition, or LOB storage are logged (LOGGING) or not logged (NOLOGGING).

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

table_compression

The *table_compression* clause is valid only for heap-organized tables. Use this clause to instruct the database whether to compress data segments to reduce disk use. The COMPRESS clauses enable table compression. The NOCOMPRESS clause disables table compression. The default is NOCOMPRESS.

COMPRESS

Specifying only the keyword COMPRESS is equivalent to specifying ROW STORE COMPRESS BASIC and enables basic table compression.

ROW STORE COMPRESS BASIC

When you enable table compression by specifying either ROW STORE COMPRESS or ROW STORE COMPRESS BASIC, you enable **basic table compression**. Oracle Database attempts to compress data during direct-path INSERT operations when it is productive to do so. The original import utility (imp) does not support direct-path INSERT, and therefore cannot import data in a compressed format.

Tables with basic table compression use a PCTFREE value of 0 to maximize compression, unless you explicitly set a value for PCTFREE in the *physical_attributes_clause*.

In earlier releases, basic table compression was enabled using COMPRESS BASIC. This syntax is still supported for backward compatibility.

See Also:

"Conventional and Direct-Path INSERT" for information on direct-path INSERT operations, including restrictions

ROW STORE COMPRESS ADVANCED

When you enable table compression by specifying ROW STORE COMPRESS ADVANCED, you enable **Advanced Row Compression**. Oracle Database compresses data during all DML operations on the table. This form of compression is recommended for OLTP environments.

Tables with ROW STORE COMPRESS ADVANCED or NOCOMPRESS use the PCTFREE default value of 10, to maximize compress while still allowing for some future DML changes to the data, unless you override this default explicitly.

In earlier releases, Advanced Row Compression was called OLTP table compression and was enabled using COMPRESS FOR OLTP. This syntax is still supported for backward compatibility.

COLUMN STORE COMPRESS FOR { QUERY | ARCHIVE }

When you specify COLUMN STORE COMPRESS FOR QUERY or COLUMN STORE COMPRESS FOR ARCHIVE, you enable **Hybrid Columnar Compression**. With Hybrid Columnar Compression, data can be compressed during direct-path inserts, conventional inserts, and array inserts. During the load process, data is transformed into a column-oriented format and then compressed. Oracle Database uses a compression algorithm appropriate for the level you specify. In general, the higher the level, the greater the compression ratio. Hybrid Columnar Compression can result in higher compression ratios, at a greater CPU cost. Therefore, this form of compression is recommended for data that is not frequently updated.

COLUMN STORE COMPRESS FOR QUERY is useful in data warehousing environments. Valid values are LOW and HIGH, with HIGH providing a higher compression ratio. The default is HIGH.

COLUMN STORE COMPRESS FOR ARCHIVE uses higher compression ratios than COLUMN STORE COMPRESS FOR QUERY, and is useful for compressing data that will be stored for long periods of time. Valid values are LOW and HIGH, with HIGH providing the highest possible compression ratio. The default is LOW.

Specifying COLUMN STORE COMPRESS is equivalent to specifying COLUMN STORE COMPRESS FOR QUERY HIGH.

Tables with COLUMN STORE COMPRESS FOR QUERY or COLUMN STORE COMPRESS FOR ARCHIVE use a PCTFREE value of 0 to maximize compression, unless you explicitly set a value for PCTFREE in the *physical_attributes_clause*. For these tables, PCTFREE has no effect for blocks loaded using direct-path INSERT. PCTFREE is honored for blocks loaded using conventional INSERT, and for blocks created as a result of DML operations on blocks originally loaded using direct-path INSERT.

[NO] ROW LEVEL LOCKING

If you specify ROW LEVEL LOCKING, then Oracle Database uses row-level locking during DML operations. This improves the performance of these operations when accessing Hybrid Columnar Compressed data. If you specify NO ROW LEVEL LOCKING, then row-level locking is not used. The default is NO ROW LEVEL LOCKING.

In earlier releases, Hybrid Columnar Compression was enabled using COMPRESS FOR QUERY and COMPRESS FOR ARCHIVE. This syntax is still supported for backward compatibility.

See Also:

Oracle Database Concepts for more information on Hybrid Columnar Compression, which is a feature of certain Oracle storage systems

Notes on Table Compression

You can specify table compression for the following portions of a heap-organized table:

- For an entire table, in the *physical_properties* clause of *relational_table* or *object_table*
- For a range partition, in the *table_partition_description* of the *range_partitions* clause
- For a composite range partition, in the *table_partition_description* of the *range_partition_desc* clause
- For a composite list partition, in the *table_partition_description* of the *list_partition_desc* clause
- For a list partition, in the *table_partition_description* of the *list_partitions* clause
- For a system or reference partition, in the *table_partition_description* of the *reference_partition_desc* clause
- For the storage table of a nested table, in the *nested_table_col_properties* clause

See Also:

Oracle Database PL/SQL Packages and Types Reference for information about the `DBMS_COMPRESSION` package, which helps you choose the correct compression level for an application, and *Oracle Database Administrator's Guide* for more information about table compression, including examples

Restrictions on Table Compression

Table compression is subject to the following restrictions:

- Data segments of BasicFiles LOBs are not compressed. For information on compression of SecureFiles LOBs, see LOB_compression_clause.
- You cannot drop a column from a table that uses `COMPRESS BASIC`, although you can set such a column as unused. All of the operations of the `ALTER TABLE` ... *`drop_column_clause`* are valid for tables that use `ROW STORE COMPRESS ADVANCED`, `COLUMN STORE COMPRESS FOR QUERY`, and `COLUMN STORE COMPRESS FOR ARCHIVE`.
- You cannot specify any type of table compression for an index-organized table, any overflow segment or partition of an overflow segment, or any mapping table segment of an index-organized table.
- You cannot specify any type of table compression for external tables or for tables that are part of a cluster.
- You cannot specify any type of table compression for tables with `LONG` or `LONG RAW` columns, tables that are owned by the `SYS` schema and reside in the `SYSTEM` tablespace, or tables with `ROWDEPENDENCIES` enabled.
- You cannot specify Hybrid Columnar Compression on tables that are enabled for flashback archiving.
- You cannot specify Hybrid Columnar Compression on the following object-relational features: object tables, `XMLType` tables, columns with abstract data types, collections stored as tables, or OPAQUE types, including `XMLType` columns stored as objects.
- When you update a row in a table compressed with Hybrid Columnar Compression, the `ROWID` of the row may change.
- In tables compressed with Hybrid Columnar Compression, updates to a single row may result in locks on multiple rows. Concurrency for write transactions may therefore be affected.
- If a table compressed with Hybrid Columnar Compression has a foreign key constraint, and you insert data using `INSERT` with the `APPEND` hint, then the data will be compressed to a lesser level than is typical with Hybrid Columnar Compression. To compress the data with Hybrid Columnar Compression, disable the foreign key constraint, insert the data using `INSERT` with the `APPEND` hint, and then reenable the foreign key constraint.

inmemory_table_clause

Use this clause to enable or disable the table for the In-Memory Column Store (IM column store). The IM column store is an optional, static SGA pool that stores copies of tables and partitions in a special columnar format optimized for rapid scans. The IM column store does not replace the buffer cache, but acts as a supplement so that both memory areas can store the same data in different formats.

- Specify `INMEMORY` to enable the table for the IM column store.

You can optionally use the *inmemory_attributes* clause to specify how table data is stored in the IM column store. This clause enables you to specify the data compression method and the data population priority. In an Oracle RAC environment, it also enables you to specify how the data is distributed and duplicated across Oracle RAC instances. Refer to the *inmemory_attributes* clause for more information.

- Specify NO INMEMORY to disable the table for the IM column store.
- Specify the *inmemory_column_clause* to enable or disable specific table columns for the IM column store, and to specify the data compression method for specific columns. Refer to the *inmemory_clause* for more information.
- Specify INMEMORY ALL to mark all columns as in-memory. Specify NO INMEMORY ALL to mark all columns as not in-memory. These options are applied first to table columns before other inmemory column clauses.

  You cannot specify INMEMORY ALL and NO INMEMORY ALL in the same DDL.

If you omit this clause, then the table is assigned the default IM column store settings for the tablespace in which it is created. Refer to the *inmemory_clause* of CREATE TABLESPACE for more information on specifying the default IM column store settings for a tablespace.

In an Oracle Active Data Guard environment, if you specify this clause for a table on the primary database, then the table is enabled or disabled for the IM column store in the Oracle Active Data Guard instance.

Note:

The INMEMORY_CLAUSE_DEFAULT initialization parameter enables you to specify a default IM column store clause for new tables and materialized views. Refer to *Oracle Database Reference* for more information on the INMEMORY_CLAUSE_DEFAULT initialization parameter.

Restrictions on the In-Memory Column Store

The following restrictions apply to the In-Memory Column Store:

- You cannot specify the INMEMORY clause for index-organized tables.
- You cannot specify the INMEMORY clause for tables that are owned by the SYS schema and reside in the SYSTEM or SYSAUX tablespace.
- Starting with Oracle Database 18*c* , you can specify the INMEMORY clause for external tables. You must set the QUERY_REWRITE_INTEGRITY initialization parameter to stale_tolerated for the DDL to parse correctly. The policy may not be changed via ALTER to anything other than stale_tolerated if INMEMORY is specified.
- The IM column store does not support LONG or LONG RAW columns, out-of-line columns (LOBs, varrays, nested table columns), or extended data type columns. If you enable a table for the IM column store and it contains any of these types of columns, then the columns will not be populated in the IM column store.
- If you enable a table for the IM column store and it contains a virtual (expression) column, then the column will be populated in the IM column store only if the value of the INMEMORY_VIRTUAL_COLUMNS initialization parameter is ENABLED and the SQL expression for the virtual (expression) column refers only to columns that are enabled for the IM column store.

See Also:

Oracle Database In-Memory Guide for an overview of the IM column store

inmemory_attributes

Use the `inmemory_memcompress`, `inmemory_priority`, `inmemory_distribute`, and `inmemory_duplicate` clauses to specify how table data is stored in the IM column store.

Specify the `inmemory_spatial` clause to apply inmemory attributes to spatial columns of type `SDO_GEOMETRY`.

inmemory_memcompress

Use this clause to specify the compression method for table data stored in the IM column store. This data is called In-Memory data.

To instruct the database to not compress In-Memory data, specify `NO MEMCOMPRESS`.

Specify `MEMCOMPRESS AUTO` to instruct the database to manage the segment including actions like evict, recompress, and populate.

To instruct the database to compress In-Memory data, specify `MEMCOMPRESS FOR` followed by one of the following methods:

- `DML` - This method is optimized for DML operations and performs little or no data compression.
- `QUERY` - Specifying `QUERY` is equivalent to specifying `QUERY LOW`.
- `QUERY LOW` - This method compresses In-Memory data the least (except for `DML`) and results in the best query performance. This is the default.
- `QUERY HIGH` -This method compress In-Memory data more than `QUERY LOW`, but less than `CAPACITY LOW`.
- `CAPACITY` - Specifying `CAPACITY` is equivalent to specifying `CAPACITY LOW`.
- `CAPACITY LOW` - This method compresses In-Memory data more than `QUERY HIGH`, but less than `CAPACITY HIGH`, and results in excellent query performance.
- `CAPACITY HIGH` - This method compresses In-Memory data the most and results in good query performance.

Any memcompress level can be specified via DDL, but will be ignored during population. All In-Memory Compression Units (IMCUs) will be populated as `QUERY LOW` transparently.

inmemory_priority

Use the `PRIORITY` clause to specify the data population priority for table data in the IM column store. This clause controls the priority of population, but not the speed of population.

- Specify `NONE` for **on-demand population**. In this case, the database populates table data in the IM column store when the table it is accessed through a full table scan. If the table is never accessed, or if it is accessed only through an index scan or fetch by rowid, then population never occurs. This is the default.
- Specify one of the following priority levels for **priority-based population**: `LOW`, `MEDIUM`, `HIGH`, or `CRITICAL`. In this case, the database automatically populates table data in the IM

column store using an internally managed priority queue; a full scan is not a necessary condition for population. The database queues population of the table data based on the specified priority level. For example, a table with the setting `INMEMORY PRIORITY CRITICAL` takes precedence over a table with the setting `INMEMORY PRIORITY HIGH`, which in turn takes precedence over a table with the setting `INMEMORY PRIORITY LOW`, and so on. If the IM column store has insufficient space, then the database does not populate additional table data until space is available.

inmemory_distribute

The `DISTRIBUTE` clause is applicable only if you are using Oracle Real Application Clusters (Oracle RAC) or Oracle Active Data Guard. It lets you specify how table data in the IM column store is distributed across Oracle RAC instances, and lets you specify the database instances in which the data is eligible to be populated.

AUTO and BY

Use the `AUTO` and `BY` clauses to specify how table data in the IM column store is distributed across Oracle RAC instances. You can specify the following options:

- `AUTO` - Oracle Database controls how data is distributed across Oracle RAC instances. Large tables are distributed across Oracle RAC instances depending on their access patterns. Smaller tables may be distributed between instances. This is the default.
- `BY ROWID RANGE` - Data in certain ranges of rowids is distributed to different Oracle RAC instances.
- `BY PARTITION` - Data in partitions is distributed to different Oracle RAC instances.
- `BY SUBPARTITION` - Data in subpartitions is distributed to different Oracle RAC instances.

You can only use `AUTO` and `BY` to distribute the In-Memory Compression Units (IMCUs) for an object between instances in a single Oracle RAC database, not between a primary instance and standby instance in Active Data Guard.

FOR SERVICE

Use the `FOR SERVICE` clause to specify the Oracle RAC or Oracle Active Data Guard instances in which the object is eligible to be populated. You can specify the following options:

- `DEFAULT` - The object is eligible for population on all instances specified with the `PARALLEL_INSTANCE_GROUP` initialization parameter. If this parameter is not set, then the object is populated on all instances. This is the default.
- `ALL` - The object is eligible for population on all instances, regardless of the value of the `PARALLEL_INSTANCE_GROUP` initialization parameter.
- *`service_name`* - The object is eligible for population only on instances belonging to the specified service and only when the service is active and not blocked on an instance.
- `NONE` - The object is not eligible for population on any instances. This option lets you disable IM column store population while preserving the other In-Memory attributes for the table. These attributes take effect if you subsequently enable IM column store population for the table by specifying `FOR SERVICE DEFAULT`, `FOR SERVICE ALL`, or `FOR SERVICE` *`service_name`* in the *`inmemory_distribute`* clause of an `ALTER TABLE` statement.

In Oracle RAC, the `FOR SERVICE` clause specifies the instances within the Oracle RAC database. In Active Data Guard, the primary and standby databases may use a single-instance or Oracle RAC configuration. In Active Data Guard, the `FOR SERVICE` clause specifies instances in the primary database, instances in the standby database, or a mixture of primary and standby instances.

inmemory_duplicate

The `DUPLICATE` clause is applicable only if you are using Oracle Real Application Clusters (Oracle RAC) on an engineered system. It controls how table data in the IM column store is duplicated across Oracle RAC instances. You can specify the following options:

- `DUPLICATE` - Data is duplicated on one Oracle RAC instance, resulting in the data existing on a total of two Oracle RAC instances.
- `DUPLICATE ALL` - Data is duplicated across all Oracle RAC instances. If you specify `DUPLICATE ALL`, then the database uses the `DISTRIBUTE AUTO` setting, regardless of whether or how you specify the *`inmemory_distribute`* clause.
- `NO DUPLICATE` - Data is not duplicated across Oracle RAC instances. This is the default.

inmemory_column_clause

Use this clause to enable or disable specific table columns for the IM column store, and to specify the data compression method for specific columns. If you specify this clause when creating a `NO INMEMORY` table, then the column settings will take effect when the table or partition is subsequently enabled for the IM column store.

- Specify `INMEMORY` to enable the specified table columns for the IM column store.

  You can optionally use the *`inmemory_memcompress`* clause to specify the data compression method for specific columns. See *inmemory_memcompress*. If you omit the *`inmemory_memcompress`* clause, then the table column uses the data compression method for the table. You cannot specify the `PRIORITY`, `DISTRIBUTE`, or `DUPLICATE` settings for a specific table column. These settings are the same for all table columns as they are for the table.
- Specify `NO INMEMORY` to disable the specified table columns for the IM column store.

If you omit the *`inmemory_column_clause`*, then all table columns use the IM column store settings for the table.

Restrictions on *inmemory_column_clause*

- You cannot specify this clause for a `LONG` or `LONG RAW` column, an out-of-line column (LOB, varray, nested table column), or an extended data type column.
- To selectively enable a virtual (expression) column for the IM column store, the value of the `INMEMORY_VIRTUAL_COLUMNS` initialization parameter must be `ENABLED` or `MANUAL`, and the SQL expression for the virtual (expression) column must refer only to columns that are enabled for the IM column store.

inmemory_clause

Use this clause to enable or disable a table partition for the IM column store. In order to specify this clause, the table must be enabled for the IM column store. If you omit this clause, then the table partition uses the IM column store settings for the table.

The *`inmemory_attributes`* clause has the same semantics for table partitions as for tables. Refer to the *inmemory_attributes* clause for full information.

INMEMORY TEXT

Specify `INMEMORY TEXT` clause to enable IM full text columns. The `PRIORITY` clause has the same effect on population of IM full text columns as standard In-Memory columns. The default priority is `NONE`.

The MEMCOMPRESS clause is not valid with INMEMORY TEXT.

Examples

```
CREATE TABLE mydoc(id NUMBER, docCreationTime DATE, doc CLOB, json_doc JSON) INMEMORY
TEXT(DOC, JSON_DOC)

CREATE TABLE mydoc(id NUMBER, docCreationTime DATE, doc CLOB, json_doc JSON) INMEMORY
PRIORITY CRITICAL
    INMEMORY TEXT(DOC, JSON_DOC)
```

You can apply the IMEMORY TEXT clause to search non-scalar columns in an In-Memory table. This clause enables fast In-Memory searching of text, XML, or JSON documents using the CONTAINS () or JSON_TEXTCONTAINS() operators.

INMEMORY TEXT (column_name1, column_name2) specifies the list of columns to be enabled as IM full text. The columns must be of type CHAR, VARCHAR2, CLOB, BLOB, or JSON. JSON columns have JSON_TEXTCONTAINS() automatically enabled.

INMEMORY TEXT (column_name1 USING policy1, column_name2 USING policy2)specifies the list of columns to be enabled as IM full text along with custom indexing policies. The columns must be of type CHAR, VARCHAR2, CLOB, or BLOB. You cannot use this clause with columns of type JSON.

You can use the IMEMORY PRIORITY clause to set the order in which objects are populated.

See Also:

IM Full Text Columns.

You can specify INMEMORY on non-partitioned tables using the ORACLE_HIVE, ORACLE_HDFS, and ORACLE_BIGDATA driver types.

ilm_clause

Use this clause to add an Automatic Data Optimization policy to *table*.

This clause has the same semantics in CREATE TABLE and ALTER TABLE, with the following additional restriction: You can specify only the ADD POLICY clause for CREATE TABLE. Refer to the *ilm_clause* for the full semantics of this clause.

See Also:

Oracle Database VLDB and Partitioning Guide for more information on managing policies for Automatic Data Optimization

Restrictions on Automatic Data Optimization

Automatic Data Optimization is subject to the following restrictions:

- Automatic Data Optimization is not supported for tables that contain object types, index-organized tables, clustered tables, or materialized views.

- Row-level policies are not supported for tables that support Temporal Validity or tables that are enabled for row archiving for In-Database Archiving.

ilm_policy_clause

Use this clause to describe the Automatic Data Optimization policy.

This clause has the same semantics in `CREATE TABLE` and `ALTER TABLE`. Refer to *ilm_policy_clause* for the full semantics of this clause.

RECOVERABLE | UNRECOVERABLE

These keywords are deprecated and have been replaced with `LOGGING` and `NOLOGGING`, respectively. Although `RECOVERABLE` and `UNRECOVERABLE` are supported for backward compatibility, Oracle strongly recommends that you use the `LOGGING` and `NOLOGGING` keywords.

Restrictions on [UN]RECOVERABLE

This clause is subject to the following restrictions:

- You cannot specify `RECOVERABLE` for partitioned tables or LOB storage characteristics.
- You cannot specify `UNRECOVERABLE` for partitioned or index-organized tables.
- You can specify `UNRECOVERABLE` only with `AS` *`subquery`*.

ORGANIZATION

The `ORGANIZATION` clause lets you specify the order in which the data rows of the table are stored.

HEAP

`HEAP` indicates that the data rows of *`table`* are stored in no particular order. This is the default.

INDEX

`INDEX` indicates that *`table`* is created as an index-organized table. In an index-organized table, the data rows are held in an index defined on the primary key for the table.

EXTERNAL

`EXTERNAL` indicates that table is a read-only table located outside the database.

See Also:

"External Table Example"

index_org_table_clause

Use the *`index_org_table_clause`* to create an index-organized table. Oracle Database maintains the table rows, both primary key column values and nonkey column values, in an index built on the primary key. Index-organized tables are therefore best suited for primary key-based access and manipulation. An index-organized table is an alternative to:

- A noncluster table indexed on the primary key by using the `CREATE INDEX` statement
- A cluster table stored in an indexed cluster that has been created using the `CREATE CLUSTER` statement that maps the primary key for the table to the cluster key

You must specify a primary key for an index-organized table, because the primary key uniquely identifies a row. The primary key cannot be DEFERRABLE. Use the primary key instead of the rowid for directly accessing index-organized rows.

If an index-organized table is partitioned and contains LOB columns, then you should specify the *index_org_table_clause* first, then the *LOB_storage_clause*, and then the appropriate *table_partitioning_clauses*.

You cannot use the TO_LOB function to convert a LONG column to a LOB column in the subquery of a CREATE TABLE ... AS SELECT statement if you are creating an index-organized table. Instead, create the index-organized table without the LONG column, and then use the TO_LOB function in an INSERT ... AS SELECT statement.

The ROWID pseudocolumn of an index-organized table returns logical rowids instead of physical rowids. A column that you create with the data type ROWID cannot store the logical rowids of the IOT. The only data you can store in a column of type ROWID is rowids from heap-organized tables. If you want to store the logical rowids of an IOT, then create a column of type UROWID instead. A column of type UROWID can store both physical and logical rowids.

See Also:

"Index-Organized Table Example"

Restrictions on Index-Organized Tables

Index-organized tables are subject to the following restrictions:

- You cannot define a virtual (expression) column for an index-organized table.
- You cannot specify the *composite_range_partitions*, *composite_list_partitions*, or *composite_hash_partitions* clauses for an index-organized table.
- If the index-organized table is a nested table or varray, then you cannot specify *table_partitioning_clauses*.
- The collations of character data type columns belonging to the primary key of an index-organized table must be BINARY, USING_NLS_COMP, USING_NLS_SORT, or USING_NLS_SORT_CS.

PCTTHRESHOLD *integer*

Specify the percentage of space reserved in the index block for an index-organized table row. PCTTHRESHOLD must be large enough to hold the primary key. All trailing columns of a row, starting with the column that causes the specified threshold to be exceeded, are stored in the overflow segment. PCTTHRESHOLD must be a value from 1 to 50. If you do not specify PCTTHRESHOLD, then the default is 50.

Restriction on PCTTHRESHOLD

You cannot specify PCTTHRESHOLD for individual partitions of an index-organized table.

mapping_table_clauses

Specify MAPPING TABLE to instruct the database to create a mapping of local to physical ROWIDs and store them in a heap-organized table. This mapping is needed in order to create a bitmap index on the index-organized table. If the index-organized table is partitioned, then the mapping table is also partitioned and its partitions have the same name and physical attributes as the base table partitions.

Oracle Database creates the mapping table or mapping table partition in the same tablespace as its parent index-organized table or partition. You cannot query, perform DML operations on, or modify the storage characteristics of the mapping table or its partitions.

prefix_compression

The *prefix_compression* clauses let you enable or disable prefix compression for index-organized tables.

- Specify COMPRESS to enable **prefix compression**, also known as key compression, for an index-organized table, which eliminates repeated occurrence of primary key column values in index-organized tables. Use *integer* to specify the prefix length, which is the number of prefix columns to compress.

  The valid range of prefix length values is from 1 to the number of primary key columns minus 1. The default prefix length is the number of primary key columns minus 1.
- Specify NOCOMPRESS to disable prefix compression in index-organized tables. This is the default.

Restriction on Prefix Compression of Index-organized Tables

At the partition level, you can specify COMPRESS, but you cannot specify the prefix length with *integer*.

iot_advanced_compression

Specify *iot_advanced_compression* to compress the indexes of index organized tables (IOTs) and table partitions in order to reduce the storage footprint of IOTs.

You can enable advanced low index compression for all IOTs on specific partitions of a table, and leave other partitions uncompressed.

index_org_overflow_clause

The *index_org_overflow_clause* lets you instruct the database that index-organized table data rows exceeding the specified threshold are placed in the data segment specified in this clause.

- When you create an index-organized table, Oracle Database evaluates the maximum size of each column to estimate the largest possible row. If an overflow segment is needed but you have not specified OVERFLOW, then the database raises an error and does not execute the CREATE TABLE statement. This checking function guarantees that subsequent DML operations on the index-organized table will not fail because an overflow segment is lacking.
- All physical attributes and storage characteristics you specify in this clause after the OVERFLOW keyword apply only to the overflow segment of the table. Physical attributes and storage characteristics for the index-organized table itself, default values for all its partitions, and values for individual partitions must be specified before this keyword.
- If the index-organized table contains one or more LOB columns, then the LOBs will be stored out-of-line unless you specify OVERFLOW, even if they would otherwise be small enough be to stored inline.
- If *table* is partitioned, then the database equipartitions the overflow data segments with the primary key index segments.

INCLUDING *column_name*

Specify a column at which to divide an index-organized table row into index and overflow portions. The primary key columns are always stored in the index. `column_name` can be either the last primary key column or any non primary key column. All non primary key columns that follow `column_name` are stored in the overflow data segment.

If an attempt to divide a row at `column_name` causes the size of the index portion of the row to exceed the specified or default `PCTTHRESHOLD` value, then the database breaks up the row based on the `PCTTHRESHOLD` value.

Restriction on the INCLUDING Clause

You cannot specify this clause for individual partitions of an index-organized table.

EXTERNAL PARTITION ATTRIBUTES
Use the **EXTERNAL PARTITION ATTRIBUTES** clause to specify table level external parameters in a hybrid partitioned table.

external_table_clause

Use the `external_table_clause` to create an external table, which allows you to process data that is stored outside the database from within the database without loading any of the data into the database.

Defining an external table only creates metadata in the data dictionary, pointing to data outside the database and providing seamless read only access to such data.

Because external tables have no data in the database, you define them with a small subset of the clauses normally available when creating tables.

In addition to supporting external data residing in operating file systems and Big Data sources and formats such as HDFS and Hive, Oracle supports external data residing in objects via the `DBMS_CLOUD` package.

You can work with data in object stores using the `DBMS_CLOUD` package or by manually defining external tables. Oracle strongly recommends using `DBMS_CLOUD` for the additional functionality that is fully compatible with Oracle autonomous database.

See Also:

- *DBMS_CLOUD*
- *Managing External Tables*

- Within the `relational_properties` clause, you can specify only `column`, `datatype`, `ENCRYPT`, `inline_constraint`, and `out_of_line_constraint`. You can specify the `ENCRYPT` clause only when you specify the `ORACLE_DATAPUMP` access driver and the `AS subquery` clause to load data into the external table. Within the `inline_constraint` and `out_of_line_constraint` clauses, you can specify all subclauses except `CHECK`.
- Within the `physical_properties_clause`, you can specify only the organization of the table (`ORGANIZATION EXTERNAL external_table_clause`).
- Within the `table_properties` clause, you can specify the `parallel_clause`. The `parallel_clause` lets you parallelize subsequent queries on the external data and subsequent operations that populate the external table.

Starting with Oracle Database 12*c* Release 2 (12.2), you can create a **partitioned external table**. To do this, within the *table_properties* clause, you can specify the following subclauses of the *table_partitioning_clauses* :

  – *range_partitions* - specify this clause to create a range-partitioned or interval-partitioned external table
  – *list_partitions* - specify this clause to create a list-partitioned external table. Within this clause, you cannot specify the AUTOMATIC clause; an automatic list-partitioned table cannot be an external table.
  – *composite_range_partitions* - specify this clause to create a range-range, range-list, interval-range, or interval-list composite-partitioned external table
  – *composite_list_partitions* - specify this clause to create a list-range or list-list composite-partitioned external table. Within this clause, you cannot specify the AUTOMATIC clause; an automatic composite-partitioned table cannot be an external table.

- You can populate the external table at create time by using the AS *subquery* clause.

No other clauses are permitted in the same CREATE TABLE statement.

See Also:

- "External Table Example"
- ALTER TABLE ... "PROJECT COLUMN Clause" for information on the effect of changing the default property of the column projection
- *Oracle Database Data Warehousing Guide*, *Oracle Database Administrator's Guide*, and *Oracle Database Utilities* for information on the uses for external tables

Restrictions on External Tables

External tables are subject to the following restrictions:

- An external table cannot be a temporary table.
- You can specify only the following types of constraints on an external table: NOT NULL constraints, unique constraints, primary key constraints, and foreign key constraints. When you specify unique constraints, primary key constraints, or foreign key constraints, you must also specify RELY DISABLE. These constraints are declarative and are not enforced. They can increase query performance and reduce resource consumption because more optimizer transformations can be taken into account. In order for the optimizer to utilize these RELY DISABLE constraints, the QUERY_REWRITE_INTEGRITY initialization parameter must be set to either trusted or stale_tolerated.
- You cannot create an index on an external table.
- An external table cannot contain INVISIBLE columns.
- An external table cannot have object type, varray, or LONG columns. However, you can populate LOB columns of an external table with varray or LONG data from an internal database table.
- Only ORACLE_LOADER and ORACLE_DATAPUMP access types are permitted for external tables that can be populated into the inmemory column store.

TYPE

`TYPE` *`access_driver_type`* indicates the **access driver** of the external table. The access driver is the API that interprets the external data for the database. Oracle Database provides the following access drivers: `ORACLE_LOADER`, `ORACLE_DATAPUMP`, `ORACLE_HDFS`, and `ORACLE_HIVE`. If you do not specify `TYPE`, then the database uses `ORACLE_LOADER` as the default access driver. You must specify the `ORACLE_DATAPUMP` access driver if you specify the `AS` *`subquery`* clause to unload data from one Oracle Database and reload it into the same or a different Oracle Database.

Restrictions

`ORACLE_HIVE` column names should be limited to `[A-ZA-Z0-9_]+` on partitioning external tables.

See Also:

Oracle Database Utilities for information about the `ORACLE_LOADER`, `ORACLE_DATAPUMP`, `ORACLE_HDFS`, and `ORACLE_HIVE` access drivers

DEFAULT DIRECTORY

`DEFAULT DIRECTORY` lets you specify a default directory object corresponding to a directory on the file system where the external data sources may reside. The default directory can also be used by the access driver to store auxiliary files such as error logs.

ACCESS PARAMETERS

The optional `ACCESS PARAMETERS` clause lets you assign values to the parameters of the specific access driver for this external table.

- The *`opaque_format_spec`* specifies all access parameters for the `ORACLE_LOADER`, `ORACLE_DATAPUMP`, `ORACLE_HDFS`, and `ORACLE_HIVE` access drivers. See *Oracle Database Utilities* for descriptions of the `ORACLE_LOADER`, `ORACLE_DATAPUMP`, `ORACLE_HDFS`, and `ORACLE_HIVE` access parameters.

  Field names specified in the *`opaque_format_spec`* must match columns in the table definition. Oracle Database ignores any field in the *`opaque_format_spec`* that is not matched by a column in the table definition.
- `USING CLOB` *`subquery`* lets you derive the parameters and their values through a subquery. The subquery cannot contain any set operators or an `ORDER BY` clause. It must return one row containing a single item of data type `CLOB`.

Whether you specify the parameters in an *`opaque_format_spec`* or derive them using a subquery, the database does not interpret anything in this clause. It is up to the access driver to interpret this information in the context of the external data.

For inline external tables and external modify query statements you must use *`opaque_format_spec`* within single quotes. For DDL statements you must use *`opaque_format_spec`* without single quotes.

LOCATION

The `LOCATION` clause lets you specify one or more external data sources. Usually the *`location_specifier`* is a file, but it need not be. Oracle Database does not interpret this clause. It is up to the access driver to interpret this information in the context of the external data.

You must specify the `LOCATION` clause as follows:

- When creating a nonpartitioned external table, you must specify the `LOCATION` clause at the table level in the *`external_table_data_props`* clause.
- When creating a partitioned external table, you must specify the `LOCATION` clause at the partition level in the *`external_part_subpart_data_props`* clause.
- When creating a composite-partitioned external table, you must specify the `LOCATION` clause at the subpartition level in the *`external_part_subpart_data_props`* clause.

REJECT LIMIT

The `REJECT LIMIT` clause lets you specify how many conversion errors can occur during a query of the external data before an Oracle Database error is returned and the query is aborted. The default value is 0.

CLUSTER Clause

The `CLUSTER` clause indicates that the table is to be part of *`cluster`*. The columns listed in this clause are the table columns that correspond to the cluster columns. Generally, the cluster columns of a table are the column or columns that make up its primary key or a portion of its primary key. Refer to CREATE CLUSTER for more information.

Specify one column from the table for each column in the cluster key. The columns are matched by position, not by name.

A cluster table uses the space allocation of the cluster. Therefore, do not use the `PCTFREE`, `PCTUSED`, or `INITRANS` parameters, the `TABLESPACE` clause, or the *`storage_clause`* with the `CLUSTER` clause.

Restrictions on Cluster Tables

Cluster tables are subject to the following restrictions:

- Object tables and tables containing LOB columns or columns of the `Any*` Oracle-supplied types cannot be part of a cluster.
- You cannot specify the *`parallel_clause`* or `CACHE` or `NOCACHE` for a table that is part of a cluster.
- You cannot specify `CLUSTER` with either `ROWDEPENDENCIES` or `NOROWDEPENDENCIES` unless the cluster has been created with the same `ROWDEPENDENCIES` or `NOROWDEPENDENCIES` setting.
- A cluster table cannot contain `INVISIBLE` columns.

table_properties

The *`table_properties`* further define the characteristics of the table.

column_properties

Use the *`column_properties`* clauses to specify the storage attributes of a column.

object_type_col_properties

The *`object_type_col_properties`* determine storage characteristics of an object column or attribute or of an element of a collection column or attribute.

column

For *`column`*, specify an object column or attribute.

substitutable_column_clause

The *substitutable_column_clause* indicates whether object columns or attributes in the same hierarchy are substitutable for each other. You can specify that a column is of a particular type, or whether it can contain instances of its subtypes, or both.

- If you specify ELEMENT, then you constrain the element type of a collection column or attribute to a subtype of its declared type.
- The IS OF [TYPE] (ONLY *type*) clause constrains the type of the object column to a subtype of its declared type.
- NOT SUBSTITUTABLE AT ALL LEVELS indicates that the object column cannot hold instances corresponding to any of its subtypes. Also, substitution is disabled for any embedded object attributes and elements of embedded nested tables and varrays. The default is SUBSTITUTABLE AT ALL LEVELS.

Restrictions on the *substitutable_column_clause*

This clause is subject to the following restrictions:

- You cannot specify this clause for an attribute of an object column. However, you can specify this clause for a object type column of a relational table and for an object column of an object table if the substitutability of the object table itself has not been set.
- For a collection type column, the only part of this clause you can specify is [NOT] SUBSTITUTABLE AT ALL LEVELS.

LOB_storage_clause

The *LOB_storage_clause* lets you specify the storage attributes of LOB data segments. You must specify at least one clause after the STORE AS keywords. If you specify more than one clause, then you must specify them in the order shown in the syntax diagram, from top to bottom.

For a nonpartitioned table, this clause specifies the storage attributes of LOB data segments of the table.

For a partitioned table, Oracle Database implements this clause depending on where it is specified:

- For a partitioned table specified at the table level—when specified in the *physical_properties* clause along with one of the partitioning clauses—this clause specifies the default storage attributes for LOB data segments associated with each partition or subpartition. These storage attributes apply to all partitions or subpartitions unless overridden by a *LOB_storage_clause* at the partition or subpartition level.
- For an individual partition of a partitioned table—when specified as part of a *table_partition_description*—this clause specifies the storage attributes of the data segments of the partition or the default storage attributes of any subpartitions of the partition. A partition-level *LOB_storage_clause* overrides a table-level *LOB_storage_clause*.
- For an individual subpartition of a partitioned table—when specified as part of *subpartition_by_hash* or *subpartition_by_list*—this clause specifies the storage attributes of the data segments of the subpartition. A subpartition-level *LOB_storage_clause* overrides both partition-level and table-level *LOB_storage_clauses*.

Restriction on the *LOB_storage_clause*:

Only the TABLESPACE clause is allowed when specifying the *LOB_storage_clause* in a subpartition.

See Also:

- *Oracle Database SecureFiles and Large Objects Developer's Guide* for detailed information about LOBs, including guidelines for creating gigabyte LOBs
- "Creating a Table: LOB Column Example"

LOB_item

Specify the LOB column name or LOB object attribute for which you are explicitly defining tablespace and storage characteristics that are different from those of the table. Oracle Database automatically creates a system-managed index for each `LOB_item` you create.

SECUREFILE | BASICFILE

Use this clause to specify the type of LOB storage, either high-performance LOB (SecureFiles), or the traditional LOB (BasicFiles).

See Also:

Oracle Database SecureFiles and Large Objects Developer's Guide for more information about SecureFiles LOBs

Note:

You cannot convert a LOB from one type of storage to the other. Instead you must migrate to SecureFiles or BasicFiles by using online redefinition or partition exchange.

LOB_segname

Specify the name of the LOB data segment. You cannot use `LOB_segname` if you specify more than one `LOB_item`.

LOB_storage_parameters

The `LOB_storage_parameters` clause lets you specify various elements of LOB storage.

TABLESPACE Clause

Use this clause to specify the tablespace in which LOB data is to be stored.

TABLESPACE SET Clause

This clause is valid only when creating a sharded table by specifying the `SHARDED` keyword of `CREATE TABLE`. Use this clause to specify the tablespace set in which LOB data is to be stored.

storage_clause

Use the `storage_clause` to specify various aspects of LOB segment storage. Of particular interest in the context of LOB storage is the `MAXSIZE` clause of the `storage_clause`, which can be used in combination with the *LOB_retention_clause* of *LOB_parameters*. Refer to *storage_clause* for more information.

LOB_parameters

Several of the *LOB_parameters* are no longer needed if you are using SecureFiles for LOB storage. The PCTVERSION and FREEPOOLS parameters are valid and useful only if you are using BasicFiles LOB storage.

ENABLE STORAGE IN ROW

If you enable storage in row, then the LOB value is stored in the row (inline) if its length is less than approximately 4000 bytes minus system control information. This is the default.

Restriction on Enabling Storage in Row

For an index-organized table, you cannot specify this parameter unless you have specified an OVERFLOW segment in the *index_org_table_clause*.

DISABLE STORAGE IN ROW

If you disable storage in row, then the LOB value is stored outside of the row out of line regardless of the length of the LOB value.

The LOB locator is always stored inline regardless of where the LOB value is stored. You cannot change the value of STORAGE IN ROW once it is set except by moving the table. See the *move_table_clause* in the ALTER TABLE documentation for more information.

CHUNK *integer*

Specify the number of bytes to be allocated for LOB manipulation. If *integer* is not a multiple of the database block size, then the database rounds up in bytes to the next multiple. For example, if the database block size is 2048 and *integer* is 2050, then the database allocates 4096 bytes (2 blocks). The maximum value is 32768 (32K), which is the largest Oracle Database block size allowed. The default CHUNK size is one Oracle Database block.

The value of CHUNK must be less than or equal to the value of NEXT, either the default value or that specified in the *storage_clause*. If CHUNK exceeds the value of NEXT, then the database returns an error. You cannot change the value of CHUNK once it is set.

PCTVERSION *integer*

Specify the maximum percentage of overall LOB storage space used for maintaining old versions of the LOB. If the database is running in manual undo mode, then the default value is 10, meaning that older versions of the LOB data are not overwritten until they consume 10% of the overall LOB storage space.

You can specify the PCTVERSION parameter whether the database is running in manual or automatic undo mode. PCTVERSION is the default in manual undo mode. RETENTION is the default in automatic undo mode. You cannot specify both PCTVERSION and RETENTION.

This clause is not valid if you have specified SECUREFILE. If you specify both SECUREFILE and PCTVERSION, then the database silently ignores the PCTVERSION parameter.

LOB_retention_clause

Use this clause to specify whether you want the LOB segment retained for flashback purposes, consistent-read purposes, both, or neither.

You can specify the RETENTION parameter only if the database is running in automatic undo mode. Oracle Database uses the value of the UNDO_RETENTION initialization parameter to determine the amount of committed undo data to retain in the database. In automatic undo mode, RETENTION is the default value unless you specify PCTVERSION. You cannot specify both PCTVERSION and RETENTION.

You can specify the optional settings after RETENTION only if you are using SecureFiles. The SECUREFILE parameter of the *LOB_storage_clause* indicates that the database will use SecureFiles to manage storage dynamically, taking into account factors such as the undo mode of the database.

- Specify MAX to signify that the undo should be retained until the LOB segment has reached MAXSIZE. If you specify MAX, then you must also specify the MAXSIZE clause in the *storage_clause*.
- Specify MIN if the database is in flashback mode to limit the undo retention **duration** for the specific LOB segment to *n* seconds.
- Specify AUTO if you want to retain undo sufficient for consistent read purposes only.
- Specify NONE if no undo is required for either consistent read or flashback purposes.

If you do not specify the RETENTION parameter, or you specify RETENTION with no optional settings, then RETENTION is set to DEFAULT, which is functionally equivalent to AUTO.

See Also:

- To set the UNDO_RETENTION initialization parameter, see Setting the Minimum Undo Retention Period
- CREATE TABLE clause *LOB_storage_parameters* for more information on simplified LOB storage using SecureFiles
- *Oracle Database SecureFiles and Large Objects Developer's Guide* for more information on using SecureFiles
- *flashback_mode_clause* of ALTER DATABASE for information on putting a database in flashback mode
- "Creating an Undo Tablespace: Example"

FREEPOOLS *integer*

Specify the number of groups of free lists for the LOB segment. Normally *integer* will be the number of instances in an Oracle Real Application Clusters environment or 1 for a single-instance database.

You can specify this parameter only if the database is running in automatic undo mode. In this mode, FREEPOOLS is the default unless you specify the FREELIST GROUPS parameter of the *storage_clause*. If you specify neither FREEPOOLS nor FREELIST GROUPS, then the database uses a default of FREEPOOLS 1 if the database is in automatic undo management mode and a default of FREELIST GROUPS 1 if the database is in manual undo management mode.

This clause is not valid if you have specified SECUREFILE. If you specify both SECUREFILE and FREEPOOLS, then the database silently ignores the FREEPOOLS parameter.

Restriction on FREEPOOLS

You cannot specify both FREEPOOLS and the FREELIST GROUPS parameter of the *storage_clause*.

LOB_deduplicate_clause

This clause is valid only for SecureFiles LOBs. Use the `LOB_deduplicate_clause` to enable or disable LOB deduplication, which is the elimination of duplicate LOB data.

The `DEDUPLICATE` keyword instructs the database to eliminate duplicate copies of LOBs. Using a secure hash index to detect duplication, the database coalesces LOBs with identical content into a single copy, reducing storage consumption and simplifying storage management.

If you omit this clause, then LOB deduplication is disabled by default.

This clause implements LOB deduplication for the entire LOB segment. To enable or disable deduplication for an individual LOB, use the `DBMS_LOB.SETOPTIONS` procedure.

> **See Also:**
>
> *Oracle Database SecureFiles and Large Objects Developer's Guide* for more information about LOB deduplication and *Oracle Database PL/SQL Packages and Types Reference* for information about about the `DBMS_LOB` package

LOB_compression_clause

This clause is valid only for SecureFiles LOBs, not for BasicFiles LOBs. Use the `LOB_compression_clause` to instruct the database to enable or disable server-side LOB compression. Random read/write access is possible on server-side compressed LOB segments. LOB compression is independent from table compression or index compression. If you omit this clause, then the default is `NOCOMPRESS`.

You can specify `HIGH`, `MEDIUM`, or `LOW` to vary the degree of compression. The `HIGH` degree of compression incurs higher latency than `MEDIUM` but provides better compression. The `LOW` degree results in significantly higher decompression and compression speeds, at the cost of slightly lower compression ratio than either `HIGH` or `MEDIUM`. If you omit this optional parameter, then the default is `MEDIUM`.

This clause implements server-side LOB compression for the entire LOB segment. To enable or disable compression on an individual LOB, use the `DBMS_LOB.SETOPTIONS` procedure.

> **See Also:**
>
> *Oracle Database SecureFiles and Large Objects Developer's Guide* for more information on server-side LOB storage and *Oracle Database PL/SQL Packages and Types Reference* for information about client-side LOB compression using the `UTL_COMPRESS` supplied package and for information about the `DBMS_LOB` package

ENCRYPT | DECRYPT

These clauses are valid only for LOBs that are using SecureFiles for LOB storage. Specify `ENCRYPT` to encrypt all LOBs in the column. Specify `DECRYPT` to keep the LOB in cleartext. If you omit this clause, then `DECRYPT` is the default.

Refer to *encryption_spec* for general information on that clause. When applied to a LOB column, `encryption_spec` is specific to the individual LOB column, so the encryption algorithm can differ from that of other LOB columns and other non-LOB columns. Use the *`encryption_spec`* as part of the *`column_definition`* to encrypt the entire LOB column. Use the *`encryption_spec`* as part of the *`LOB_storage_clause`* in the *`table_partition_description`* to encrypt a LOB partition.

Restriction on *encryption_spec* for LOBs

You cannot specify the `SALT` or `NO SALT` clauses of *`encryption_spec`* for LOB encryption.

See Also:

Oracle Database SecureFiles and Large Objects Developer's Guide for more information on LOB encryption and *Oracle Database PL/SQL Packages and Types Reference* for information the `DBMS_LOB` package

CACHE | NOCACHE | CACHE READS

Refer to CACHE | NOCACHE | CACHE READS for information on these clauses.

LOB_partition_storage

The *`LOB_partition_storage`* clause lets you specify a separate *`LOB_storage_clause`* or *`varray_col_properties`* clause for each partition. You must specify the partitions in the order of partition position. You can find the order of the partitions by querying the `PARTITION_NAME` and `PARTITION_POSITION` columns of the `USER_IND_PARTITIONS` view.

If you do not specify a *`LOB_storage_clause`* or *`varray_col_properties`* clause for a particular partition, then the storage characteristics are those specified for the LOB item at the table level. If you also did not specify any storage characteristics for the LOB item at the table level, then Oracle Database stores the LOB data partition in the same tablespace as the table partition to which it corresponds.

Restrictions on *LOB_partition_storage*

`LOB_partition_storage` is subject to the following restrictions:

- In the *`LOB_parameters`* of the *`LOB_storage_clause`*, you cannot specify *`encryption_spec`*, because it is invalid to specify an encryption algorithm for partitions and subpartitions.
- You can only specify the `TABLESPACE` clause for hash partitions and all types of subpartitions.

varray_col_properties

The *`varray_col_properties`* let you specify separate storage characteristics for the LOB in which a varray will be stored. If *`varray_item`* is a multilevel collection, then the database stores all collection items nested within *`varray_item`* in the same LOB in which *`varray_item`* is stored.

- For a nonpartitioned table—when specified in the *`physical_properties`* clause without any of the partitioning clauses—this clause specifies the storage attributes of the LOB data segments of the varray.
- For a partitioned table specified at the table level—when specified in the *`physical_properties`* clause along with one of the partitioning clauses—this clause specifies the default storage attributes for the varray LOB data segments associated with each partition (or its subpartitions, if any).

- For an individual partition of a partitioned table—when specified as part of a *table_partition_description*—this clause specifies the storage attributes of the varray LOB data segments of that partition or the default storage attributes of the varray LOB data segments of any subpartitions of this partition. A partition-level *varray_col_properties* overrides a table-level *varray_col_properties*.
- For an individual subpartition of a partitioned table—when specified as part of *subpartition_by_hash* or *subpartition_by_list*—this clause specifies the storage attributes of the varray data segments of this subpartition. A subpartition-level *varray_col_properties* overrides both partition-level and table-level *varray_col_properties*.

STORE AS [SECUREFILE | BASICFILE] LOB Clause

If you specify `STORE AS LOB`, then:

- If the maximum varray size is less than approximately 4000 bytes, then the database stores the varray as an inline LOB unless you have disabled storage in row.
- If the maximum varray size is greater than approximately 4000 bytes or if you have disabled storage in row, then the database stores in the varray as an out-of-line LOB.

If you do not specify `STORE AS LOB`, then storage is based on the maximum possible size of the varray rather than on the actual size of a varray column. The maximum size of the varray is the number of elements times the element size, plus a small amount for system control information. If you omit this clause, then:

- If the maximum size of the varray is less than approximately 4000 bytes, then the database does not store the varray as a LOB, but as inline data.
- If the maximum size is greater than approximately 4000 bytes, then the database always stores the varray as a LOB.
  - If the actual size is less than approximately 4000 bytes, then it is stored as an inline LOB
  - If the actual size is greater than approximately 4000 bytes, then it is stored as an out-of-line LOB, as is true for other LOB columns.

substitutable_column_clause

The *substitutable_column_clause* has the same behavior as described for object_type_col_properties.

See Also:

"Substitutable Table and Column Examples"

Restriction on Varray Column Properties

You cannot specify this clause on an interval partitioned table.

nested_table_col_properties

The *nested_table_col_properties* let you specify separate storage characteristics for a nested table, which in turn enables you to define the nested table as an index-organized table. Unless you explicitly specify otherwise in this clause:

- For a nonpartitioned table, the storage table is created in the same schema and the same tablespace as the parent table.
- For a partitioned table, the storage table is created in the default tablespace of the schema. By default, nested tables are equipartitioned with the partitioned base table.
- In either case, the storage table uses default storage characteristics, and stores the nested table values of the column for which it was created.

You must include this clause when creating a table with columns or column attributes whose type is a nested table. Clauses within `nested_table_col_properties` that function the same way they function for the parent table are not repeated here.

nested_item

Specify the name of a column, or of a top-level attribute of the object type of the tables, whose type is a nested table.

COLUMN_VALUE

If the nested table is a multilevel collection, then the inner nested table or varray may not have a name. In this case, specify `COLUMN_VALUE` in place of the `nested_item` name.

See Also:

"Creating a Table: Multilevel Collection Example" for examples using `nested_item` and `COLUMN_VALUE`

LOCAL | GLOBAL

Specify `LOCAL` to equipartition the nested table with the base table. This is the default. Oracle Database automatically creates a local partitioned index for the partitioned nested table.

Specify `GLOBAL` to indicate that the nested table is a nonpartitioned nested table of a partitioned base table.

storage_table

Specify the name of the table where the rows of `nested_item` reside.

You cannot query or perform DML statements on `storage_table` directly, but you can modify its storage characteristics by specifying its name in an `ALTER TABLE` statement.

See Also:

ALTER TABLE for information about modifying nested table column storage characteristics

RETURN [AS]

Specify what Oracle Database returns as the result of a query.

- `VALUE` returns a copy of the nested table itself.
- `LOCATOR` returns a collection locator to the copy of the nested table.

The locator is scoped to the session and cannot be used across sessions. Unlike a LOB locator, the collection locator cannot be used to modify the collection instance.

If you do not specify the *segment_attributes_clause* or the *LOB_storage_clause*, then the nested table is heap organized and is created with default storage characteristics.

Restrictions on Nested Table Column Properties

Nested table column properties are subject to the following restrictions:

- You cannot specify this clause for a temporary table.
- You cannot specify this clause on an interval partitioned table.
- You cannot specify the *oid_clause*.
- At create time, you cannot use *object_properties* to specify an *out_of_line_ref_constraint*, *inline_ref_constraint*, or foreign key constraint for the attributes of a nested table.

See Also:

- ALTER TABLE for information about modifying nested table column storage characteristics
- "Nested Table Example" and "Creating a Table: Multilevel Collection Example"

XMLType_column_properties

The *XMLType_column_properties* let you specify storage attributes for an `XMLTYPE` column.

XMLType_storage

`XMLType` tables and columns data can be stored as transportable binary XML, binary XML, or a set of objects in object-relational storage.

- Specify `TRANSPORTABLE BINARY XML` to store XML data in a transportable format that is scalable and distributed. See Create Tables and Columns as Transportable Binary XML for examples.
  - `TRANSPORTABLE BINARY XML` can only be stored in `SECUREFILE`.
  - Any LOB parameters you specify are applied to the underlying BLOB column created for storing the transportable binary XML encoded value. .
  - You can not specify the *XMLSchema_spec* clause for `TRANSPORTABLE BINARY XML`.
- Specify `BINARY XML` to store the XML data in compact binary XML format.

  Any LOB parameters you specify are applied to the underlying `BLOB` column created for storing the binary XML encoded value.

  In earlier releases, binary XML data is stored by default in a BasicFiles LOB. Beginning with Oracle Database 11*g* Release 2 (11.2.0.2), if the `COMPATIBLE` initialization parameter is 11.2 or higher and you do not specify `BASICFILE` or `SECUREFILE`, then binary XML data is stored in a SecureFiles LOB whenever possible. If SecureFiles LOB storage is not possible then the binary XML data is stored in a BasicFiles LOB. This can occur if either of the following is true:

  - The tablespace for the XMLType table does not use automatic segment space management.
  - A setting in file init.ora prevents SecureFiles LOB storage. For example, see parameter DB_SECUREFILE in *Oracle Database Reference*.

- Specify CLOB if you want the database to store the XMLType data in a CLOB column. Storing data in a CLOB column preserves the original content and enhances retrieval time.

  If you specify LOB storage, then you can specify either LOB parameters or the *XMLSchema_spec* clause, but not both. Specify the *XMLSchema_spec* clause if you want to restrict the table or column to particular schema-based XML instances.

  If you do not specify BASICFILE or SECUREFILE with this clause, then the CLOB column is stored in a BasicFiles LOB.

  > **Note:**
  >
  > Oracle recommends against storing XMLType data in a CLOB column. CLOB storage of XMLType is deprecated. Use binary XML storage of XMLType instead.

- Specify OBJECT RELATIONAL if you want the database to store the XMLType data in object-relational columns. Storing data objects relationally lets you define indexes on the relational columns and enhances query performance.

  If you specify object-relational storage, then you must also specify the *XMLSchema_spec* clause.

  Use the ALL VARRAYS AS clause if you want the database to store all varrays in an XMLType column.

In earlier releases, XMLType data is stored in a CLOB column in a BasicFiles LOB by default. Beginning with Oracle Database 11*g* Release 2 (11.2.0.2), if the COMPATIBLE initialization parameter is 11.2 or higher and you do not specify the *XMLType_storage* clause, then XMLType data is stored in a binary XML column in a SecureFiles LOB. If SecureFiles LOB storage is not possible, then it is stored in a binary XML column in a BasicFiles LOB.

> **See Also:**
>
> *Oracle Database SecureFiles and Large Objects Developer's Guide* for more information on SecureFiles LOBs

XMLSchema_spec

Refer to the XMLSchema_spec for the full semantics of this clause.

See Also:

- *LOB_storage_clause* for information on the *`LOB_segname`* and *`LOB_parameters`* clauses
- "XMLType Column Examples" for examples of `XMLType` columns in object-relational tables and "Using XML in SQL Statements " for an example of creating an XMLSchema
- *Oracle XML DB Developer's Guide* for more information on `XMLType` columns and tables and on creating XMLSchemas
- *Oracle Database PL/SQL Packages and Types Reference* for information on the `DBMS_XMLSCHEMA` package

XMLType_virtual_columns

This clause is valid only for `XMLType` tables with binary XML storage, which you designate in the *`XMLType_storage`* clause. Specify the `VIRTUAL COLUMNS` clause to define virtual (expression) columns, which can be used as in a function-based index or in the definition of a constraint. You cannot define a constraint on such a virtual (expression) column during creation of the table, but you can use a subsequent `ALTER TABLE` statement to add a constraint to the column.

See Also:

Oracle XML DB Developer's Guide for examples of how to use this clause in an XML environment

json_storage_clause

With support for `JSON` data type you can define a column of `JSON` data type using the *`JSON_storage_clause`*.

You can specify the JSON type modifier when you specify a JSON type column definition as follows:

Create a Table with a JSON Type Column: Example

This example creates table `j_purchaseorder` with `JSON` data type column `po_document`. Oracle recommends that you store JSON data as `JSON` type.

```
CREATE TABLE j_purchaseorder
        (id VARCHAR2 (32) NOT NULL PRIMARY KEY,
         date_loaded TIMESTAMP (6) WITH TIME ZONE,
         po_document JSON );
```

See Also:

For more information on creating a JSON column see *Creating a Table with a JSON Column of the JSON developer's Guide.*

read_only_clause

This clause lets you specify whether to create a table, partition, or subpartition in read-only or read/write mode.

- Use `READ ONLY` to specify read-only mode. When an object is in read-only mode, you cannot issue any DML statements that affect the object or any `SELECT` ... `FOR UPDATE` ... statements on the object. You can issue DDL statements as long as they do not modify any table data. See *Oracle Database Administrator’s Guide* for the complete list of operations that are allowed and disallowed on read-only objects.
- Use `READ WRITE` to specify read/write mode. This is the default.

When you specify this clause for a partitioned table, you specify the default read-only or read/write mode for the table. This mode is assigned to all partitions in the table at creation time, as well as any partitions that are subsequently added to the table, unless you override this behavior by specifying the mode at the partition level.

When you specify this clause for a composite-partitioned table, you specify the default read-only or read/write mode for all partitions in the table. You can override this behavior by specifying this clause for a particular partition. The default mode of a partition is assigned to all subpartitions in the partition at creation time, as well as any subpartitions that are subsequently added to the partition, unless you override this behavior by specifying the mode at the subpartition level.

indexing_clause

The *`indexing_clause`* is valid only for partitioned tables. Use this clause to set the **indexing property** for a table, table partition, or table subpartition.

- Specify `INDEXING ON` to set the indexing property to `ON`. This is the default.
- Specify `INDEXING OFF` to set the indexing property to `OFF`.

The indexing property determines whether table partitions and subpartitions are included in partial indexes on the table.

- For simple partitioned tables, partitions with an indexing property of `ON` are included in partial indexes on the table. Partitions with an indexing property of `OFF` are excluded.
- For composite-partitioned tables, subpartitions with an indexing property of `ON` are included in partial indexes on the table. Subpartitions with an indexing property of `OFF` are excluded.

You can specify the *`indexing_clause`* at the table, partition, or subpartition level. When you specify the *`indexing_clause`* at the table level, in the *`table_properties`* clause, you set the default indexing property for the table. Interval partitions, which are automatically created by the database, always inherit the default indexing property for the table. Other types of partitions and subpartitions inherit the default indexing property as follows:

- For simple partitioned tables, partitions inherit the default indexing property for the table. You can override this behavior by specifying the *`indexing_clause`* for an individual partition:
  - For a range partition, in the *`table_partition_description`* of the *`range_partitions`* clause
  - For a hash partition, in the *`individual_hash_partitions`* clause of the *`hash_partitions`* clause
  - For a list partition, in the *`table_partition_description`* of the *`list_partitions`* clause

  - For a reference partition, in the *table_partition_description* of the *reference_partition_desc* clause of the *reference_partitioning* clause
  - For a system partition, in the *table_partition_description* of the *reference_partition_desc* clause of the *system_partitioning* clause

- For composite-partitioned tables, subpartitions inherit the default indexing property for the table. You can override this behavior by specifying the *indexing_clause* for an individual partition or subpartition.

  If you specify the *indexing_clause* for a partition, then its subpartitions inherit the indexing property of the partition:

  - For composite range partitions, in the *table_partition_description* of the *composite_range_partitions* clause
  - For composite list partitions, in the *table_partition_description* of the *composite_list_partitions* clause
  - For composite hash partitions, in the *individual_hash_partitions* clause of the *composite_hash_partitions* clause

  You can set the indexing property of a subpartition by specifying the *indexing_clause* for the subpartition:

  - For range subpartitions, in the *range_subpartition_desc* clause of the *composite_range_partitions* clause
  - For list subpartitions, in the *list_subpartition_desc* clause of the *composite_list_partitions* clause
  - For hash subpartitions, in the *individual_hash_subparts* clause of the *composite_hash_partitions* clause

See Also:

Oracle Database Reference for information on viewing the indexing property of a table, table partition, or table subpartition.

- To view the default indexing property of a table, query the `DEF_INDEXING` column of the `*_PART_TABLES` views.
- To view the indexing property of a table partition, query the `INDEXING` column of the `*_TAB_PARTITIONS` views.
- To view the indexing property of a table subpartition, query the `INDEXING` column of the `*_TAB_SUBPARTITIONS` views.

Restrictions on the *indexing_clause*

The *indexing_clause* is subject to the following restrictions:

- You cannot specify the *indexing_clause* for nonpartitioned tables.
- You cannot specify the *indexing_clause* for index-organized tables.

See Also:

The *partial_index_clause* of `CREATE INDEX` for more information on partial indexes

table_partitioning_clauses

Use the *`table_partitioning_clauses`* to create a partitioned table.

Notes on Partitioning in General

The following notes pertain to all types of partitioning:

- You can specify up to a total of 1024K-1 partitions and subpartitions.
- You can create a partitioned table with just one partition. A table with one partition is different from a nonpartitioned table. For example, you cannot add a partition to a nonpartitioned table.
- You can specify a name for every table and LOB partition and for every table and LOB subpartition, but you need not do so. If you specify a 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 name, then the database generates names as follows:
  - If you omit a partition name, then the database generates a name of the form `SYS_P`*`n`*. System-generated names for LOB data and LOB index partitions take the form `SYS_LOB_P`*`n`* and `SYS_IL_P`*`n`*, respectively.
  - If you specify a subpartition name in *`subpartition_template`*, then for each subpartition created with that template, the database generates a name by concatenating the partition name with the template subpartition name. For LOB subpartitions, the generated LOB subpartition name is a concatenation of the partition name and the template LOB segment name. If the `COMPATIBLE` initialization parameter is set to `12.2` or higher, then the maximum length of the concatenation is 128 bytes; otherwise, the maximum length is 30 bytes. If the concatenation exceeds the maximum length, then the database returns an error and the statement fails.
  - If you omit a subpartition name when specifying an individual subpartition, and you have not specified *`subpartition_template`*, then the database generates a name of the form `SYS_SUBP`*`n`*. The corresponding system-generated names for LOB data and index subpartitions are `SYS_LOB_SUBP`*`n`* and `SYS_IL_SUBP`*`n`*, respectively.
- Tablespace storage can be specified at various levels in the `CREATE TABLE` statement for both table segments and LOB segments. The number of tablespaces does not have to equal the number of partitions or subpartitions. If the number of partitions or subpartitions is greater than the number of tablespaces, then the database cycles through the names of the tablespaces.

  The database evaluates tablespace storage in the following order of descending priority:
  - Tablespace storage specified at the individual table subpartition or LOB subpartition level has the highest priority, followed by storage specified for the partition or LOB in the *`subpartition_template`*.
  - Tablespace storage specified at the individual table partition or LOB partition level. Storage parameters specified here take precedence over the *`subpartition_template`*.
  - Tablespace storage specified for the table
  - Default tablespace storage specified for the user

- By default, nested tables are equipartitioned with the partitioned base table.

Restrictions on Partitioning in General

All partitioning is subject to the following restrictions:

- You cannot partition a table that is part of a cluster.
- You cannot partition a nested table or varray that is defined as an index-organized table.
- You cannot partition a table containing any `LONG` or `LONG RAW` columns.

Restrictions on Hybrid Partitioned Tables

- Restrictions that apply to external tables also apply to hybrid partitioned tables unless explicitly noted. Only DML operations are supported on internal partitions of a hybrid partitioned table (external partitions are treated as read-only partitions).
- Only single level `LIST`, `RANGE`, interval and autolist partitioning are supported.
- The following DDLs are supported: `ADD PARTITION`, `ADD SUBPARTITION`, `DROP PARTITION`, `DROP SUBPARTITION`, `EXCHANGE PARTITION`, `EXCHANGE SUBPARTITION` on internal and external partitions and subpartitions. Partitions and subpartitions can also be renamed.
- The DDLs `MOVE`, `SPLIT` and `MERGE` are supported on internal partition and subpartitions only.
- Existing DMLs are supported on internal partitions only. External partitions are treated as read-only. All triggers are supported.
- Table level non-enforced constraints in mandatory `RELY DISABLE` mode are supported. The only supported enforced constraint is `NOT NULL` constraint.
- Unique indexes and global unique indexes are not supported. Only non-unique partial indexes are supported on internal partitions. External partitions are not indexable.
- Only single level list partitioning is supported for HIVE.
- Attribute clustering (`CLUSTERING` clause) is not allowed.
- In-memory defined on the table level only has an effect on internal partitions of the hybrid partitioned table.
- No column default value is allowed.
- Invisible columns are not allowed.
- The `CELLMEMORY` clause is not allowed.
- `SPLIT`, `MERGE`, and `MOVE` maintenance operations are not allowed on external partitions.

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:

"Partitioning Examples"

range_partitions

Use the *`range_partitions`* clause to partition the table on ranges of values from the column list. For an index-organized table, the column list must be a subset of the primary key columns of the table.

Restrictions on Range Partitioning

Range partitioning is subject to the restrictions listed in "Restrictions on Partitioning in General". The following additional restrictions apply:

- You cannot specify more than 16 partitioning key columns.
- Partitioning key columns must be of type `CHAR`, `NCHAR`, `VARCHAR2`, `NVARCHAR2`, `VARCHAR`, `NUMBER`, `FLOAT`, `DATE`, `TIMESTAMP`, `TIMESTAMP WITH LOCAL TIMEZONE`, or `RAW`.
- Each range partitioning key column with a character data type that belongs to an `XMLType` table or a table with an `XMLType` column, or that is used as a sharding key column must have one of the following declared collations: `BINARY`, `USING_NLS_COMP`, `USING_NLS_SORT`, or `USING_NLS_SORT_CS`. For all these collations, partition bounds are checked using the collation `BINARY`.
- You cannot specify NULL in the `VALUES` clause.

column

Specify an ordered list of columns used to determine into which partition a row belongs. These columns are the **partitioning key**. You can specify virtual (expression) columns and `INVISIBLE` columns as partitioning key columns.

INTERVAL Clause

Use this clause to establish **interval partitioning** for the table. Interval partitions are partitions based on a numeric range or datetime interval. They extend range partitioning by instructing the database to create partitions of the specified range or interval automatically when data inserted into the table exceeds all of the range partitions. For each automatically created partition, the database generates a name of the form `SYS_Pn`. The database guarantees that automatically generated partition names are unique and do not violate namespace rules.

- For *`expr`*, specify a valid number or interval expression.
- The optional `STORE IN` clause lets you specify one or more tablespaces into which the database will store interval partition data.
- You must also specify at least one range partition using the `PARTITION` clause of *`range_partitions`*. The range partition key value determines the high value of the range partitions, which is called the **transition point**, and the database creates interval partitions for data beyond that transition point.

Restrictions on Interval Partitioning

The `INTERVAL` clause is subject to the restrictions listed in "Restrictions on Partitioning in General" and "Restrictions on Range Partitioning". The following additional restrictions apply:

- You can specify only one partitioning key column, and it must be of type `NUMBER`, `DATE`, `FLOAT`, `TIMESTAMP`, or `TIMESTAMP WITH LOCAL TIME ZONE`.
- This clause is not supported for index-organized tables.
- This clause is not supported for tables containing varray columns.
- You cannot create an interval-partitioned table with equipartitioned nested tables. If you create an interval-partitioned table using nested tables or XML object-relational data types, then the nested tables will be created as nonpartitioned tables.
- This clause is supported for tables containing `XMLType` columns only if the XML data is stored as binary XML.
- Interval partitioning is not supported at the subpartition level.

- Serializable transactions do not work with interval partitioning. Trying to insert data into a partition of an interval partitioned table that does not yet have a segment causes an error.
- In the `VALUES` clause:
  - You cannot specify `MAXVALUE` (an infinite upper bound), because doing so would defeat the purpose of the automatic addition of partitions as needed.
  - You cannot specify `NULL` values for the partitioning key column.

See Also:

Oracle Database VLDB and Partitioning Guide for more information on interval partitioning

PARTITION *partition*

If you specify a partition name, then the name *`partition`* must conform to the rules for naming schema objects and their part as described in "Database Object Naming Rules ". If you omit *`partition`*, then the database generates a name as described in "Notes on Partitioning in General".

range_values_clause

Specify the noninclusive upper bound for the current partition. The value list is an ordered list of literal values corresponding to the column list in the *`range_partitions`* clause. You can substitute the keyword `MAXVALUE` for any literal in the value list. `MAXVALUE` specifies a maximum value that will always sort higher than any other value, including null.

Specifying a value other than `MAXVALUE` for the highest partition bound imposes an implicit integrity constraint on the table.

Note:

If *`table`* 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 the `YYYY` 4-character format mask for the year. The `RRRR` format mask is not supported in this clause. 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:

Oracle Database Concepts for more information about partition bounds and "Range Partitioning Example"

table_partition_description

Use the *`table_partition_description`* to define the physical and storage characteristics of the table.

The clauses *deferred_segment_creation*, *segment_attributes_clause*, *table_compression*, *inmemory_clause*, and *ilm_clause* have the same function as described for the *physical_properties* of the table as a whole.

Use the *indexing_clause* to set the indexing property for a range, list, system, or reference table partition. Refer to the *indexing_clause* for more information.

The *prefix_compression* clause and OVERFLOW clause, have the same function as described for the *index_org_table_clause*.

LOB_storage_clause

The *LOB_storage_clause* lets you specify LOB storage characteristics for one or more LOB items in this partition or in any range or list subpartitions of this partition. If you do not specify the *LOB_storage_clause* for a LOB item, then the database generates a name for each LOB data partition as described in "Notes on Partitioning in General".

varray_col_properties

The *varray_col_properties* let you specify storage characteristics for one or more varray items in this partition or in any range or list subpartitions of this partition.

nested_table_col_properties

The *nested_table_col_properties* let you specify storage characteristics for one or more nested table storage table items in this partition or in any range or list subpartitions of this partition. Storage characteristics specified in this clause override any storage attributes specified at the table level.

partitioning_storage_clause

Use the *partitioning_storage_clause* to specify storage characteristics for hash partitions and for range, hash, and list subpartitions.

Restrictions on *partitioning_storage_clause*

This clause is subject to the following restrictions:

- The TABLESPACE SET clause is valid only when creating a sharded table by specifying the SHARDED keyword of CREATE TABLE. Use this clause to specify the tablespace set in which table partition data is to be stored.
- The OVERFLOW clause is relevant only for index-organized partitioned tables and is valid only within the *individual_hash_partitions* clause. It is not valid for range or hash partitions or for subpartitions of any type.
- You cannot specify the *advanced_index_compression* clause of the *index_compression* clause.
- You can specify the *prefix_compression* clause of the *indexing_clause* only for partitions of index-organized tables and you can specify COMPRESS or NOCOMPRESS, but you cannot specify the prefix length with *integer*.

list_partitions

Use the *list_partitions* clause to partition the table on a list of literal values for each column in the *column* list. List partitioning is useful for controlling how individual rows map to specific partitions.

Restrictions on List Partitioning

List partitioning is subject to the restrictions listed in "Restrictions on Partitioning in General". The following additional restrictions apply:

- You cannot specify more than 16 partitioning key columns.
- You cannot specify more than one partitioning key column when partitioning an index-organized table.
- The partitioning key columns must be of type `CHAR`, `NCHAR`, `VARCHAR2`, `NVARCHAR2`, `VARCHAR`, `NUMBER`, `FLOAT`, `DATE`, `TIMESTAMP`, `TIMESTAMP WITH LOCAL TIMEZONE`, or `RAW`.
- Each list partitioning key column with a character data type that belongs to an `XMLType` table or a table with an `XMLType` column, or that is used as a sharding key column must have one of the `BINARY` following declared collations: `BINARY` , `USING_NLS_COMP`, `USING_NLS_SORT`, or `USING_NLS_SORT_CS` . For all these collations, partitions are matched using the collation BINARY.

AUTOMATIC

Specify `AUTOMATIC` to create an automatic list-partitioned table. This type of table enables the database to create additional partitions on demand.

When you create an automatic list-partitioned table, you specify partitions and partitioning key values just as you would when creating a regular list-partitioned table. However, you do not specify a `DEFAULT` partition. As data is loaded into the table, the database automatically creates a new partition when the loaded partitioning key values do not correspond to any of the existing partitions. If list partitioning is defined with a single partitioning key value, then the database creates a new partition for each new partitioning key value. If list partitioning is defined with multiple partitioning key columns, then the database creates a new partition for each new and unique set of partitioning key values. For each automatically created partition, the database generates a name of the form `SYS_P`*`n`*. The database guarantees that automatically generated partition names are unique and do not violate namespace rules.

You can specify the `AUTOMATIC` keyword for list-partitioned tables, and list-range, list-list, list-hash, and list-interval composite-partitioned tables. For composite-partitioned tables, each automatically created list partition will have one subpartition, unless a subpartition template is defined for the table.

If a local partitioned index is defined on an automatic list-partitioned table, then local index partitions will be created when the corresponding table partitions are created.

Restrictions on Automatic List Partitioning

Automatic list partitioning is subject to the restrictions listed in "Restrictions on List Partitioning". The following additional restrictions apply:

- An automatic list-partitioned table must have at least one partition when created. Because new partitions are automatically created for new, and unknown, partitioning key values, an automatic list-partitioned table cannot have a `DEFAULT` partition.
- Automatic list partitioning is not supported for index-organized tables or external tables.
- Automatic list partitioning is not supported for tables containing varray columns.
- You cannot create a local domain index on an automatic list-partitioned table. You can create a global domain index on an automatic list-partitioned table.
- An automatic list-partitioned table cannot be a child table or a parent table for reference partitioning.
- Automatic list partitioning is not supported at the subpartition level.

STORE IN

The optional `STORE IN` clause lets you specify one or more tablespaces into which the database will store data for the automatically created list partitions.

> **Note:**
>
> You can change an automatic list-partitioned table to a regular list-partitioned table, and vice versa. You can also change the tablespaces into which the database will store data for automatically created list partitions. See the clause *alter_automatic_partitioning* of `ALTER TABLE` for more information.

list_values_clause

The *`list_values_clause`* of each partition must have at least one value. If the table is partitioned on one key column, then use the upper branch of the *`list_values`* syntax to specify a list of values for that column. In this case, no value, including NULL, can appear in more than one partition. If the table is partitioned on multiple key columns, then use the lower branch of the *`list_values`* syntax to specify a list of value lists. Each value list is enclosed in parentheses and represents a list of values for the key columns. In this case, individual key column values can appear in more than one partition; however, no complete value list can appear in more than one partition. List partitions are not ordered.

If you specify the literal `NULL` for a partition value in the `VALUES` clause, then to access data in that partition in subsequent queries, you must use an `IS NULL` condition in the `WHERE` clause, rather than a comparison condition.

The `DEFAULT` keyword creates a partition into which the database will insert any row that does not map to another partition. Therefore, you can specify `DEFAULT` for only one partition, and you cannot specify any other values for that partition. Further, the default partition must be the last partition you define. The use of `DEFAULT` is similar to the use of `MAXVALUE` for range partitions.

The string comprising the list of values for each partition can be up to 4K bytes. The total number of values for all partitions cannot exceed 64K-1.

The partitioning key column for a list partition can be an extended data type column, which has a maximum size of 32767 bytes. In this case, the list of values that you want to specify for a partition may exceed the 4K byte limit. You can work around this limitation by using one of the following methods:

- Use the `DEFAULT` partition for values that exceed the 4K byte limit.
- Use a hash function, such as `STANDARD_HASH`, in the partition key column to create unique identifiers of lengths less than 4K bytes. See STANDARD_HASH for more information.

Restriction on the *list_values_clause*

You cannot specify a `DEFAULT` partition for an automatic list-partitioned table.

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

table_partition_description

The subclauses of the `table_partition_description` have the same behavior as described for range partitions in *table_partition_description*.

hash_partitions

Use the `hash_partitions` clause to specify that the table is to be partitioned using the hash method. Oracle Database assigns rows to partitions using a hash function on values found in columns designated as the partitioning key. You can specify individual hash partitions, or you can specify how many hash partitions the database should create.

Restrictions on Hash Partitioning

Hash partitioning is subject to the restrictions listed in "Restrictions on Partitioning in General". The following additional restrictions apply:

- You cannot specify more than 16 partitioning key columns.
- Partitioning key columns must be of type `CHAR`, `NCHAR`, `VARCHAR2`, `NVARCHAR2`, `VARCHAR`, `NUMBER`, `FLOAT`, `DATE`, `TIMESTAMP`, `TIMESTAMP WITH LOCAL TIMEZONE`, or `RAW`.
- Each hash partitioning key column with a character data type that belongs to an `XMLType` table or a table with an `XMLType` column, or that is used as a sharding key column must have one of the following declared collations: `BINARY`, `USING_NLS_COMP`, `USING_NLS_SORT`, or `USING_NLS_SORT_CS`.

column

Specify an ordered list of columns used to determine into which partition a row belongs (the partitioning key).

individual_hash_partitions

Use this clause to specify individual partitions by name.

Use the `indexing_clause` to set the indexing property for a hash partition. Refer to the *indexing_clause* for more information.

Restriction on Specifying Individual Hash Partitions

The only clauses you can specify in the `partitioning_storage_clause` are the `TABLESPACE` clause and table 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. Refer to *Oracle Database Globalization Support Guide* for more information on character set support.

hash_partitions_by_quantity

An alternative to defining individual partitions is to specify the number of hash partitions. In this case, the database assigns partition names of the form `SYS_Pn`. The `STORE IN` clause lets you specify one or more tablespaces where the hash partition data is to be stored. The number of tablespaces need not equal the number of partitions. If the number of partitions is greater than the number of tablespaces, then the database cycles through the names of the tablespaces.

For both methods of hash partitioning, for optimal load balancing you should specify a number of partitions that is a power of 2. When you specify individual hash partitions, you can specify both `TABLESPACE` and table compression in the *`partitioning_storage_clause`*. When you specify hash partitions by quantity, you can specify only `TABLESPACE`. Hash partitions inherit all other attributes from table-level defaults.

The *`table_compression`* clause has the same function as described for the *table_properties* of the table as a whole.

The *`prefix_compression`* clause and the `OVERFLOW` clause have the same function as described for the *index_org_table_clause*.

Tablespace storage specified at the table level is overridden by tablespace storage specified at the partition level, which in turn is overridden by tablespace storage specified at the subpartition level.

In the *`individual_hash_partitions`* clause, the `TABLESPACE` clause of the *`partitioning_storage_clause`* determines tablespace storage only for the individual partition being created. In the *`hash_partitions_by_quantity`* clause, the `STORE IN` clause determines placement of partitions as the table is being created and the default storage location for subsequently added partitions.

Restriction on Specifying Hash Partitions by Quantity

You cannot specify the *`advanced_index_compression`* clause of the *`index_compression`* clause.

See Also:

Oracle Database VLDB and Partitioning Guide for more information on hash partitioning

composite_range_partitions

Use the *`composite_range_partitions`* clause to first partition *`table`* by range, and then partition the partitions further into range, hash, or list subpartitions.

The `INTERVAL` clause has the same semantics for composite range partitioning that it has for range partitioning. Refer to "INTERVAL Clause" for more information.

Specify *subpartition_by_range*, *subpartition_by_hash* or *subpartition_by_list* to indicate the type of subpartitioning you want for each composite range partition. Within these clauses you can specify a subpartition template, which establishes default subpartition characteristics for subpartitions created as part of this statement or subsequently created subpartitions.

After establishing the type of subpartitioning you want for the table, and optionally a subpartition template, you must define at least one range partition.

- You must specify the *range_values_clause* , which has the same requirements as for noncomposite range partitions.
- Use the *table_partition_description* to define the physical and storage characteristics of the each partition.

- In the *range_partition_desc*, use *range_subpartition_desc*, *list_subpartition_desc*, *individual_hash_subparts*, or *hash_subparts_by_quantity* to specify characteristics for the individual subpartitions of the partition. The values you specify in these clauses supersede for these subpartitions any values you have specified in the *subpartition_template*.
- The only characteristics you can specify for a hash or list subpartition or any LOB subpartition are TABLESPACE and *table_compression*.

Restrictions on Composite Range Partitioning

Regardless of the type of subpartitioning, composite range partitioning is subject to the following restrictions:

- The only physical attributes you can specify at the subpartition level are TABLESPACE and table compression.
- You cannot specify composite partitioning for an index-organized table. Therefore, the OVERFLOW clause of the *table_partition_description* is not valid for composite-partitioned tables.
- You cannot specify composite partitioning for tables containing XMLType columns.
- Each range, list, or hash subpartitioning key column with a character data type that belongs to an XMLType table or a table with an XMLType column must have one of the following declared collations: BINARY, USING_NLS_COMP, USING_NLS_SORT, or USING_NLS_SORT_CS.

See Also:

"Composite-Partitioned Table Examples" for examples of composite range partitioning and *Oracle Database VLDB and Partitioning Guide* for examples of composite list partitioning

composite_list_partitions

Use the *composite_list_partitions* clause to first partition *table* by list, and then partition the partitions further into range, hash, or list subpartitions.

Specify *subpartition_by_range*, *subpartition_by_hash* or *subpartition_by_list* to indicate the type of subpartitioning you want for each composite list partition. Within these clauses you can specify a subpartition template, which establishes default subpartition characteristics for subpartitions created as part of this statement and for subsequently created subpartitions.

After establishing the type of subpartitioning you want for each composite partition, and optionally defining a subpartition template, you must define at least one list partition.

- In the *list_partition_desc*, you must specify the *list_values_clause*, which has the same requirements as for noncomposite list partitions.
- Use the *table_partition_description* to define the physical and storage characteristics of the each partition.
- In the *list_partition_desc*, use *range_subpartition_desc*, *list_subpartition_desc*, *individual_hash_subparts*, or *hash_subparts_by_quantity* to specify characteristics for the individual subpartitions of the partition. The values you specify in these clauses supersede the for these subpartitions any values you have specified in the *subpartition_template*.

Specify `AUTOMATIC` to create an automatic list-range, list-list, list-hash, or list-interval composite-partitioned table. This type of table enables the database to create additional partitions on demand. The optional `STORE IN` clause lets you specify one or more tablespaces into which the database will store data for the automatically created partitions. The `AUTOMATIC` and `STORE IN` clauses have the same semantics here as they have for noncomposite list partitions. Refer to AUTOMATIC and STORE IN in the documentation on *`list_partitions`* for the full semantics of these clauses. Automatic composite-partitioned tables are subject to the restrictions listed in Restrictions on Composite List Partitioning and Restrictions on Automatic List Partitioning.

Restrictions on Composite List Partitioning

Composite list partitioning is subject to the same restrictions as described in "Restrictions on Composite Range Partitioning".

composite_hash_partitions

Use the *`composite_hash_partitions`* clause to first partition *`table`* using the hash method, and then partition the partitions further into range, hash, or list subpartitions.

Specify *subpartition_by_range*, *subpartition_by_hash* or *subpartition_by_list* to indicate the type of subpartitioning you want for each composite range partition. Within these clauses you can specify a subpartition template, which establishes default subpartition characteristics for subpartitions created as part of this statement or subsequently created subpartitions.

After establishing the type of subpartitioning you want for the table, you must specify *individual_hash_partitions* or *hash_partitions_by_quantity*.

Restrictions on Composite Hash Partitioning

Composite hash partitioning is subject to the same restrictions as described in "Restrictions on Composite Range Partitioning".

subpartition_template

The *`subpartition_template`* is an optional element of range, list, and hash subpartitioning. The template lets you define default subpartitions for each table partition. Oracle Database will create these default subpartition characteristics in any partition for which you do not explicitly define subpartitions. This clause is useful for creating symmetric partitions. You can override this clause by explicitly defining subpartitions at the partition level, in the *`range_subpartition_desc`*, *`list_subpartition_desc`*, *`individual_hash_subparts`*, or *`hash_subparts_by_quantity`* clause.

When defining subpartitions with a template, you can explicitly define range, list, or hash subpartitions, or you can define a quantity of hash subpartitions.

- To explicitly define subpartitions, use *`range_subpartition_desc`*, *`list_subpartition_desc`*, or *`individual_hash_subparts`*. You must specify a name for each subpartition. If you specify the *`LOB_partitioning_clause`* of the *`partitioning_storage_clause`*, then you must specify *`LOB_segname`*.
- To define a quantity of hash subpartitions, specify a positive integer for *`hash_subpartition_quantity`*. The database creates that number of subpartitions in each partition and assigns subpartition names of the form `SYS_SUBP`*`n`*.

Note:

When you specify tablespace storage for the subpartition template, it does not override any tablespace storage you have specified explicitly for the partitions of *table*. To specify tablespace storage for subpartitions, do one of these things:

- Omit tablespace storage at the partition level and specify tablespace storage in the subpartition template.
- Define individual subpartitions with specific tablespace storage.

Restrictions on Subpartition Templates

Subpartition templates are subject to the following restrictions:

- If you specify TABLESPACE for one LOB subpartition, then you must specify TABLESPACE for all of the LOB subpartitions of that LOB column. You can specify the same tablespace for more than one LOB subpartition.
- If you specify separate LOB storage for list subpartitions using the *partitioning_storage_clause*, either in the *subpartition_template* or when defining individual subpartitions, then you must specify *LOB_segname* for both LOB and varray columns.

subpartition_by_range

Use the *subpartition_by_range* clause to indicate that the database should subpartition by range each partition in *table*. The subpartitioning column list is unrelated to the partitioning key but is subject to the same restrictions (see column).

You can use the *subpartition_template* to specify default subpartition characteristic values. See *subpartition_template*. The database uses these values for any subpartition in this partition for which you do not explicitly specify the characteristic.

You can also define range subpartitions individually for each partition using the *range_subpartition_desc* of *range_partition_desc* or *list_partition_desc*. If you omit both *subpartition_template* and the *range_subpartition_desc*, then the database creates a single MAXVALUE subpartition.

subpartition_by_list

Use the *subpartition_by_list* clause to indicate that the database should subpartition each partition in the table on lists of literal values from the *column* list. You can specify a maximum of 16 list subpartitioning key columns.

You can use the *subpartition_template* to specify default subpartition characteristic values. See *subpartition_template*. The database uses these values for any subpartition in this partition for which you do not explicitly specify the characteristic.

You can also define list subpartitions individually for each partition using the *list_subpartition_desc* of *range_partition_desc* or *list_partition_desc*. If you omit both *subpartition_template* and the *list_subpartition_desc*, then the database creates a single DEFAULT subpartition.

Restrictions on List Subpartitioning

List subpartitioning is subject to the same restrictions as described in Restrictions on Composite Range Partitioning.

subpartition_by_hash

Use the `subpartition_by_hash` clause to indicate that the database should subpartition by hash each partition in `table`. The subpartitioning column list is unrelated to the partitioning key but is subject to the same restrictions (see column).

You can define the subpartitions using the `subpartition_template` or the `SUBPARTITIONS integer` clause. See *subpartition_template*. In either case, for optimal load balancing you should specify a number of partitions that is a power of 2.

If you specify `SUBPARTITIONS integer`, then you determine the default number of subpartitions in each partition of `table`, and optionally one or more tablespaces in which they are to be stored. The default value is 1. If you omit both this clause and `subpartition_template`, then the database will create each partition with one hash subpartition.

Notes on Composite Partitions

The following notes apply to composite partitions:

- For all subpartitions, you can use the `range_subpartition_desc`, `list_subpartition_desc`, `individual_hash_subparts`, or `hash_subparts_by_quantity` to specify individual subpartitions by name, and optionally some other characteristics.
- Alternatively, for hash and list subpartitions:
  - You can specify the number of subpartitions and optionally one or more tablespaces where they are to be stored. In this case, Oracle Database assigns subpartition names of the form `SYS_SUBPn`.
  - If you omit the subpartition description and if you have created a subpartition template, then the database uses the template to create subpartitions. If you have not created a subpartition template, then the database creates one hash subpartition or one `DEFAULT` list subpartition.
- For all types of subpartitions, if you omit the subpartitions description entirely, then the database assigns subpartition names as follows:
  - If you have specified a subpartition template *and* you have specified partition names, then the database generates subpartition names of the form `partition_name` underscore (_) `subpartition_name` (for example, `P1_SUB1`).
  - If you have not specified a subpartition template *or* if you have specified a subpartition template but did not specify partition names, then the database generates subpartition names of the form `SYS_SUBPn`.

reference_partitioning

Use this clause to partition the table by reference. Partitioning by reference is a method of equipartitioning the table being created (the **child table**) by a referential constraint to an existing partitioned table (the **parent table**). When you partition a table by reference, partition maintenance operations subsequently performed on the parent table automatically cascade to the child table. Therefore, you cannot perform partition maintenance operations on a reference-partitioned table directly.

If the parent table is an interval-partitioned table, then partitions in the reference-partitioned child table that correspond to interval partitions in the parent table will be created during inserts into the child table. When an interval partition in a child table is created, the partition name is inherited from the associated parent table partition. If the child table has a table-level default tablespace, then it will be used as the tablespace for the new interval partition. Otherwise, the tablespace will be inherited from the parent table partition. Refer to *Oracle Database VLDB and Partitioning Guide* for more information on referencing an interval-partitioned table.

constraint

The partitioning referential constraint must meet the following conditions:

- You must specify a referential integrity constraint defined on the table being created, which must refer to a primary key or unique constraint on the parent table. The constraint must be in `ENABLE VALIDATE NOT DEFERRABLE` state, which is the default when you specify a referential integrity constraint during table creation.
- All foreign key columns referenced in the constraint must be `NOT NULL`.
- When you specify the constraint, you cannot specify the `ON DELETE SET NULL` clause of the *`references_clause`*.
- The parent table referenced in the constraint must be an existing partitioned table. It can be partitioned by any method.
- The foreign and parent keys cannot contain any virtual (expression) columns that reference PL/SQL functions or LOB columns.

reference_partition_desc

Use this optional clause to specify partition names and to define the physical and storage characteristics of the partition. The subclauses of the *`table_partition_description`* have the same behavior as described for range partitions in *table_partition_description*.

If you specify this clause when creating a reference-partitioned child table whose parent is an interval-partitioned table, then the partition descriptors are used for the child table's non-interval partitions. Partition descriptors cannot be specified for interval partitions.

Restrictions on Reference Partitioning

Reference partitioning is subject to the restrictions listed in Restrictions on Partitioning in General. The following additional restrictions apply:

- Restrictions for reference partitioning are derived from the partitioning strategy of the parent table.
- Neither the parent table nor the child table can be an automatic list-partitioned table.
- You cannot specify this clause for an index-organized table, an external table, or a domain index storage table.
- The parent table can be partitioned by reference, but *`constraint`* cannot be self-referential. The table being created cannot be partitioned based on a reference to itself.
- If `ROW MOVEMENT` is enabled for the parent table, it must also be enabled for the child table.

See Also:

Oracle Database VLDB and Partitioning Guide for more information on partitioning by reference and "Reference Partitioning Example"

system_partitioning

Use this clause to create system partitions. System partitioning does not entail any partitioning key columns, nor do system partitions have any range or list bounds or hash algorithms. Rather, they provide a way to equipartition dependent tables such as nested table or domain index storage tables with partitioned base tables.

- If you specify only `PARTITION BY SYSTEM`, then the database creates one partition with a system-generated name of the form `SYS_Pn`.
- If you specify `PARTITION BY SYSTEM PARTITIONS` *`integer`*, then the database creates as many partitions as you specify in *`integer`*, which can range from 1 to 1024K-1.
- The description of the partition takes the same syntax as reference partitions, so they share the *`reference_partition_desc`*. You can specify additional partition attributes with the *`reference_partition_desc`* syntax. However, within the *`table_partition_description`*, you cannot specify the `OVERFLOW` clause.

Restrictions on System Partitioning

System partitioning is subject to the following restrictions:

- You cannot system partition an index-organized table or a table that is part of a cluster.
- Composite partitioning is not supported with system partitioning.
- You cannot split a system partition.
- You cannot specify system partitioning in a `CREATE TABLE` ... `AS SELECT` statement.
- To insert data into a system-partitioned table using an `INSERT INTO` ... `AS` *`subquery`* statement, you must use partition-extended syntax to specify the partition into which the values returned by the subquery will be inserted.

See Also:

Refer to *Oracle Database Data Cartridge Developer's Guide* for information on the uses for system partitioning and "References to Partitioned Tables and Indexes "

consistent_hash_partitions

This clause is valid only for sharded tables. Use this clause to create consistent hash partitions.

Each sharding 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`.

directory_based_partitions

Use this clause to create a sharded table with directory-based sharding.

The restrictions that apply to sharded tables apply to tables sharded by directory.

Example: Create Sharded Table and Partition by Directory

```
CREATE SHARDED TABLE departments
  ( department_id  NUMBER(6)
  , department_name VARCHAR2(30) CONSTRAINT dept_name_nn NOT NULL
  , manager_id    NUMBER(6)
  , location_id   NUMBER(4)
  , CONSTRAINT dept_id_pk PRIMARY KEY(department_id)
  )
  PARTITION BY DIRECTORY (department_id)
  (
    PARTITION p_1 TABLESPACE tbs1,
    PARTITION p_2 TABLESPACE tbs2
  );
```

In the example above, when you partition by directory the partition names `p_1` and `p_2` do not a have value list after them. This is different from partition by list.

consistent_hash_with_subpartitions

This clause is valid only for sharded tables. Use this clause to create consistent hash with subpartitions.

Each sharding 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`.

range_partitionset_clause

Use this clause to create a range partition set.

In the `SUBPARTITION BY` clause, within the *`subpartition_template`* clause, you cannot specify a tablespace for a subpartition. That is, for range, list, and individual hash subpartitions, you cannot specify the `TABLESPACE` clause of the *`partitioning_storage_clause`*, and in the *`hash_subpartitions_by_quantity`* clause, you cannot specify the `STORE IN (`*`tablespace`*`)` clause.

In the `PARTITIONS AUTO` clause, within the *`subpartition_template`* clause of the *`range_partitionset_desc`* clause, you *can* specify a tablespace for a subpartition.

Each super sharding or sharding 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`.

list_partitionset_clause

Use this clause to create a list partition set.

In the `SUBPARTITION BY` clause, within the *`subpartition_template`* clause, you cannot specify a tablespace for a subpartition. That is, for range, list, and individual hash subpartitions, you cannot specify the `TABLESPACE` clause of the *`partitioning_storage_clause`*, and in the *`hash_subpartitions_by_quantity`* clause, you cannot specify the `STORE IN (`*`tablespace`*`)` clause.

In the `PARTITIONS AUTO` clause, within the *`subpartition_template`* clause of the *`list_partitionset_desc`* clause, you *can* specify a tablespace for a subpartition.

Each super sharding or sharding 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`.

attribute_clustering_clause

Use this clause to enable the table for attribute clustering. Attribute clustering lets you cluster data in close physical proximity based on the content of specified columns.

Attribute clustering can be based only on columns in *`table`* or on joined values from other tables. The latter is called join attribute clustering.

> **See Also:**
>
> *Oracle Database Data Warehousing Guide* for more information on attribute clustering

clustering_join

Use this clause to specify join attribute clustering. Use the `JOIN` clause to specify the joined values from other tables on which to base the attribute clustering. You can specify a maximum of four `JOIN` clauses.

cluster_clause

Use this clause to specify the type of ordering to use for the table: linear ordering or interleaved ordering. If you do not specify the `LINEAR` or `INTERLEAVED` keyword, then the default is `LINEAR`.

BY LINEAR ORDER

Use this clause to specify linear ordering. This type of ordering stores data according to the order of the specified columns. If you specify this clause, then you can specify only one clustering column group, which can contain at most 10 columns.

BY INTERLEAVED ORDER

Use this clause to specify interleaved ordering. This type of ordering uses a special multidimensional clustering technique, similar to z-ordering, that permits multicolumn clustering. If you specify this clause, then you can specify at most four clustering column groups, with a maximum of 40 columns across all groups.

clustering_columns

Use this clause to specify one or more clustering column groups.

clustering_column_group

Use this clause to specify one or more columns to be included in the clustering column group.

Restriction on Attribute Clustering Columns

Each character column in the clustering column group must have one of the following declared collations: `BINARY` or `USING_NLS_COMP`.

clustering_when

Use these clauses to allow or disallow attribute clustering during direct-path insert operations and data movement operations.

ON LOAD

Specify `YES ON LOAD` to allow, or `NO ON LOAD` to disallow, attribute clustering during direct-path inserts (serial or parallel) resulting either from an `INSERT` or a `MERGE` operation.

The default is `YES ON LOAD`.

ON DATA MOVEMENT

Specify `YES ON DATA MOVEMENT` to allow, or `NO ON DATA MOVEMENT` to disallow, attribute clustering for data movement that occurs during the following operations:

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

The default is `YES ON DATA MOVEMENT`.

zonemap_clause

Use this clause to create a zone map on the table. The zone map tracks the columns specified in the *`clustering_columns`* clause.

- Specify `WITH MATERIALIZED ZONEMAP` to create a zone map. For *`zonemap_name`*, specify the name of the zone map to be created. If you omit *`zonemap_name`*, then the name of the zone map is `ZMAP$_`*`table`*.
- Specify `WITHOUT MATERIALIZED ZONEMAP` to not create a zone map. This is the default.

If you subsequently drop the table or use the `ALTER TABLE` statement to `DROP CLUSTERING` or `MODIFY CLUSTERING` ... `WITHOUT MATERIALIZED ZONEMAP`, then the zone map will be dropped.

See Also:

CREATE MATERIALIZED ZONEMAP for more information on zone maps

Restrictions on Attribute Clustering

The following restrictions apply to attribute clustering:

- Attribute clustering is not supported for temporary tables or external tables.
- The table being created must be a heap-organized table. However, tables specified in the *`clustering_join`* clause can be heap-organized or index-organized tables.
- Clustering columns must be of a scalar data type and cannot be encrypted.
- If you specify `BY LINEAR ORDER`, then you can specify only one clustering column group, which can contain at most 10 columns.
- If you specify `BY INTERLEAVED ORDER`, then you can specify at most four clustering column groups, with a maximum of 40 columns across all groups.
- For join attribute clustering:
  - The number of dimension tables cannot exceed four.
  - The join to the table or tables providing the attribute clustering columns must be on a unique key or primary key column to avoid row duplication.
- Attribute clustering will not order rows that are inserted using `MERGE` statements or multitable insert operations.

CACHE | NOCACHE | CACHE READS

Use these clauses to indicate how Oracle Database should store blocks in the buffer cache. For LOB storage, you can specify `CACHE`, `NOCACHE`, or `CACHE READS`. For other types of storage, you can specify only `CACHE` or `NOCACHE`.

If you omit these clauses, then:

- In a `CREATE TABLE` statement, `NOCACHE` is the default.
- In an `ALTER TABLE` statement, the existing value is not changed.

The behavior of `CACHE` and `NOCACHE` described in this section does not apply when Oracle Database chooses to use direct reads or to perform table scans using parallel query.

CACHE

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

> **See Also:**
>
> *Oracle Database Concepts* for more information on how the database maintains the least recently used (LRU) list

As a parameter in the *`LOB_storage_clause`*, `CACHE` specifies that the database places LOB data values in the buffer cache for faster access. The database evaluates this parameter in conjunction with the *`logging_clause`*. If you omit this clause, then the default value for both BasicFiles and SecureFiles LOBs is `NOCACHE LOGGING`.

Restriction on CACHE

You cannot specify `CACHE` for an index-organized table. However, index-organized tables implicitly provide `CACHE` behavior.

NOCACHE

For data that is not accessed frequently, this clause indicates that the blocks retrieved for this table are placed at the least recently used end of the LRU list in the buffer cache when a full table scan is performed. `NOCACHE` is the default for LOB storage.

As a parameter in the *`LOB_storage_clause`*, `NOCACHE` specifies that the LOB values are not brought into the buffer cache. `NOCACHE` is the default for LOB storage.

Restriction on NOCACHE

You cannot specify `NOCACHE` for an index-organized table.

CACHE READS

`CACHE READS` applies only to LOB storage. It specifies that LOB values are brought into the buffer cache only during read operations but not during write operations.

logging_clause

Use this clause to indicate whether the storage of data blocks should be logged or not.

> **See Also:**
>
> *logging_clause* for a description of the *`logging_clause`* when specified as part of *`LOB_parameters`*

result_cache_clause

Use this clause to determine whether the results of statements or query blocks that name this table are considered for storage in the result cache.

You can use mode `DEFAULT` or mode `FORCE` for result caching, with `STANDBY` enabled or disabled.

- `DEFAULT`: Result caching is not determined at the table level. The query is considered for result caching if the `RESULT_CACHE_MODE` initialization parameter is set to `FORCE`, or if that parameter is set to `MANUAL` and the `RESULT_CACHE` hint is specified in the query. This is the default if you omit this clause.
- `FORCE`: If all tables names in the query have this setting, then the query is always considered for caching unless the `NO_RESULT_CACHE` hint is specified for the query. If one or more tables named in the query are set to `DEFAULT`, then the effective table annotation for that query is considered to be `DEFAULT`, with the semantics described above.
- The default value of `STANDBY` is `DISABLE`.
- You must enable `STANDBY` on all the dependent objects of a query to save the result of the query into the result cache.
- A transaction must enable `STANDBY` on an object in order to generate a redo marker at transaction commit time on the primary.

You can query the `RESULT_CACHE` column of the `DBA_`, `ALL_`, and `USER_TABLES` data dictionary views to learn the result cache mode of the table.

Precedence

The `RESULT_CACHE` and `NO_RESULT_CACHE` SQL hints take precedence over these result cache table annotations and the `RESULT_CACHE_MODE` initialization parameter.

The `RESULT_CACHE_MODE` setting of `FORCE` in turn takes precedence over this table annotation clause.

If you set the initialization parameter `RESULT_CACHE_INTEGRITY` to `ENFORCED`, then only deterministic constructs will be eligible for result caching. The `ENFORCED` setting overrides the setting of `RESULT_CACHE_MODE` or specified hints.

If you set `RESULT_CACHE_INTEGRITY` to `TRUSTED`, then the database honors the setting of `RESULT_CACHE_MODE` and specified hints and considers queries using non-deterministic constructs as candidates for result caching.

Note:

The `RESULT_CACHE_MODE` setting of `FORCE` is not recommended, as it can cause significant performance and latching overhead, as database and clients will try to cache all queries.

See Also:

- *Oracle Call Interface Programmer's Guide* and *Oracle Database Concepts* for general information about result caching
- *Oracle Database Performance Tuning Guide* for information about using this clause
- *Oracle Database Reference* for information about the `RESULT_CACHE_MODE` initialization parameter and the `*_TABLES` data dictionary views
- "RESULT_CACHE Hint " and "NO_RESULT_CACHE Hint " for information about the hints

parallel_clause

The *`parallel_clause`* lets you parallelize creation of the table and set the default degree of parallelism for queries and the DML `INSERT`, `UPDATE`, `DELETE`, and `MERGE` after table creation.

Note:

The syntax of the *`parallel_clause`* supersedes syntax appearing in earlier releases of Oracle. The superseded syntax is still supported for backward compatibility, but may result in slightly different behavior from that documented.

NOPARALLEL

Specify `NOPARALLEL` for serial execution. This is the default.

PARALLEL

Specify `PARALLEL` if you want Oracle to select a degree of parallelism equal to the number of CPUs available on all participating instances times the value of the `PARALLEL_THREADS_PER_CPU` initialization parameter.

PARALLEL *integer*

Specification of *`integer`* indicates the **degree of parallelism**, which is the number of parallel threads used in the parallel operation. Each parallel thread may use one or two parallel execution servers. Normally Oracle calculates the optimum degree of parallelism, so it is not necessary for you to specify *`integer`*.

See Also:

parallel_clause for more information on this clause

NOROWDEPENDENCIES | ROWDEPENDENCIES

This clause lets you specify whether *`table`* will use **row-level dependency tracking**. With this feature, each row in the table has a system change number (SCN) that represents a time greater than or equal to the commit time of the last transaction that modified the row. You cannot change this setting after *table* is created.

ROWDEPENDENCIES

Specify ROWDEPENDENCIES if you want to enable row-level dependency tracking. This setting is useful primarily to allow for parallel propagation in replication environments. It increases the size of each row by 6 bytes.

Restriction on the ROWDEPENDENCIES Clause

Oracle does not support table compression for tables that use row-level dependency tracking. If you specify both the ROWDEPENDENCIES clause and the *table_compression* clause, then the *table_compression* clause is ignored. To remove the ROWDEPENDENCIES attribute, you must redefine the table using the DBMS_REDEFINITION package or recreate the table.

NOROWDEPENDENCIES

Specify NOROWDEPENDENCIES if you do not want *table* to use the row-level dependency tracking feature. This is the default.

enable_disable_clause

The *enable_disable_clause* lets you specify whether Oracle Database should apply a constraint. By default, constraints are created in ENABLE VALIDATE state.

Restrictions on Enabling and Disabling Constraints

Enabling and disabling constraints are subject to the following restrictions:

- To enable or disable any integrity constraint, you must have defined the constraint in this or a previous statement.
- You cannot enable a foreign key constraint unless the referenced unique or primary key constraint is already enabled.
- In the *index_properties* clause of the *using_index_clause*, the INDEXTYPE IS ... clause is not valid in the definition of a constraint.

See Also:

constraint for more information on constraints and "Creating a Table: ENABLE/ DISABLE Examples"

ENABLE Clause

Use this clause if you want the constraint to be applied to the data in the table. This clause is described fully in "ENABLE Clause" in the documentation on constraints.

DISABLE Clause

Use this clause if you want to disable the integrity constraint. This clause is described fully in "DISABLE Clause" in the documentation on constraints.

UNIQUE

The UNIQUE clause lets you enable or disable the unique constraint defined on the specified column or combination of columns.

PRIMARY KEY

The `PRIMARY KEY` clause lets you enable or disable the primary key constraint defined on the table.

CONSTRAINT

The `CONSTRAINT` clause lets you enable or disable the integrity constraint named *`constraint_name`*.

KEEP | DROP INDEX

This clause lets you either preserve or drop the index Oracle Database has been using to enforce a unique or primary key constraint.

Restriction on Preserving and Dropping Indexes

You can specify this clause only when disabling a unique or primary key constraint.

using_index_clause

The *`using_index_clause`* lets you specify an index for Oracle Database to use to enforce a unique or primary key constraint, or lets you instruct the database to create the index used to enforce the constraint.

See Also:

- CREATE INDEX for a description of *index_attributes*, the *global_partitioned_index* and *local_partitioned_index* clauses, `NOSORT`, and the *`logging_clause`* in relation to indexes
- *constraint* for information on the *`using_index_clause`* and on `PRIMARY KEY` and `UNIQUE` constraints
- "Explicit Index Control Example" for an example of using an index to enforce a constraint

CASCADE

Specify `CASCADE` to disable any integrity constraints that depend on the specified integrity constraint. To disable a primary or unique key that is part of a referential integrity constraint, you must specify this clause.

Restriction on CASCADE

You can specify `CASCADE` only if you have specified `DISABLE`.

row_movement_clause

The *`row_movement_clause`* lets you specify whether the database can move a table row. It is possible for a row to move, for example, during table compression or an update operation on partitioned data.

Note:

If you need static rowids for data access, then do not enable row movement. For a normal (heap-organized) table, moving a row changes the rowid of the row. For a moved row in an index-organized table, the logical rowid remains valid, although the physical guess component of the logical rowid becomes inaccurate.

- Specify `ENABLE` to allow the database to move a row, thus changing the rowid.
- Specify `DISABLE` if you want to prevent the database from moving a row, thus preventing a change of rowid.

If you omit this clause, then the database disables row movement.

Restriction on Row Movement

You cannot specify this clause for a nonpartitioned index-organized table.

logical_replication_clause

Use this clause to perform partial database replication for users such as Oracle GoldenGate, and reduce the supplemental logging overhead of uninteresting tables in interesting schema where supplemental logging is enabled.

ENABLE LOGICAL REPLICATION

You can specify `ENABLE LOGICAL REPLICATION` with `ALLKEYS`, `ALLOW NOVALIDATE KEYS`, and `[NO] PARTIAL JOSN` in any order. The supplemtental log settings of all levels (database, the container, schema, table) are honored. No additional ID or scheduling-key supplemental logging is added for this table.

DISABLE LOGICAL REPLICATION

When logical replication is disabled for a table, it means that only database level supplemental logging is honored. This provides a way for partial database replication users (who will not enable database level column data supplemental logging) to disable supplemental logging for uninteresting tables, so that even when supplemental logging is enabled at the schema level, there is no column data supplemental logging for uninteresting tables.

If you create a table with `DISABLE LOGICAL REPLICATION`, logical replication is disabled for the table. Database and container level supplemental log settings are honored but table-level and schema-level supplemental log settings are ignored.

ENABLE LOGICAL REPLICATION ALL KEYS

Use this option to enable the table for Golden Gate `AUTO_CAPTURE`.

ID and scheduling keys, primary key (`PK`), foreign key (`FK`), unique index (`UI`), and all key supplemental logging (`ALLKEYS`) are implicitly enabled for the table.

ENABLE LOGICAL REPLICATION ALLOW NOVALIDATE KEYS

Use this option to enable the table for Golden Gate `AUTO_CAPTURE`.

ID and scheduling keys, primary key (`PK`), foreign key (`FK`), unique index (`UI`), and all key supplemental logging (`ALLKEYS`) are implicitly enabled for the table. The primary key constraint in `NOVALIDATE` mode can be supplementally logged as a unique identifier for the table. `NOVALIDATE KEYS` are allowed as a row identification key.

If you create a table with `ENABLE LOGICAL REPLICATION ALLOW NOVALIDATE KEYS`, ID and scheduling-key is implicitly enabled for the table. The primary key constraint in `NOVALIDATE` mode can be supplementally logged as a unique identifier for the table.

NO PARTIAL JSON

Use this clause if your replication target database does not support native `JSON`, partial `JSON` or `JSON DIFF` . This includes non-Oracle target databases and Oracle target databases with DB compatible lower than 23.

You can specify `NO PARTIAL JSON` whether a column of type `JSON` exists or not.

When you enable column-level data supplemental logging on the table or schema, partial `JSON` update is disabled. This means that the `JSON` update will always generate a new `JSON` instance with full `JSON` document. This is the default when `[NO] PARTIAL JSON` is not specified.

In order to use `NO PARTIAL JSON` without errors, you must set the database compatible initialization parameter must be set to 23 or higher.

PARTIAL JSON

Use this option if your replication target database is Oracle Database with the database compatible initialization parameter set to 23 or higher and where partial `JSON` and `JSON DIFF` can be supported at the target database.

You can specify `PARTIAL JSON` whether a column of type `JSON` exists or not.

With `PARTIAL JSON` you can enable a partial `JSON` update for the table. A `JSON` column updated using `JSON_TRANSFORM` may internally partially update the existing `JSON` instance, even when column-level data supplemental logging has been added for the table or schema.

In order to use `PARTIAL JSON` without errors, you must set the database compatible initialization parameter must be set to 23 or higher.

flashback_archive_clause

You must have the `FLASHBACK ARCHIVE` object privilege on the specified flashback archive to specify this clause. Use this clause to enable or disable historical tracking for the table.

- Specify `FLASHBACK ARCHIVE` to enable tracking for the table. You can specify *`flashback_archive`* to designate a particular flashback archive for this table. The flashback archive you specify must already exist.

  If you omit *`flashback_archive`*, then the database uses the default flashback archive designated for the system. If no default flashback archive has been designated for the system, then you must specify *`flashback_archive`*.
- Specify `NO FLASHBACK ARCHIVE` to disable tracking for the table. This is the default.

Restrictions on *flashback_archive_clause*

Flashback data archives are subject to the following restrictions:

- You cannot specify this clause for a nested table, clustered table, temporary table, remote table, or external table.
- You cannot specify this clause for a table compressed with Hybrid Columnar Compression.
- The table for which you are specifying this clause cannot contain any `LONG` or nested table columns.

- If you specify this clause and subsequently copy the table to a different database—using the export and import utilities or the transportable tablespace feature—then the copied table will not be enabled for tracking and the archived data for the original table will not be available for the copied table.

See Also:

- *Oracle Database Development Guide* for general information on using Flashback Time Travel
- 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

ROW ARCHIVAL

Specify this clause to enable `table` for row archival. This clause lets you implement In-Database Archiving, which allows you to designate table rows as active or archived. You can then perform queries on only the active rows within the table.

When you specify this clause, a hidden column `ORA_ARCHIVE_STATE` is created in the table. The column is of data type `VARCHAR2`. You can specify a value of `0` or `1` for this column to indicate whether a row is active (`0`) or archived (`1`). If you do not specify a value for `ORA_ARCHIVE_STATE` when inserting data into the table, then the value is set to `0`.

- If `ROW ARCHIVE VISIBILITY = ACTIVE` for the session, then the database will consider only active rows when performing queries on the table.
- If `ROW ARCHIVE VISIBILITY = ALL` for the session, then the database will consider all rows when performing queries on the table.

See Also:

- The `ALTER SESSION` Semantics clause to learn how to configure row archival visibility for a session
- The `ALTER TABLE` [NO] ROW ARCHIVAL clause to learn how to enable or disable an existing table for row archival
- *Oracle Database VLDB and Partitioning Guide* for more information on In-Database Archiving

FOR EXCHANGE WITH TABLE

This clause lets you create a table that matches the structure of an existing partitioned table. The two tables are then eligible for exchanging partitions and subpartitions. For `table`, specify an existing partitioned table. For `schema`, specify the schema that contains the existing partitioned table. If you omit `schema`, then the database assumes the table is in your own schema.

This operation creates a metadata clone, without data, of the partitioned table. The clone has the same column ordering and column properties of the original table. Column properties copied to the clone during this operation include unusable columns, invisible columns, virtual expression columns, functional index expression columns, and other internal settings and attributes. Indexes on the existing partitioned table are not created on the clone table.

You can subsequently use the *exchange_partition_subpart* clause of ALTER TABLE to exchange partitions or subpartitions between the two tables. Refer to *exchange_partition_subpart* in the documentation on ALTER TABLE for more information.

Each super sharding or sharding 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.

Restrictions on FOR EXCHANGE WITH TABLE

The following restrictions apply to the FOR EXCHANGE WITH TABLE clause:

- If you specify this clause, then you cannot specify the *relational_properties* clause.
- If you specify this clause, then within the *table_properties* clause, you can specify only the *table_partitioning_clause*.
- Within the table_partitioning_clause each 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.
- When you create a clone for a partition of a composite-partitioned table, you must explicitly specifying the appropriate *table_partitioning_clause* that matches exactly the subpartitioning of the partition you want to exchange.
- Oracle does not clone the statistics setup of the partitioned table. For example, if you plan to perform an exchange with a partitioned table for which incremental statistics are enabled, you must manually enable the creation of a table synopsis on the clone table. See *Oracle Database SQL Tuning Guide* for more information on maintaining incremental statistics on partitioned tables.
- You cannot create a clone of an external table.
- If the table specified in the FOR EXCHANGE WITH TABLE clause is protected by a fine-grained audit (FGA) policy, then the CREATE TABLE statement will fail, if the user who is creating it is not the SYS user.
- You cannot use this clause if you have VPD policies enabled on the target table.

AS *subquery*

Specify a subquery to determine the contents of the table. The rows returned by the subquery are inserted into the table upon its creation.

For object tables, *subquery* can contain either one expression corresponding to the table type, or the number of top-level attributes of the table type. Refer to SELECT for more information.

If *subquery* returns the equivalent of part or all of an existing materialized view, then the database may rewrite the query to use the materialized view in place of one or more tables specified in *subquery*.

See Also:

Oracle Database Data Warehousing Guide for more information on materialized views and query rewrite

Oracle Database derives data types and lengths from the subquery. Oracle Database follows the following rules for integrity constraints and other column and table attributes:

- Oracle Database automatically defines on columns in the new table any `NOT NULL` constraints that have a state of `NOT DEFERRABLE` and `VALIDATE`, and were explicitly created on the corresponding columns of the selected table if the subquery selects the column rather than an expression containing the column. If any rows violate the constraint, then the database does not create the table and returns an error.
- `NOT NULL` constraints that were implicitly created by Oracle Database on columns of the selected table (for example, for primary keys) are not carried over to the new table.
- In addition, primary keys, unique keys, foreign keys, check constraints, partitioning criteria, indexes, and column default values are not carried over to the new table.
- If the selected table is partitioned, then you can choose whether the new table will be partitioned the same way, partitioned differently, or not partitioned. Partitioning is not carried over to the new table. Specify any desired partitioning as part of the `CREATE TABLE` statement before the `AS` *`subquery`* clause.
- A column that is encrypted using Transparent Data Encryption in the selected table will not be encrypted in the new table unless you define the column in the new table as encrypted at create time.

Note:

Oracle recommends that you encrypt sensitive columns before populating them with data. This will avoid creating clear text copies of sensitive data.

If each column returned by *`subquery`* has a column name or is an expression with a specified column alias, then you can omit the columns from the table definition entirely. In this case, the names of the columns of *`table`* are the same as the columns in *`subquery`*. The exception is creating an index-organized table, for which you must specify the columns in the table definition because you must specify a primary key column.

You can use *`subquery`* in combination with the `TO_LOB` function to convert the values in a `LONG` column in another table to LOB values in a column of the table you are creating.

See Also:

- *Oracle Database SecureFiles and Large Objects Developer's Guide* for a discussion of why and when to copy `LONG` data to a LOB
- "Conversion Functions " for a description of how to use the `TO_LOB` function
- SELECT for more information on the *`order_by_clause`*
- *Oracle Database SQL Tuning Guide* for information on statistics gathering when using the `AS` *`subquery`* clause

parallel_clause

If you specify the *`parallel_clause`* in this statement, then the database will ignore any value you specify for the `INITIAL` storage parameter and will instead use the value of the `NEXT` parameter.

See Also:

storage_clause for information on these parameters

ORDER BY

The ORDER BY clause lets you order rows returned by the subquery.

When specified with CREATE TABLE, this clause does not necessarily order data across the entire table. For example, it does not order across partitions. Specify this clause if you intend to create an index on the same key as the ORDER BY key column. Oracle Database will cluster data on the ORDER BY key so that it corresponds to the index key.

Restrictions on the Defining Query of a Table

The table query is subject to the following restrictions:

- The number of columns in the table must equal the number of expressions in the subquery.
- The column definitions can specify only column names, default values, and integrity constraints, not data types.
- You cannot define a foreign key constraint in a CREATE TABLE statement that contains AS *subquery* unless the table is reference partitioned and the constraint is the table's partitioning referential constraint. In all other cases, you must create the table without the constraint and then add it later with an ALTER TABLE statement.

FOR STAGING

Specify FOR STAGING to create a staging table optimized for staging data in Oracle Database. Staging tables are typically shortlived and volatile with constantly changing data.

You can create a staging table with or without partitions with CREATE TABLE t FOR STAGING or you can convert an exisiting table into a staging table with ALTER TABLE t FOR STAGING.

Examples: Create a Staging Table

```
CREATE TABLE staging_table (col1 number, col2 varchar2(100)) FOR STAGING;
```

Examples: Create a Staging Table With Partitions

```
CREATE TABLE part_staging_table (col1 number, col2 varchar2(100))
PARTITION BY RANGE (col1) (PARTITION p1 VALUES LESS THAN (100), PARTITION pmax VALUES
LESS THAN (MAXVALUE))
FOR STAGING;
```

A staging table with or without partitions has the following characteristics:

- Compression is explicitly turned off and disallowed for any future data load on the table and its partitions and subpartitions. Changing existing tables into staging tables will not impact the storage of existing data but only impact future data loads.
- You cannot change the default attritbutes of a staging table, its partitions or subpartitions, or future data loads using ALTER TABLE.
- You cannot perform any partition maintenance operations that will move the data and compress it using ALTER TABLE .
- You cannot partition a staging table and specify compression on any of its partitions.

- Dynamic sampling is used for queries on a staging table. This means that you cannot gather statistics on a staging table or any of its partitions.
- If you drop a staging table it will be dropped immediately, bypassing the recyclebin irrespective of your setting.

The dictionary views `USER_TABLES`, `ALL_TABLES`, and `DBA_TABLES` have a staging table property in a new column `STAGING`. The value of `STAGING` is `YES` for a staged table, and `NO` otherwise.

object_table

The `OF` clause lets you explicitly create an **object table** of type *`object_type`*. The columns of an object table correspond to the top-level attributes of type *`object_type`*. Each row will contain an object instance, and each instance will be assigned a unique, system-generated object identifier when a row is inserted. If you omit *`schema`*, then the database creates the object table in your own schema.

Object tables, as well as `XMLType` tables, object views, and `XMLType` views, do not have any column names specified for them. Therefore, Oracle defines a system-generated pseudocolumn `OBJECT_ID`. You can use this column name in queries and to create object views with the `WITH OBJECT IDENTIFIER` clause.

See Also:

"Object Column and Table Examples"

object_table_substitution

Use the *`object_table_substitution`* clause to specify whether row objects corresponding to subtypes can be inserted into this object table.

NOT SUBSTITUTABLE AT ALL LEVELS

`NOT SUBSTITUTABLE AT ALL LEVELS` indicates that the object table being created is not substitutable. In addition, substitution is disabled for all embedded object attributes and elements of embedded nested tables and arrays. The default is `SUBSTITUTABLE AT ALL LEVELS`.

See Also:

- CREATE TYPE for more information about creating object types
- "User-Defined Types ", "About User-Defined Functions ", "About SQL Expressions ", CREATE TYPE , and *Oracle Database Object-Relational Developer's Guide* for more information about using `REF` types

object_properties

The properties of object tables are essentially the same as those of relational tables. However, instead of specifying columns, you specify attributes of the object.

For *`attribute`*, specify the qualified column name of an item in an object.

oid_clause

The *oid_clause* lets you specify whether the object identifier of the object table should be system generated or should be based on the primary key of the table. The default is SYSTEM GENERATED.

Restrictions on the *oid_clause*

This clause is subject to the following restrictions:

- You cannot specify OBJECT IDENTIFIER IS PRIMARY KEY unless you have already specified a PRIMARY KEY constraint for the table.
- You cannot specify this clause for a nested table.

Note:

A primary key object identifier is locally unique but not necessarily globally unique. If you require a globally unique identifier, then you must ensure that the primary key is globally unique.

oid_index_clause

This clause is relevant only if you have specified the *oid_clause* as SYSTEM GENERATED. It specifies an index, and optionally its storage characteristics, on the hidden object identifier column.

For *index*, specify the name of the index on the hidden system-generated object identifier column. If you omit *index*, then the database generates a name.

physical_properties* and *table_properties

The semantics of these clauses are documented in the corresponding sections under relational tables. See *physical_properties* and *table_properties*.

XMLType_table

Use the *XMLType_table* syntax to create a table of data type XMLType. Most of the clauses used to create an XMLType table have the same semantics that exist for object tables. The clauses specific to XMLType tables are described in this section.

Object tables, as well as XMLType tables, object views, and XMLType views, do not have any column names specified for them. Therefore, Oracle defines a system-generated pseudocolumn OBJECT_ID. You can use this column name in queries and to create object views with the WITH OBJECT IDENTIFIER clause.

XMLSchema_spec

This clause lets you specify the URL of a registered XMLSchema, either in the XMLSCHEMA clause or as part of the ELEMENT clause, and an XML element name.

You must specify an element, although the XMLSchema URL is optional. If you do specify an XMLSchema URL, then you must already have registered the XMLSchema using the DBMS_XMLSCHEMA package.

The optional STORE ALL VARRAYS AS clause lets you specify how all varrays in the XMLType table or column are to be stored.

- STORE ALL VARRAYS AS LOBS indicates that all varrays are to be stored as LOBs.

- `STORE ALL VARRAYS AS TABLES` indicates that all varrays are to be stored as tables.

The optional `ALLOW` | `DISALLOW` clauses are valid only if you have specified `BINARY XML` storage.

- `ALLOW NONSCHEMA` indicates that non-schema-based documents can be stored in the `XMLType` column.
- `DISALLOW NONSCHEMA` indicates that non-schema-based documents cannot be stored in the `XMLType` column. This is the default.
- `ALLOW ANYSCHEMA` indicates that any schema-based document can be stored in the `XMLType` column.
- `DISALLOW ANYSCHEMA` indicates that any schema-based document cannot be stored in the `XMLType` column. This is the default.

See Also:

- *Oracle Database PL/SQL Packages and Types Reference* for information on the `DBMS_XMLSCHEMA` package
- *Oracle XML DB Developer's Guide* for information on creating and working with XML data
- "XMLType Table Examples"

JSON_Collection_table

Specify `JSON COLLECTION` to create a special table that stores JSON documents in a single column called `DATA`. The column `DATA` is of type `JSON`.

Each document in a collection table automatically has a document-identifier field, `_id`, at the top level, whose value is unique for the collection.

You can perform `SELECT`s, `JOIN`s, `UPDATE`s and all the normal table operations with JSON collections that you do in other tables.

- `WITH ETAG` : If specified, then each JSON document contains a document-handling field `_metadata`, whose value is an object with `etag` as its only field.

  The default is not to use `ETAG`.
- *`expression_column`*: Use to create one or more virtual (expression) column expressions on the `DATA` column that you can use for partitioning.

  `DATA` is the only visible column. User-defined expression columns are always `INVISIBLE`, whether or not you explictly specify `INVISIBLE` in the definition.
- *`constraints`* : Use to add constraints on the `DATA` column or other user-defined expression columns that you create.

Use `ALTER TABLE` to modify the JSON collection table. You can add or drop constraints, user defined virtual (expression) columns, or partitions.

Restrictions

You can only add virtual (expression) columns.

You can only drop user defined virtual (expression) columns.

You cannot drop internally generated columns like `DATA`, `RESID`, or `ETAG`. Columns `RESID` and `ETAG` are internal Oracle-managed columns that are not relevant to working with JSON collection tables. These columns should not be used in customer applications. They are used internally to enforce uniqueness and provide lock-free concurrency control.

Note:

JSON Collections of the *JSON Developer's Guide*.

MEMOPTIMIZE FOR READ

Use this clause to enable fast lookup. Fast lookup improves the performance high frequency data query operations. The `MEMOPTIMIZE_POOL_SIZE` initialization parameter controls the size of the memoptimize pool. Note that the feature uses additional memory from the SGA.

- You must specify this clause as a top-level attribute of the table, it cannot be specified at the partition or subpartition level.
- You must explicitly enable the table for `MEMOPTIMIZE FOR READ` before you can read data from the table.

MEMOPTIMIZE FOR WRITE

Use this clause to enable fast ingest. Fast ingest optimizes memory processing of high frequency single row data inserts from Internet of Things (IoT) applications.

- `MEMOPTIMZE FOR WRITE` is a top-level attribute and cannot be used at the partition or subpartition level.
- A table must be enabled for `MEMOPTIMIZE FOR WRITE` before data for that table can be written to the IGA.

Restrictions

Blockchain and immutable tables do not support `MEMOPTIMZE FOR WRITE`.

Columns of `BFILE` data type do not support `MEMOPTIMZE FOR WRITE` .

PARENT

You can use this clause to create a child table in a sharded table family.

A sharded table family is a set of tables that are sharded in the same way. Corresponding partitions of all tables in a table family are stored in the same shard. This enables you to minimize the number of multishard joins when querying data in the table family.

There are two methods for creating a sharded table family. The recommended method involves using reference partitioning. However, if it is impossible or undesirable to create the primary and foreign key constraints that are required for reference partitioning, then you can use the `PARENT` clause to create a sharded table family.

The rules for creating a sharded table family differ depending on which method you use. When you create a sharded table family by using the `PARENT` clause, the following rules apply:

- The sharded table family can contain only two levels of tables: a parent table, and one or more child tables.
- All tables in the family must be explicitly partitioned using the same partitioning scheme. Each table can use a different subpartitioning scheme, or none at all.

- You must first create the parent table, and it must be a sharded table.
- You can then use the `CREATE SHARDED TABLE` ... `PARENT` ... statement to create each child table. For *`table`*, specify the name of the parent table. For *`schema`*, specify the schema that contains the parent table. If you omit *`schema`*, then the database assumes the parent table is in your own schema.

You can create multiple sharded table families with system sharding but at most one with composite or user-defined sharding.

See Also:

- *Using Oracle Sharding*

Examples

Creating Tables: General Examples

This statement shows how the `employees` table owned by the sample human resources (`hr`) schema was created. A hypothetical name is given to the table and constraints so that you can duplicate this example in your test database:

```
CREATE TABLE employees_demo
    ( employee_id    NUMBER(6)
    , first_name     VARCHAR2(20)
    , last_name      VARCHAR2(25)
       CONSTRAINT emp_last_name_nn_demo NOT NULL
    , email          VARCHAR2(25)
       CONSTRAINT emp_email_nn_demo     NOT NULL
    , phone_number   VARCHAR2(20)
    , hire_date      DATE  DEFAULT SYSDATE
       CONSTRAINT emp_hire_date_nn_demo  NOT NULL
    , job_id         VARCHAR2(10)
     CONSTRAINT     emp_job_nn_demo  NOT NULL
    , salary         NUMBER(8,2)
     CONSTRAINT     emp_salary_nn_demo  NOT NULL
    , commission_pct NUMBER(2,2)
    , manager_id     NUMBER(6)
    , department_id  NUMBER(4)
    , dn             VARCHAR2(300)
    , CONSTRAINT     emp_salary_min_demo
                     CHECK (salary > 0)
    , CONSTRAINT     emp_email_uk_demo
                     UNIQUE (email)
    ) ;
```

This table contains twelve columns. The `employee_id` column is of data type `NUMBER`. The `hire_date` column is of data type `DATE` and has a default value of `SYSDATE`. The `last_name` column is of type `VARCHAR2` and has a `NOT NULL` constraint, and so on.

Creating a Table: Storage Example

To define the same `employees_demo` table in the `example` tablespace with a small storage capacity, issue the following statement:

```
CREATE TABLE employees_demo
    ( employee_id    NUMBER(6)
    , first_name     VARCHAR2(20)
    , last_name      VARCHAR2(25)
         CONSTRAINT emp_last_name_nn_demo NOT NULL
    , email          VARCHAR2(25)
         CONSTRAINT emp_email_nn_demo     NOT NULL
    , phone_number   VARCHAR2(20)
    , hire_date      DATE  DEFAULT SYSDATE
         CONSTRAINT emp_hire_date_nn_demo  NOT NULL
    , job_id         VARCHAR2(10)
       CONSTRAINT     emp_job_nn_demo  NOT NULL
    , salary         NUMBER(8,2)
       CONSTRAINT     emp_salary_nn_demo  NOT NULL
    , commission_pct NUMBER(2,2)
    , manager_id     NUMBER(6)
    , department_id  NUMBER(4)
    , dn             VARCHAR2(300)
    , CONSTRAINT     emp_salary_min_demo
                     CHECK (salary > 0)
    , CONSTRAINT     emp_email_uk_demo
                     UNIQUE (email)
    )
   TABLESPACE example
   STORAGE (INITIAL 8M);
```

Creating a Table with a DEFAULT ON NULL Column Value: Example

The following statement creates a table `myemp`, which can be used to store employee data. The `department_id` column is defined with a `DEFAULT ON NULL` column value of 50. Therefore, if a subsequent `INSERT` statement attempts to assign a NULL value to `department_id`, then the value of 50 will be assigned instead.

```
CREATE TABLE myemp (employee_id number, last_name varchar2(25),
                    department_id NUMBER DEFAULT ON NULL 50 NOT NULL);
```

In the `employees` table, `employee_id` 178 has a NULL value for `department_id`:

```
SELECT employee_id, last_name, department_id
  FROM employees
  WHERE department_id IS NULL;

EMPLOYEE_ID LAST_NAME                 DEPARTMENT_ID
----------- ------------------------- -------------
        178 Grant
```

Populate the `myemp` table with the `employee_id`, `last_name`, and `department_id` column data from the `employees` table:

```
INSERT INTO myemp (employee_id, last_name, department_id)
  (SELECT employee_id, last_name, department_id from employees);
```

In the `myemp` table, `employee_id` 178 has a value of 50 for `department_id`:

```
SELECT employee_id, last_name, department_id
  FROM myemp
  WHERE employee_id = 178;

EMPLOYEE_ID LAST_NAME                 DEPARTMENT_ID
----------- ------------------------- -------------
        178 Grant                                50
```

Creating a Table with an Identity Column: Examples

The following statement creates a table t1 with an identity column id. The sequence generator will always assign increasing integer values to id, starting with 1.

```
CREATE TABLE t1 (id NUMBER GENERATED AS IDENTITY);
```

The following statement creates a table t2 with an identity column id. The sequence generator will, by default, assign increasing integer values to id in increments of 10 starting with 100.

```
CREATE TABLE t2 (id NUMBER GENERATED BY DEFAULT AS IDENTITY (START WITH 100 INCREMENT BY
10));
```

Creating a Table: Temporary Table Example

The following statement creates a temporary table today_sales for use by sales representatives in the sample database. Each sales representative session can store its own sales data for the day in the table. The temporary data is deleted at the end of the session.

```
CREATE GLOBAL TEMPORARY TABLE today_sales
   ON COMMIT PRESERVE ROWS
   AS SELECT * FROM orders WHERE order_date = SYSDATE;
```

Creating a Table with Deferred Segment Creation: Example

The following statement creates a table with deferred segment creation. Oracle Database will not create a segment for the data of this table until data is inserted into the table:

```
CREATE TABLE later (col1 NUMBER, col2 VARCHAR2(20))    SEGMENT CREATION DEFERRED;
```

Substitutable Table and Column Examples

The following statements create a type hierarchy, which can be used to create a substitutable table. Type employee_t inherits the name and ssn attributes from type person_t and in addition has department_id and salary attributes. Type part_time_emp_t inherits all of the attributes from employee_t and, through employee_t, those of person_t and in addition has a num_hrs attribute. Type part_time_emp_t is final by default, so no further subtypes can be created under it.

```
CREATE TYPE person_t AS OBJECT (name VARCHAR2(100), ssn NUMBER)
   NOT FINAL;
/

CREATE TYPE employee_t UNDER person_t
   (department_id NUMBER, salary NUMBER) NOT FINAL;
/

CREATE TYPE part_time_emp_t UNDER employee_t (num_hrs NUMBER);
/
```

The following statement creates a substitutable table from the person_t type:

```
CREATE TABLE persons OF person_t;
```

The following statement creates a table with a substitutable column of type person_t:

```
CREATE TABLE books (title VARCHAR2(100), author person_t);
```

When you insert into persons or books, you can specify values for the attributes of person_t or any of its subtypes. Examples of insert statements appear in "Inserting into a Substitutable Tables and Columns: Examples".

You can extract data from such tables using built-in functions and conditions. For examples, see the functions TREAT and SYS_TYPEID , and the "IS OF *type* Condition " condition.

Creating a Table: Parallelism Examples

The following statement creates a table using an optimum number of parallel execution servers to scan `employees` and to populate `dept_80`:

```
CREATE TABLE dept_80
   PARALLEL
   AS SELECT * FROM employees
   WHERE department_id = 80;
```

Using parallelism speeds up the creation of the table, because the database uses parallel execution servers to create the table. After the table is created, querying the table is also faster, because the same degree of parallelism is used to access the table.

The following statement creates the same table serially. Subsequent DML and queries on the table will also be serially executed.

```
CREATE TABLE dept_80
   AS SELECT * FROM employees
   WHERE department_id = 80;
```

Creating a Table: ENABLE/DISABLE Examples

The following statement shows how the sample table `departments` was created. The example defines a `NOT NULL` constraint, and places it in `ENABLE VALIDATE` state:

```
CREATE TABLE departments_demo
    ( department_id    NUMBER(4)
    , department_name  VARCHAR2(30)
      CONSTRAINT  dept_name_nn  NOT NULL
    , manager_id       NUMBER(6)
    , location_id      NUMBER(4)
    , dn               VARCHAR2(300)
    ) ;
```

The following statement creates the same `departments_demo` table but also defines a disabled primary key constraint:

```
CREATE TABLE departments_demo
    ( department_id    NUMBER(4)   PRIMARY KEY DISABLE
    , department_name  VARCHAR2(30)
           CONSTRAINT  dept_name_nn  NOT NULL
    , manager_id       NUMBER(6)
    , location_id      NUMBER(4)
    , dn               VARCHAR2(300)
    ) ;
```

Nested Table Example

The following statement shows how the sample table `pm.print_media` was created with a nested table column `ad_textdocs_ntab`:

```
CREATE TABLE print_media
    ( product_id        NUMBER(6)
    , ad_id             NUMBER(6)
    , ad_composite      BLOB
    , ad_sourcetext     CLOB
    , ad_finaltext      CLOB
    , ad_fltextn        NCLOB
    , ad_textdocs_ntab  textdoc_tab
    , ad_photo          BLOB
    , ad_graphic        BFILE
    , ad_header         adheader_typ
    ) NESTED TABLE ad_textdocs_ntab STORE AS textdocs_nestedtab;
```

Creating a Table: Multilevel Collection Example

The following example shows how an account manager might create a table of customers using two levels of nested tables:

```
CREATE TYPE phone AS OBJECT (telephone NUMBER);
/
CREATE TYPE phone_list AS TABLE OF phone;
/
CREATE TYPE my_customers AS OBJECT (
   cust_name VARCHAR2(25),
   phones phone_list);
/
CREATE TYPE customer_list AS TABLE OF my_customers;
/
CREATE TABLE business_contacts (
   company_name VARCHAR2(25),
   company_reps customer_list)
   NESTED TABLE company_reps STORE AS outer_ntab
   (NESTED TABLE phones STORE AS inner_ntab);
```

The following variation of this example shows how to use the `COLUMN_VALUE` keyword if the inner nested table has no column or attribute name:

```
CREATE TYPE phone AS TABLE OF NUMBER;
/
CREATE TYPE phone_list AS TABLE OF phone;
/
CREATE TABLE my_customers (
   name VARCHAR2(25),
   phone_numbers phone_list)
   NESTED TABLE phone_numbers STORE AS outer_ntab
   (NESTED TABLE COLUMN_VALUE STORE AS inner_ntab);
```

Creating a Table: LOB Column Example

The following statement is a variation of the statement that created the `print_media` table with some added LOB storage characteristics:

```
CREATE TABLE print_media_new
    ( product_id        NUMBER(6)
    , ad_id             NUMBER(6)
    , ad_composite      BLOB
    , ad_sourcetext     CLOB
    , ad_finaltext      CLOB
    , ad_fltextn        NCLOB
    , ad_textdocs_ntab  textdoc_tab
    , ad_photo          BLOB
    , ad_graphic        BFILE
    , ad_header         adheader_typ
    ) NESTED TABLE ad_textdocs_ntab STORE AS textdocs_nestedtab_new
    LOB (ad_sourcetext, ad_finaltext) STORE AS
      (TABLESPACE example
       STORAGE (INITIAL 6144)
       CHUNK 4000
       NOCACHE LOGGING);
```

In the example above, the database rounds the value of `CHUNK` up to 4096 (the nearest multiple of the block size of 2048).

Index-Organized Table Example

The following statement is a variation of the sample table `hr.countries`, which is index organized:

```
CREATE TABLE countries_demo
    ( country_id      CHAR(2)
      CONSTRAINT country_id_nn_demo NOT NULL
    , country_name    VARCHAR2(40)
    , currency_name   VARCHAR2(25)
    , currency_symbol VARCHAR2(3)
    , region          VARCHAR2(15)
    , CONSTRAINT    country_c_id_pk_demo
                    PRIMARY KEY (country_id ) )
    ORGANIZATION INDEX
    INCLUDING   country_name
    PCTTHRESHOLD 2
    STORAGE
     ( INITIAL  4K )
   OVERFLOW
    STORAGE
      ( INITIAL  4K );
```

External Table Example

The following statement creates an external table that represents a subset of the sample table `hr.departments`. The `TYPE` clause specifies that the access driver type for the table is `ORACLE_LOADER`. The `ACCESS PARAMETERS()` clause specifies parameter values for the `ORACLE_LOADER` access driver. These parameters are shown in italics and form the *`opaque_format_spec`*. The syntax for *`opaque_format_spec`* depends on the access driver type and is outside the scope of this document. Refer to *Oracle Database Utilities* for details on the `ORACLE_LOADER` access driver and the *`opaque_format_spec`* syntax.

```
CREATE TABLE dept_external (
   deptno     NUMBER(6),
   dname      VARCHAR2(20),
   loc        VARCHAR2(25)
)
ORGANIZATION EXTERNAL
(TYPE oracle_loader
 DEFAULT DIRECTORY admin
 ACCESS PARAMETERS
 (
  RECORDS DELIMITED BY newline
  BADFILE 'ulcase1.bad'
  DISCARDFILE 'ulcase1.dis'
  LOGFILE 'ulcase1.log'
  SKIP 20
  FIELDS TERMINATED BY ","  OPTIONALLY ENCLOSED BY '"'
  (
   deptno     INTEGER EXTERNAL(6),
   dname      CHAR(20),
   loc        CHAR(25)
  )
 )
 LOCATION ('ulcase1.ctl')
)
REJECT LIMIT UNLIMITED;
```

See Also:

"Creating a Directory: Examples" to see how the `admin` directory was created

XMLType Examples

This section contains brief examples of creating an `XMLType` table or `XMLType` column. For a more expanded version of these examples, refer to "Using XML in SQL Statements ".

XMLType Table Examples

The following example creates a very simple `XMLType` table with one implicit binary XML column:

```
CREATE TABLE xwarehouses OF XMLTYPE;
```

The following example creates an XMLSchema-based table. The XMLSchema must already have been created (see "Using XML in SQL Statements " for more information):

```
CREATE TABLE xwarehouses OF XMLTYPE
   XMLSCHEMA "http://www.example.com/xwarehouses.xsd"
   ELEMENT "Warehouse";
```

You can define constraints on an XMLSchema-based table, and you can also create indexes on XMLSchema-based tables, which greatly enhance subsequent queries. You can create object-relational views on `XMLType` tables, and you can create `XMLType` views on object-relational tables.

See Also:

- "Using XML in SQL Statements " for an example of adding a constraint
- "Creating an Index on an XMLType Table: Example" for an example of creating an index
- "Creating an XMLType View: Example" for an example of creating an `XMLType` view

XMLType Column Examples

The following example creates a table with an `XMLType` column stored as a `CLOB`. This table does not require an XMLSchema, so the content structure is not predetermined:

```
CREATE TABLE xwarehouses (
   warehouse_id        NUMBER,
   warehouse_spec      XMLTYPE)
   XMLTYPE warehouse_spec STORE AS CLOB
   (TABLESPACE example
    STORAGE (INITIAL 6144)
    CHUNK 4000
    NOCACHE LOGGING);
```

The following example creates a similar table, but stores `XMLType` data in an object relational `XMLType` column whose structure is determined by the specified schema:

```
CREATE TABLE xwarehouses (
   warehouse_id    NUMBER,
   warehouse_spec  XMLTYPE)
   XMLTYPE warehouse_spec STORE AS OBJECT RELATIONAL
      XMLSCHEMA "http://www.example.com/xwarehouses.xsd"
      ELEMENT "Warehouse";
```

The following example creates another similar table with an `XMLType` column stored as a SecureFiles `CLOB`. This table does not require an XMLSchema, so the content structure is not predetermined. SecureFiles LOBs require a tablespace with automatic segment-space management, so the example uses the tablespace created in "Specifying Segment Space Management for a Tablespace: Example".

```
CREATE TABLE xwarehouses (
  warehouse_id   NUMBER,
  warehouse_spec XMLTYPE)
  XMLTYPE        warehouse_spec STORE AS SECUREFILE CLOB
  (TABLESPACE auto_seg_ts
  STORAGE (INITIAL 6144)
  CACHE);
```

Partitioning Examples

Range Partitioning Example

The `sales` table in the sample schema `sh` is partitioned by range. The following example shows an abbreviated variation of the `sales` table. Constraints and storage elements have been omitted from the example.

```
CREATE TABLE range_sales
    ( prod_id        NUMBER(6)
    , cust_id        NUMBER
    , time_id        DATE
    , channel_id     CHAR(1)
    , promo_id       NUMBER(6)
    , quantity_sold  NUMBER(3)
    , amount_sold         NUMBER(10,2)
    )
PARTITION BY RANGE (time_id)
  (PARTITION SALES_Q1_1998 VALUES LESS THAN (TO_DATE('01-APR-1998','DD-MON-YYYY')),
   PARTITION SALES_Q2_1998 VALUES LESS THAN (TO_DATE('01-JUL-1998','DD-MON-YYYY')),
   PARTITION SALES_Q3_1998 VALUES LESS THAN (TO_DATE('01-OCT-1998','DD-MON-YYYY')),
   PARTITION SALES_Q4_1998 VALUES LESS THAN (TO_DATE('01-JAN-1999','DD-MON-YYYY')),
   PARTITION SALES_Q1_1999 VALUES LESS THAN (TO_DATE('01-APR-1999','DD-MON-YYYY')),
   PARTITION SALES_Q2_1999 VALUES LESS THAN (TO_DATE('01-JUL-1999','DD-MON-YYYY')),
   PARTITION SALES_Q3_1999 VALUES LESS THAN (TO_DATE('01-OCT-1999','DD-MON-YYYY')),
   PARTITION SALES_Q4_1999 VALUES LESS THAN (TO_DATE('01-JAN-2000','DD-MON-YYYY')),
   PARTITION SALES_Q1_2000 VALUES LESS THAN (TO_DATE('01-APR-2000','DD-MON-YYYY')),
   PARTITION SALES_Q2_2000 VALUES LESS THAN (TO_DATE('01-JUL-2000','DD-MON-YYYY')),
   PARTITION SALES_Q3_2000 VALUES LESS THAN (TO_DATE('01-OCT-2000','DD-MON-YYYY')),
   PARTITION SALES_Q4_2000 VALUES LESS THAN (MAXVALUE))
;
```

For information about partitioned table maintenance operations, see *Oracle Database VLDB and Partitioning Guide*.

Range Partitioning Live SQL Example

The following statement creates a table partitioned by range:

```
CREATE TABLE empl_h
  (
     employee_id  NUMBER(6) PRIMARY KEY,
     first_name   VARCHAR2(20),
     last_name    VARCHAR2(25),
     email        VARCHAR2(25),
     phone_number VARCHAR2(20),
     hire_date    DATE DEFAULT SYSDATE,
     job_id       VARCHAR2(10),
     salary       NUMBER(8, 2),
     part_name    VARCHAR2(25)
  ) PARTITION BY RANGE (hire_date) (
PARTITION hire_q1 VALUES less than(to_date('01-APR-2014', 'DD-MON-YYYY')),
PARTITION hire_q2 VALUES less than(to_date('01-JUL-2014', 'DD-MON-YYYY')),
PARTITION hire_q3 VALUES less than(to_date('01-OCT-2014', 'DD-MON-YYYY')),
PARTITION hire_q4 VALUES less than(to_date('01-JAN-2015', 'DD-MON-YYYY'))
);
```

The following statements insert rows into the partitions:

```
INSERT INTO empl_h (employee_id, first_name, last_name, email, phone_number, hire_date, 
job_id, salary, Part_name)
VALUES (1, 'Jane', 'Doe', 'example.com', '415.555.0100', '10-Feb-2014', '1001', 
5001,'HIRE_Q1');

INSERT INTO empl_h (employee_id, first_name, last_name, email, phone_number, hire_date, 
job_id, salary, Part_name)
VALUES (2, 'John', 'Doe', 'example.net', '415.555.0101', '10-Apr-2014', '1002', 
7001,'HIRE_Q2');

INSERT INTO empl_h (employee_id, first_name, last_name, email, phone_number, hire_date, 
job_id, salary, Part_name)
VALUES (3, 'Isabelle', 'Owl', 'example.org', '415.555.0102', '10-Sep-2014', '1003', 
10001,'HIRE_Q3');

INSERT INTO empl_h (employee_id, first_name, last_name, email, phone_number, hire_date, 
job_id, salary, Part_name)
VALUES (4, 'Smith', 'Jones', 'example.in', '415.555.0103', '10-Dec-2014', '1004', 
12001,'HIRE_Q4');
```

The following statements display the partition names using data dictionary tables:

```
SELECT PARTITION_NAME FROM USER_TAB_PARTITIONS WHERE TABLE_NAME = 'EMPL_H';

PARTITION_NAME
----------------
HIRE_Q1
HIRE_Q2
HIRE_Q3
HIRE_Q4

SELECT TABLE_NAME, PARTITIONING_TYPE, STATUS FROM USER_PART_TABLES WHERE TABLE_NAME = 
'EMPL_H';

TABLE_NAME    PARTITIONING_TYPE    STATUS
----------    -----------------    ------
EMPL_H        RANGE             VALID
```

The following statement creates a table named `parts` by selecting a particular column from the data dictionary table `user_tab_partitions`:

```
CREATE TABLE parts (p_name) AS SELECT PARTITION_NAME FROM USER_TAB_PARTITIONS WHERE 
TABLE_NAME = 'EMPL_H';
```

The following statement displays the table data:

```
select * from parts;

P_NAME
-----------
HIRE_Q1
HIRE_Q2
HIRE_Q3
HIRE_Q4
```

The following statement compares the columns from the two tables and displays the information based on the comparison:

```
select E.HIRE_DATE,E.JOB_ID,P.p_name from empl_h E, parts P where E.Part_name = P.p_name;

HIRE_DATE JOB_ID     P_NAME
--------- ---------- ------------
10-FEB-14 1001         HIRE_Q1
10-APR-14 1002         HIRE_Q2
10-SEP-14 1003         HIRE_Q3
10-DEC-14 1004         HIRE_Q4
```

Interval Partitioning Example

The following example creates a variation of the `oe.customers` table that is partitioned by interval on the `credit_limit` column. One range partition is created to establish the transition point. All of the original data in the table is within the bounds of the range partition. Then data is added that exceeds the range partition, and the database creates a new interval partition.

```
CREATE TABLE customers_demo (
  customer_id number(6),
  cust_first_name varchar2(20),
  cust_last_name varchar2(20),
  credit_limit number(9,2))
PARTITION BY RANGE (credit_limit)
INTERVAL (1000)
(PARTITION p1 VALUES LESS THAN (5001));

INSERT INTO customers_demo
  (customer_id, cust_first_name, cust_last_name, credit_limit)
  (select customer_id, cust_first_name, cust_last_name, credit_limit
  from customers);
```

Query the `USER_TAB_PARTITIONS` data dictionary view before the database creates the interval partition:

```
SELECT partition_name, high_value FROM user_tab_partitions  WHERE table_name =
'CUSTOMERS_DEMO';

PARTITION_NAME                 HIGH_VALUE
------------------------------ ---------------
P1                             5001
```

Insert data into the table that exceeds the high value of the range partition:

```
INSERT INTO customers_demo
  VALUES (699, 'Fred', 'Flintstone', 5500);
```

Query the USER_TAB_PARTITIONS view again after the insert to learn the system-generated name of the interval partition created to accommodate the inserted data. (The system-generated name will vary for each session.)

```
SELECT partition_name, high_value FROM user_tab_partitions
  WHERE table_name = 'CUSTOMERS_DEMO'
  ORDER BY partition_name;

PARTITION_NAME                 HIGH_VALUE
------------------------------ ---------------
P1                             5001
SYS_P44                        6001
```

List Partitioning Example

The following statement shows how the sample table oe.customers might have been created as a list-partitioned table. Some columns and all constraints of the sample table have been omitted in this example.

```
CREATE TABLE list_customers
   ( customer_id             NUMBER(6)
   , cust_first_name         VARCHAR2(20)
   , cust_last_name          VARCHAR2(20)
   , cust_address            CUST_ADDRESS_TYP
   , nls_territory           VARCHAR2(30)
   , cust_email              VARCHAR2(40))
   PARTITION BY LIST (nls_territory) (
   PARTITION asia VALUES ('CHINA', 'THAILAND'),
   PARTITION europe VALUES ('GERMANY', 'ITALY', 'SWITZERLAND'),
   PARTITION west VALUES ('AMERICA'),
   PARTITION east VALUES ('INDIA'),
   PARTITION rest VALUES (DEFAULT));
```

Partitioned Table with LOB Columns Example

This statement creates a partitioned table print_media_demo with two partitions p1 and p2, and a number of LOB columns. The statement uses the sample table pm.print_media.

```
CREATE TABLE print_media_demo
   ( product_id NUMBER(6)
   , ad_id NUMBER(6)
   , ad_composite BLOB
   , ad_sourcetext CLOB
   , ad_finaltext CLOB
   , ad_fltextn NCLOB
   , ad_textdocs_ntab textdoc_tab
   , ad_photo BLOB
   , ad_graphic BFILE
   , ad_header adheader_typ
   ) NESTED TABLE ad_textdocs_ntab STORE AS textdocs_nestedtab_demo
      LOB (ad_composite, ad_photo, ad_finaltext)
      STORE AS(STORAGE (INITIAL 20M))
   PARTITION BY RANGE (product_id)
      (PARTITION p1 VALUES LESS THAN (3000) TABLESPACE tbs_01
         LOB (ad_composite, ad_photo)
         STORE AS (TABLESPACE tbs_02 STORAGE (INITIAL 10M))
         NESTED TABLE ad_textdocs_ntab STORE AS nt_p1 (TABLESPACE example),
       PARTITION P2 VALUES LESS THAN (MAXVALUE)
         LOB (ad_composite, ad_finaltext)
         STORE AS SECUREFILE (TABLESPACE auto_seg_ts)
         NESTED TABLE ad_textdocs_ntab STORE AS nt_p2
       )
   TABLESPACE tbs_03;
```

Partition p1 will be in tablespace tbs_01. The LOB data partitions for ad_composite and ad_photo will be in tablespace tbs_02. The LOB data partition for the remaining LOB columns will be in tablespace tbs_01. The storage attribute INITIAL is specified for LOB columns ad_composite and ad_photo. Other attributes will be inherited from the default table-level specification. The default LOB storage attributes not specified at the table level will be inherited from the tablespace tbs_02 for columns ad_composite and ad_photo and from tablespace tbs_01 for the remaining LOB columns. LOB index partitions will be in the same tablespaces as the corresponding LOB data partitions. Other storage attributes will be based on values of the corresponding attributes of the LOB data partitions and default attributes of the tablespace where the index partitions reside. The nested table partition for ad_textdocs_ntab will be stored as nt_p1 in tablespace example.

Partition p2 will be in the default tablespace tbs_03. The LOB data for ad_composite and ad_finaltext will be in tablespace auto_seg_ts as SecureFiles LOBs. The LOB data for the remaining LOB columns will be in tablespace tbs_03. The LOB index for columns ad_composite and ad_finaltext will be in tablespace auto_seg_ts. The LOB index for the remaining LOB columns will be in tablespace tbs_03. The nested table partition for ad_textdocs_ntab will be stored as nt_p2 in the default tablespace tbs_03.

Hash Partitioning Example

The sample table oe.product_information is not partitioned. However, you might want to partition such a large table by hash for performance reasons, as shown in this example. The tablespace names are hypothetical in this example.

```
CREATE TABLE hash_products
    ( product_id          NUMBER(6)   PRIMARY KEY
    , 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          product_status_lov_demo
                          CHECK (product_status in ('orderable'
                                                   ,'planned'
                                                   ,'under development'
                                                   ,'obsolete')
 ) )
 PARTITION BY HASH (product_id)
 PARTITIONS 4
 STORE IN (tbs_01, tbs_02, tbs_03, tbs_04);
```

Reference Partitioning Example

The next statement uses the hash_products partitioned table created in the preceding example. It creates a variation of the oe.order_items table that is partitioned by reference to the hash partitioning on the product id of hash_products. The resulting child table will be created with five partitions. For each row of the child table part_order_items, the database evaluates the foreign key value (product_id) to determine the partition number of the parent table hash_products to which the referenced key belongs. The part_order_items row is placed in its corresponding partition.

```
CREATE TABLE part_order_items (
    order_id        NUMBER(12) PRIMARY KEY,
    line_item_id    NUMBER(3),
    product_id      NUMBER(6) NOT NULL,
    unit_price      NUMBER(8,2),
    quantity        NUMBER(8),
    CONSTRAINT product_id_fk
    FOREIGN KEY (product_id) REFERENCES hash_products(product_id))
 PARTITION BY REFERENCE (product_id_fk);
```

Composite-Partitioned Table Examples

The table created in the "Range Partitioning Example" divides data by time of sale. If you plan to access recent data according to distribution channel as well as time, then composite partitioning might be more appropriate. The following example creates a copy of that `range_sales` table but specifies range-hash composite partitioning. The partitions with the most recent data are subpartitioned with both system-generated and user-defined subpartition names. Constraints and storage attributes have been omitted from the example.

```
CREATE TABLE composite_sales
    ( prod_id        NUMBER(6)
    , cust_id        NUMBER
    , time_id        DATE
    , channel_id     CHAR(1)
    , promo_id       NUMBER(6)
    , quantity_sold  NUMBER(3)
    , amount_sold         NUMBER(10,2)
    )
PARTITION BY RANGE (time_id)
SUBPARTITION BY HASH (channel_id)
  (PARTITION SALES_Q1_1998 VALUES LESS THAN (TO_DATE('01-APR-1998','DD-MON-YYYY')),
   PARTITION SALES_Q2_1998 VALUES LESS THAN (TO_DATE('01-JUL-1998','DD-MON-YYYY')),
   PARTITION SALES_Q3_1998 VALUES LESS THAN (TO_DATE('01-OCT-1998','DD-MON-YYYY')),
   PARTITION SALES_Q4_1998 VALUES LESS THAN (TO_DATE('01-JAN-1999','DD-MON-YYYY')),
   PARTITION SALES_Q1_1999 VALUES LESS THAN (TO_DATE('01-APR-1999','DD-MON-YYYY')),
   PARTITION SALES_Q2_1999 VALUES LESS THAN (TO_DATE('01-JUL-1999','DD-MON-YYYY')),
   PARTITION SALES_Q3_1999 VALUES LESS THAN (TO_DATE('01-OCT-1999','DD-MON-YYYY')),
   PARTITION SALES_Q4_1999 VALUES LESS THAN (TO_DATE('01-JAN-2000','DD-MON-YYYY')),
   PARTITION SALES_Q1_2000 VALUES LESS THAN (TO_DATE('01-APR-2000','DD-MON-YYYY')),
   PARTITION SALES_Q2_2000 VALUES LESS THAN (TO_DATE('01-JUL-2000','DD-MON-YYYY'))
      SUBPARTITIONS 8,
   PARTITION SALES_Q3_2000 VALUES LESS THAN (TO_DATE('01-OCT-2000','DD-MON-YYYY'))
     (SUBPARTITION ch_c,
      SUBPARTITION ch_i,
      SUBPARTITION ch_p,
      SUBPARTITION ch_s,
      SUBPARTITION ch_t),
   PARTITION SALES_Q4_2000 VALUES LESS THAN (MAXVALUE)
      SUBPARTITIONS 4)
;
```

The following examples creates a partitioned table of customers based on the sample table `oe.customers`. In this example, the table is partitioned on the `credit_limit` column and list subpartitioned on the `nls_territory` column. The subpartition template determines the subpartitioning of any subsequently added partitions, unless you override the template by defining individual subpartitions. This composite partitioning makes it possible to query the table based on a credit limit range within a specified region:

```
CREATE TABLE customers_part (
   customer_id        NUMBER(6),
   cust_first_name    VARCHAR2(20),
   cust_last_name     VARCHAR2(20),
   nls_territory      VARCHAR2(30),
   credit_limit       NUMBER(9,2))
   PARTITION BY RANGE (credit_limit)
   SUBPARTITION BY LIST (nls_territory)
      SUBPARTITION TEMPLATE
         (SUBPARTITION east  VALUES
            ('CHINA', 'JAPAN', 'INDIA', 'THAILAND'),
          SUBPARTITION west VALUES
             ('AMERICA', 'GERMANY', 'ITALY', 'SWITZERLAND'),
          SUBPARTITION other VALUES (DEFAULT))
      (PARTITION p1 VALUES LESS THAN (1000),
       PARTITION p2 VALUES LESS THAN (2500),
       PARTITION p3 VALUES LESS THAN (MAXVALUE));
```

Object Column and Table Examples

Creating Object Tables: Examples

Consider object type `department_typ`:

```
CREATE TYPE department_typ AS OBJECT
   ( d_name   VARCHAR2(100),
     d_address VARCHAR2(200) );
/
```

Object table `departments_obj_t` holds department objects of type `department_typ`:

```
CREATE TABLE departments_obj_t OF department_typ;
```

The following statement creates object table `salesreps` with a user-defined object type, `salesrep_typ`:

```
CREATE OR REPLACE TYPE salesrep_typ AS OBJECT
  ( repId NUMBER,
    repName VARCHAR2(64));

CREATE TABLE salesreps OF salesrep_typ;
```

Creating a Table with a User-Defined Object Identifier: Example

This example creates an object type and a corresponding object table whose object identifier is primary key based:

```
CREATE TYPE employees_typ AS OBJECT
   (e_no NUMBER, e_address CHAR(30));
/

CREATE TABLE employees_obj_t OF employees_typ (e_no PRIMARY KEY)
   OBJECT IDENTIFIER IS PRIMARY KEY;
```

You can subsequently reference the `employees_obj_t` object table using either *`inline_ref_constraint`* or *`out_of_line_ref_constraint`* syntax:

```
CREATE TABLE departments_t
   (d_no    NUMBER,
    mgr_ref REF employees_typ SCOPE IS employees_obj_t);

CREATE TABLE departments_t (
    d_no NUMBER,
    mgr_ref REF employees_typ
       CONSTRAINT mgr_in_emp REFERENCES employees_obj_t);
```

Specifying Constraints on Type Columns: Example

The following example shows how to define constraints on attributes of an object type column:

```
CREATE TYPE address_t AS OBJECT
  ( hno    NUMBER,
    street VARCHAR2(40),
    city   VARCHAR2(20),
    zip    VARCHAR2(5),
    phone  VARCHAR2(10) );
/

CREATE TYPE person AS OBJECT
  ( name        VARCHAR2(40),
    dateofbirth DATE,
    homeaddress address_t,
    manager     REF person );
/

CREATE TABLE persons OF person
  ( homeaddress NOT NULL,
      UNIQUE (homeaddress.phone),
      CHECK (homeaddress.zip IS NOT NULL),
      CHECK (homeaddress.city <> 'San Francisco') );
```

Add Annotations at Table Creation: Example

The following example adds two operations with values *`Sort`* and *`Group`*, and a standalone *`Hidden`* without a value, to table *`t1`*.

```
CREATE TABLE t1 (T NUMBER) ANNOTATIONS(Operations 'Sort', Operations 'Group', Hidden);
```

The annotation can be preceded by the keyword `ADD` which is the default operation if nothing is specified as the following example shows:

```
CREATE TABLE t1 (T NUMBER) ANNOTATIONS (ADD Hidden);
```

Add Annotations to Table Columns

```
CREATE TABLE t1 (T NUMBER ANNOTATIONS(Operations 'Sort' , Hidden) );
```

Add Annotations to Table and Columns

```
CREATE TABLE Employee (
  Id  NUMBER(5) ANNOTATIONS(Identity, Display ’Employee ID’, Group ’Emp_Info’),
  Ename VARCHAR2(50) ANNOTATIONS(Display ’Employee Name’,  Group ’Emp_Info’),
  Sal NUMBER TAG ANNOTATIONS(Display ’Employee Salary’, UI_Hidden)
) ANNOTATIONS (Display ’Employee Table’);
```

Associating Table Columns to Domains Using CREATE TABLE: Example

You can associate table columns with a domain with `CREATE TABLE` or with `ALTER TABLE MODIFY`.

You must specify domain associations at the end of the statement, after all columns have been defined.

The `DOMAIN` keyword, when specified, must be followed by the domain name.

Associate Table Columns With a Domain: Example

The following example creates domain `dn1`:

```
CREATE DOMAIN dn1 AS NUMBER;
```

The following example creates domain dn2:

```
CREATE DOMAIN dn2 AS (c1 AS NUMBER NOT NULL, c2 as NUMBER DEFAULT 1);
```

The following example creates domain dm1:

```
CREATE DOMAIN dm1 AS
 (ann AS NUMBER NOT NULL ,
  bnnpos AS NUMBER NOT NULL CONSTRAINT CHECK (bnnpos > 0),
  c AS VARCHAR2(10) DEFAULT 'abc',
  ddon AS NUMBER DEFAULT ON NULL 10)
  CONSTRAINT CHECK (ann+ddon < = 100)
  CONSTRAINT CHECK (length(c) > bnnpos);
```

The following example associates columns c1, c2, c3 , c4 with domain dm1, columns c5 and c6 with domain dn2, and column c7 with dn1.

```
CREATE TABLE tm1 (c1 NUMBER, c2 NUMBER, c3 VARCHAR2(15),c4 NUMBER, c5 NUMBER,
                  c6 NUMBER, c7 NUMBER, DOMAIN dm1 (c1, c2, c3, c4),
                  DOMAIN dn2(c5, c6), DOMAIN dn1(c7));
```

Create Tables and Columns as Transportable Binary XML

Create Tables Stored as Transportable Binary XML

```
CREATE TABLE t1 OF XMLTYPE
  XMLTYPE STORE AS TRANSPORTABLE BINARY XML;
```

Create Columns Stored as Transportable Binary XML

```
CREATE TABLE t2 (id NUMBER, doc XMLTYPE)
  XMLTYPE doc STORE AS TRANSPORTABLE BINARY XML;
```

Add Columns Stored as Transportable Binary XML to Tables

```
ALTER TABLE t3 ADD (doc XMLTYPE)
  XMLTYPE doc STORE AS TRANSPORTABLE BINARY XML;
```

CREATE TABLESPACE

Purpose

Use the CREATE TABLESPACE statement to create a **tablespace**, which is an allocation of space in the database that can contain schema objects.

- A **permanent tablespace** contains persistent schema objects. Objects in permanent tablespaces are stored in **data files**.
- An **undo tablespace** is a type of permanent tablespace used by Oracle Database to manage undo data if you are running your database in automatic undo management mode. Oracle strongly recommends that you use automatic undo management mode rather than using rollback segments for undo.

- A **temporary tablespace** contains schema objects only for the duration of a session. Objects in temporary tablespaces are stored in **temp files**.

When you create a tablespace, it is initially a read/write tablespace. You can subsequently use the `ALTER TABLESPACE` statement to take the tablespace offline or online, add data files or temp files to it, or make it a read-only tablespace.

You can also drop a tablespace from the database with the `DROP TABLESPACE` statement.

See Also:

- *Oracle Database Concepts* for information on tablespaces
- ALTER TABLESPACE and DROP TABLESPACE for information on modifying and dropping tablespaces

Prerequisites

You must have the `CREATE TABLESPACE` system privilege. To create the `SYSAUX` tablespace, you must have the `SYSDBA` system privilege.

Before you can create a tablespace, you must create a database to contain it, and the database must be open.

See Also:

CREATE DATABASE

To use objects in a tablespace other than the `SYSTEM` tablespace:

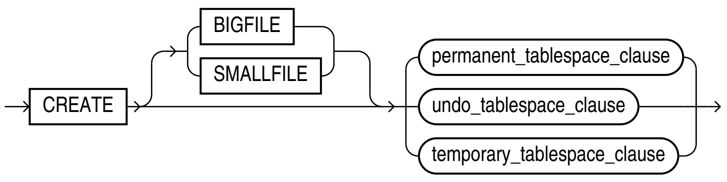
- If you are running the database in automatic undo management mode, then at least one `UNDO` tablespace must be online.
- If you are running the database in manual undo management mode, then at least one rollback segment other than the `SYSTEM` rollback segment must be online.

Note:

Oracle strongly recommends that you run your database in automatic undo management mode. For more information, refer to *Oracle Database Administrator's Guide*.

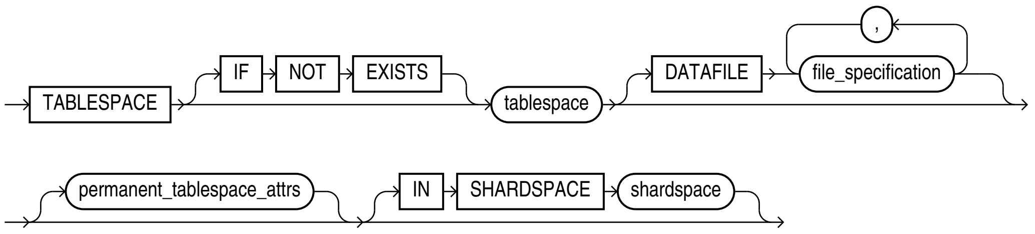
Syntax

***create_tablespace*::=**



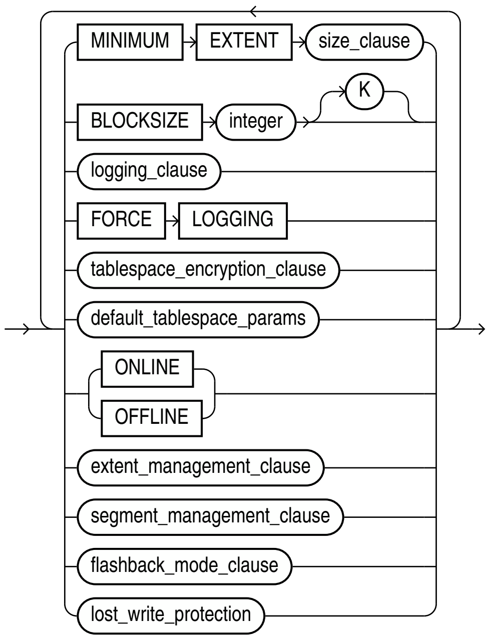
(*permanent_tablespace_clause*::=, *temporary_tablespace_clause*::=, *undo_tablespace_clause*::=)

***permanent_tablespace_clause*::=**



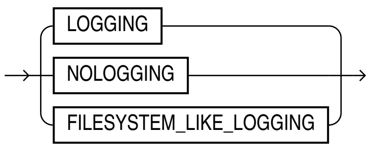
(*file_specification*::=, *permanent_tablespace_attrs*::=)

***permanent_tablespace_attrs*::=**

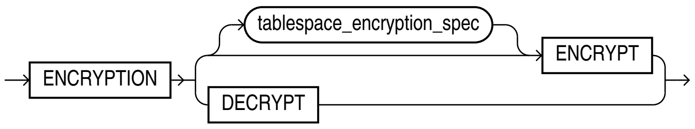


(*size_clause*::=, *logging_clause*::=, *tablespace_encryption_clause*::=, *default_tablespace_params*::=, *extent_management_clause*::=, *segment_management_clause*::=, *flashback_mode_clause*::=)

***logging_clause*::=**

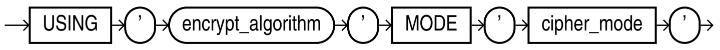


***tablespace_encryption_clause*::=**

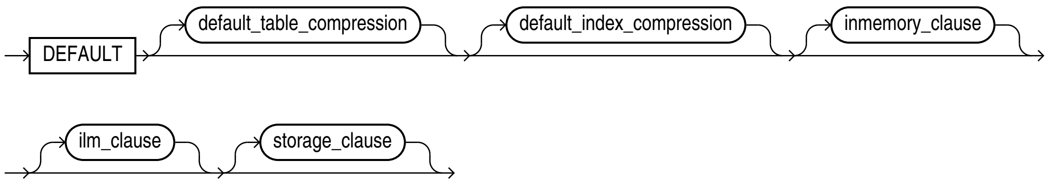


(*tablespace_encryption_spec*::=)

***tablespace_encryption_spec*::=**



***default_tablespace_params*::=**

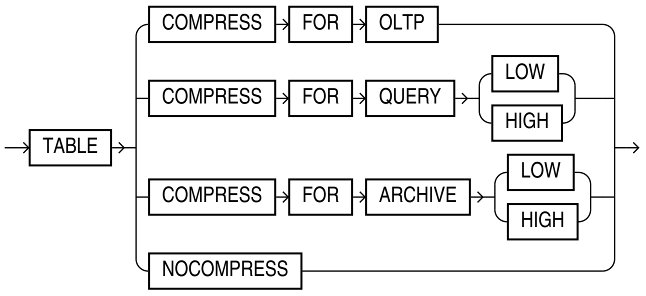


(*default_table_compression*::=, *default_index_compression*::=, *inmemory_clause*::=, *ilm_clause*::=—part of `CREATE TABLE` syntax, *storage_clause*::=)

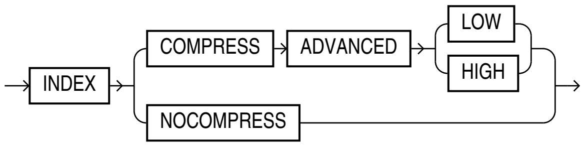
Note:

If you specify the `DEFAULT` clause, then you must specify at least one of the clauses *`default_table_compression`*, *`default_index_compression`*, *`inmemory_clause`*, *`ilm_clause`*, or *`storage_clause`*.

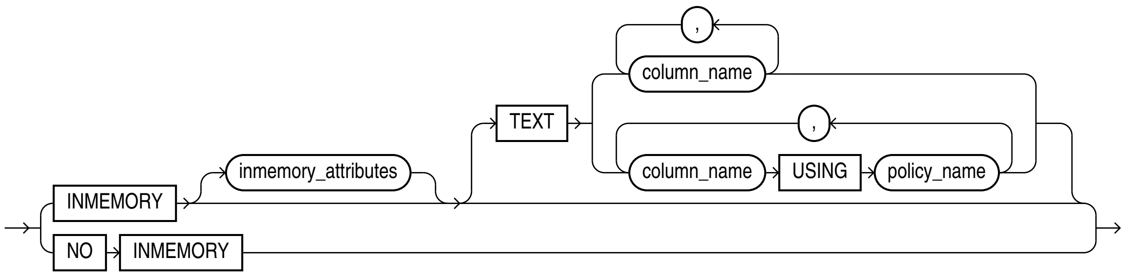
***default_table_compression*::=**



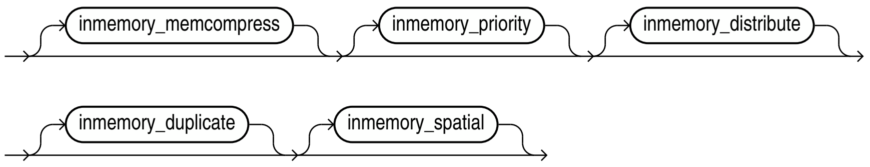
***default_index_compression*::=**



***inmemory_clause*::=**

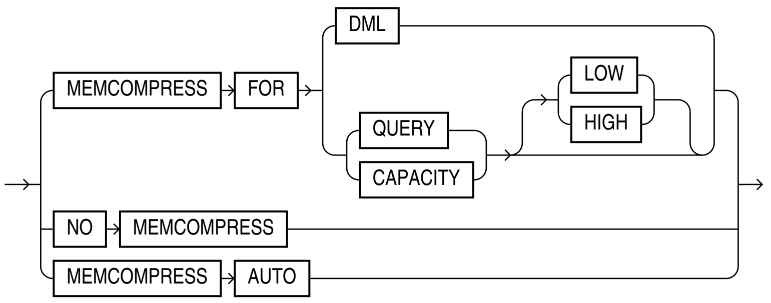


***inmemory_attributes*::=**

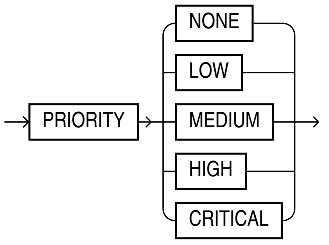


(*inmemory_memcompress*::=, *inmemory_priority*::=, *inmemory_distribute_tablespace*::=, *inmemory_duplicate*::=)

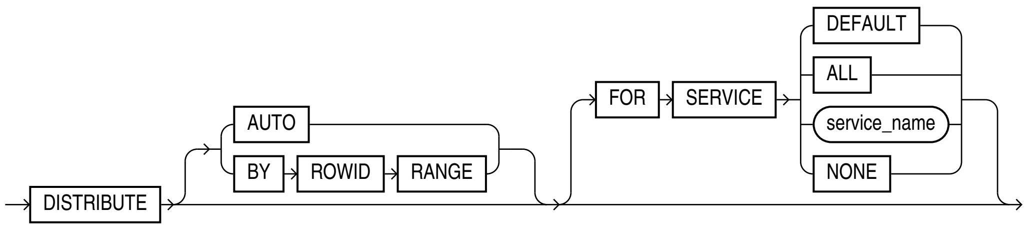
inmemory_memcompress::=



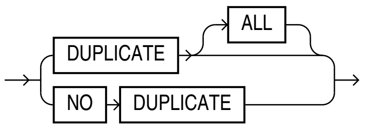
inmemory_priority::=



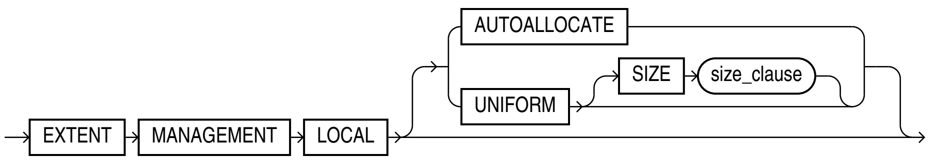
inmemory_distribute_tablespace::=



inmemory_duplicate::=

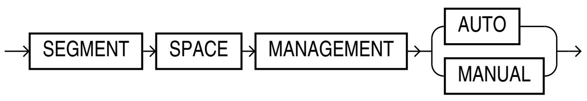


extent_management_clause::=

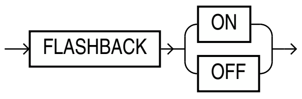


(*size_clause*::=)

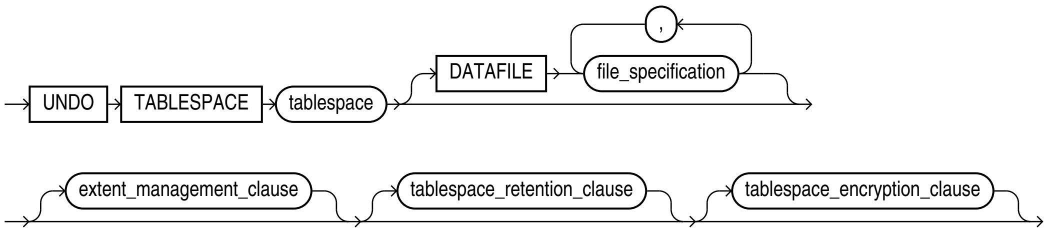
***segment_management_clause*::=**



***flashback_mode_clause*::=**

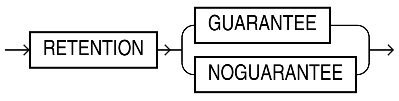


***undo_tablespace_clause*::=**

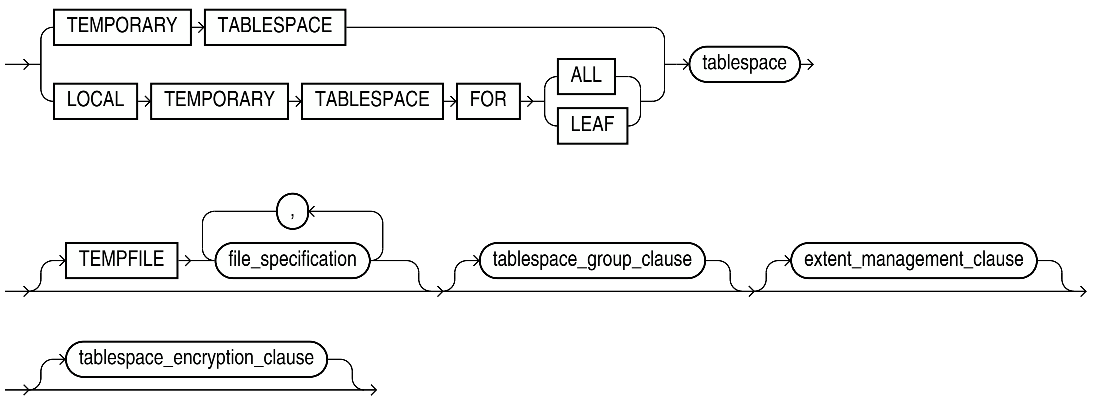


(*file_specification*::=, *extent_management_clause*::=, *tablespace_retention_clause*::=)

***tablespace_retention_clause*::=**

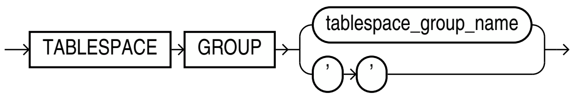


***temporary_tablespace_clause*::=**

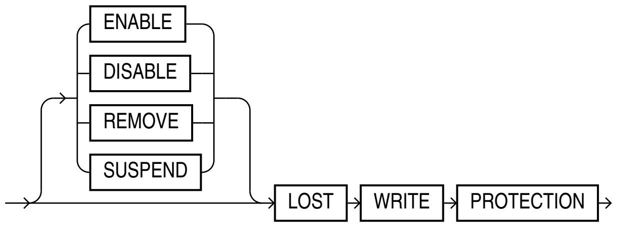


(*file_specification*::=, *tablespace_group_clause*::=, *extent_management_clause*::=, *tablespace_encryption_clause*::=)

***tablespace_group_clause*::=**



***lost_write_protection* ::=**



Semantics

BIGFILE | SMALLFILE

Use this clause to determine whether the tablespace is a bigfile or smallfile tablespace. This clause overrides any default tablespace type setting for the database.

- A **bigfile tablespace** contains only one data file or temp file, which can contain up to approximately 4 billion (2^{32}) blocks. The minimum size of the single data file or temp file is 12 megabytes (MB) for a tablespace with 32K blocks and 7MB for a tablespace with 8K 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 uses the current default tablespace type of permanent or temporary tablespace that is set for the database. If you specify `BIGFILE` for a permanent tablespace, then the database by default creates a locally managed tablespace with automatic segment-space management.

Restriction on Bigfile Tablespaces

You can specify only one data file in the `DATAFILE` clause or one temp file in the `TEMPFILE` clause.

Note:

Starting with Oracle Database 23ai, `BIGFILE` functionality is the default for `SYSAUX`, `SYSTEM`, and `USER` tablespaces.

See Also:

- *Oracle Database Administrator's Guide* for more information on using bigfile tablespaces
- "Creating a Bigfile Tablespace: Example"

permanent_tablespace_clause

Use the following clauses to create a permanent tablespace. (Some of these clauses are also used to create a temporary or undo tablespace.)

tablespace

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

IF NOT EXISTS

Specifying `IF NOT EXISTS` has the following effects:

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

Using `IF EXISTS` with `CREATE` results in error: Incorrect `IF NOT EXISTS` clause for `CREATE` statement.

Note on the SYSAUX Tablespace

`SYSAUX` is a required auxiliary system tablespace. You must use the `CREATE TABLESPACE` statement to create the `SYSAUX` tablespace if you are upgrading from a release earlier than Oracle Database 11*g*. You must have the `SYSDBA` system privilege to specify this clause, and you must have opened the database in `UPGRADE` mode.

You must specify `EXTENT MANAGEMENT LOCAL` and `SEGMENT SPACE MANAGEMENT AUTO` for the `SYSAUX` tablespace. The `DATAFILE` clause is optional only if you have enabled Oracle Managed Files. See "DATAFILE | TEMPFILE Clause" for the behavior of the `DATAFILE` clause.

Take care to allocate sufficient space for the `SYSAUX` tablespace. For guidelines on creating this tablespace, refer to *Oracle Database Upgrade Guide*.

Restrictions on the SYSAUX Tablespace

You cannot specify `OFFLINE` or `TEMPORARY` for the `SYSAUX` tablespace.

DATAFILE | TEMPFILE Clause

Specify the data files to make up the permanent tablespace or the temp files to make up the temporary tablespace. Use the *`datafile_tempfile_spec`* form of *`file_specification`* to create regular data files and temp files in an operating system file system or to create Oracle Automatic Storage Management (Oracle ASM) disk group files.

You must specify the `DATAFILE` or `TEMPFILE` clause unless you have enabled Oracle Managed Files by setting a value for the `DB_CREATE_FILE_DEST` initialization parameter. For Oracle ASM disk group files, the parameter must be set to a multiple file creation form of Oracle ASM filenames. If this parameter is set, then the database creates a system-named 100 MB file in the default file destination specified in the parameter. The file has `AUTOEXTEND` enabled and an unlimited maximum size.

Note:

Media recovery does not recognize temp files.

See Also:

- *Oracle Automatic Storage Management Administrator's Guide* for more information on using Oracle ASM
- *file_specification* for a full description, including the `AUTOEXTEND` parameter and the multiple file creation form of Oracle ASM filenames

Notes on Specifying Data Files and Temp Files

- You can create a tablespace within an Oracle ASM disk group by providing only the disk group name in the *`datafile_tempfile_spec`*. In this case, Oracle ASM creates a data file in the specified disk group with a system-generated filename. The data file is auto-extensible with an unlimited maximum size and a default size of 100 MB. You can use the *`autoextend_clause`* to override the default size.
- If you use one of the reference forms of the *`ASM_filename`*, which refers to an existing file, then you must also specify `REUSE`.

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.

See Also:

- *file_specification* for a full description, including the `AUTOEXTEND` parameter
- "Enabling Autoextend for a Tablespace: Example" and "Creating Oracle Managed Files: Examples"

permanent_tablespace_attrs

Use the `permanent_tablespace_attrs` clauses to set the attributes of the tablespace.

MINIMUM EXTENT Clause

This clause is valid only for a dictionary-managed tablespace. Specify the minimum size of an extent in the tablespace. This clause lets you control free space fragmentation in the tablespace by ensuring that the size of every used or free extent in a tablespace is at least as large as, and is a multiple of, the value specified in the `size_clause`.

> **See Also:**
>
> *size_clause* for information on that clause and *Oracle Database VLDB and Partitioning Guide* for more information about using `MINIMUM EXTENT` to control fragmentation

BLOCKSIZE Clause

Use the `BLOCKSIZE` clause to specify a nonstandard block size for the tablespace. In order to specify this clause, the `DB_CACHE_SIZE` and at least one `DB_nK_CACHE_SIZE` parameter must be set, and the integer you specify in this clause must correspond with the setting of one `DB_nK_CACHE_SIZE` parameter setting.

Restriction on BLOCKSIZE

You cannot specify nonstandard block sizes for a temporary tablespace or if you intend to assign this tablespace as the temporary tablespace for any users.

> **Note:**
>
> Oracle recommend that you do not store tablespaces with a 2K block size on 4K sector size disks, because performance degradation can result.

> **See Also:**
>
> *Oracle Database Reference* for information on the `DB_nK_CACHE_SIZE` parameter and *Oracle Database Concepts* for information on multiple block sizes

logging_clause

Specify the default logging attributes of all tables, indexes, materialized views, materialized view logs, and partitions within the tablespace. This clause is not valid for a temporary or undo tablespace.

If you omit this clause, then the default is `LOGGING`. The exception is creating a tablespace in a PDB. In this case, if you omit this clause, then the tablespace uses the logging attribute of the PDB. Refer to the *logging_clause* of `CREATE PLUGGABLE DATABASE` for more information.

The tablespace-level logging attribute can be overridden by logging specifications at the table, index, materialized view, materialized view log, and partition levels.

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

FORCE LOGGING

Use this clause to put the tablespace into `FORCE LOGGING` mode. Oracle Database will log all changes to all objects in the tablespace except changes to temporary segments, overriding any `NOLOGGING` setting for individual objects. The database must be open and in `READ WRITE` mode.

This setting does not exclude the `NOLOGGING` attribute. You can specify both `FORCE LOGGING` and `NOLOGGING`. In this case, `NOLOGGING` is the default logging mode for objects subsequently created in the tablespace, but the database ignores this default as long as the tablespace or the database is in `FORCE LOGGING` mode. If you subsequently take the tablespace out of `FORCE LOGGING` mode, then the `NOLOGGING` default is once again enforced.

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

Restriction on Forced Logging

You cannot specify `FORCE LOGGING` for an undo or temporary tablespace.

tablespace_encryption_clause

Use this clause to specify whether to create an encrypted or unencrypted tablespace. If you create an encrypted tablespace, then Transparent Data Encryption (TDE) is applied to all data files of the tablespace.

`ENCRYPT | DECRYPT`
Specify `ENCRYPT` to create an encrypted tablespace. Specify `DECRYPT` to create an unencrypted tablespace.

If you omit this clause, then the value of the `ENCRYPT_NEW_TABLESPACES` initialization parameter determines whether the tablespace is encrypted upon creation. Refer to *Oracle Database Reference* for more information on the `ENCRYPT_NEW_TABLESPACES` initialization parameter.

Before issuing this clause, you must already have loaded the TDE master key into database memory or established a connection to the HSM. For more information, see the *open_keystore* clause of `ADMINISTER KEY MANAGEMENT` .

tablespace_encryption_spec

Use `USING` '*`encrypt_algorithm`*' to specify the encryption algorithm.

Valid algorithms are `AES256`, `AES192`, `AES128`, and `3DES168`.

Specify '*cipher_mode*' to determine how the TDE encrypted tablespace uses the tablespace key to encrypt data blocks. You can specify XTS (an XEX-based mode with ciphertext stealing mode) only with the encyrption algorithms AES128 and AES256. For AES192 use CFB.

If you set the COMPATIBLE initialization parameter to 12.2 or higher, then the following algorithms are also valid: ARIA128, ARIA192, ARIA256, GOST256, and SEED128.

If you omit this clause, then the database uses AES128.

Starting with Oracle Database 23ai, the Transparent Data Encryption (TDE) decryption libraries for the GOST and SEED algorithms are deprecated, and encryption to GOST and SEED are desupported.

GOST 28147-89 has been deprecated by the Russian government, and SEED has been deprecated by the South Korean government. If you need South Korean government-approved TDE cryptography, then use ARIA instead. If you are using GOST 28147-89, then you must decrypt and encrypt with another supported TDE algorithm. The decryption algorithms for GOST 28147-89 and SEED are included in Oracle Database 23ai, but are deprecated, and the GOST encryption algorithm is desupported with Oracle Database 23ai. If you are using GOST or SEED for TDE encryption, then Oracle recommends that you decrypt and encrypt with another algorithm before upgrading to Oracle Database 23ai. However, with the exception of the HP Itanium platform, the GOST and SEED decryption libraries are available with Oracle Database 23ai, so you can also decrypt after upgrading.

See Also:

- "Creating an Encrypted Tablespace: Example"
- *Encryption Conversions for Tablespaces and Databases*

default_tablespace_params

The DEFAULT clause lets you specify default parameters for the tablespace.

default_table_compression

Use this clause to specify default compression of data for all tables created in the tablespace. This clause is not valid for a temporary tablespace. The subclauses of this clause have the same semantics as they have for the *table_compression* clause of the CREATE TABLE statement, with one exception: The COMPRESS FOR OLTP clause here is equivalent to the ROW STORE COMPRESS ADVANCED clause of CREATE TABLE. Refer to the *table_compression* clauses of CREATE TABLE for the full semantics of these subclauses.

default_index_compression

Use this clause to specify default compression of data for all indexes created in the tablespace. This clause is not valid for a temporary tablespace. The subclauses of this clause have the same semantics as they have for the *advanced_index_compression* clause of the CREATE INDEX statement. Refer to the advanced_index_compression clause of CREATE INDEX for the full semantics of these subclauses.

inmemory_clause

Use the *inmemory_clause* to specify the default In-Memory Column Store (IM column store) settings for all tables and materialized views created in the tablespace. This clause is not valid for a temporary tablespace.

- Specify INMEMORY to enable all tables and materialized views for the IM column store.

  You can optionally use the *inmemory_attributes* clause to specify how the table or materialized view data is stored in the IM column store. The *inmemory_attributes* clause has the same semantics in CREATE TABLE and CREATE TABLESPACE. Refer to the *inmemory_attributes* clause of CREATE TABLE for the full semantics of this clause.
- Specify NO INMEMORY to disable all tables and materialized views for the IM column store. This is the default.

ilm_clause

Use the *ilm_clause* to specify default Automatic Data Optimization settings for all tables created in the tablespace. This clause is not valid for a temporary tablespace. Refer to the *ilm_clause* of CREATE TABLE for the full semantics of this clause.

storage_clause

Use the *storage_clause* to specify storage parameters for all objects created in the tablespace. This clause is not valid for a temporary tablespace or a locally managed tablespace. For a dictionary-managed tablespace, you can specify the following storage parameters with this clause: ENCRYPT, INITIAL, NEXT, MINEXTENTS, MAXEXTENTS, MAXSIZE, and PCTINCREASE. Refer to *storage_clause* for more information.

Note:

The ENCRYPT clause of the *storage_clause* is supported for backward compatibility. However, beginning with Oracle Database 12*c* Release 2 (12.2), you can instead specify ENCRYPT in the *tablespace_encryption_clause*. Refer to *tablespace_encryption_clause* for more information.

See Also:

"Creating Basic Tablespaces: Examples"

ONLINE | OFFLINE Clauses

Use these clauses to determine whether the tablespace is online or offline. This clause is not valid for a temporary tablespace.

ONLINE

Specify ONLINE to make the tablespace available immediately after creation to users who have been granted access to the tablespace. This is the default.

OFFLINE

Specify `OFFLINE` to make the tablespace unavailable immediately after creation.

The data dictionary view `DBA_TABLESPACES` indicates whether each tablespace is online or offline.

extent_management_clause

The *`extent_management_clause`* lets you specify how the extents of the tablespace will be managed.

> **Note:**
>
> After you have specified extent management with this clause, you can change extent management only by migrating the tablespace.

- `AUTOALLOCATE` specifies that the tablespace is system managed. Users cannot specify an extent size. You cannot specify `AUTOALLOCATE` for a temporary tablespace.
- `UNIFORM` specifies that the tablespace is managed with uniform extents of `SIZE` bytes.The default `SIZE` is 1 megabyte. All extents of temporary tablespaces are of uniform size, so this keyword is optional for a temporary tablespace. However, you must specify `UNIFORM` in order to specify `SIZE`. You cannot specify `UNIFORM` for an undo tablespace.

If you do not specify `AUTOALLOCATE` or `UNIFORM`, then the default is `UNIFORM` for temporary tablespaces and `AUTOALLOCATE` for all other types of tablespaces.

If you do not specify the *`extent_management_clause`*, then Oracle Database interprets the `MINIMUM EXTENT` clause and the `DEFAULT` *`storage_clause`* to determine extent management.

> **Note:**
>
> The `DICTIONARY` keyword is deprecated. It is still supported for backward compatibility. However, Oracle recommends that you create locally managed tablespaces. Locally managed tablespaces are much more efficiently managed than dictionary-managed tablespaces. The creation of new dictionary-managed tablespaces is scheduled for desupport.

> **See Also:**
>
> *Oracle Database Concepts* for a discussion of locally managed tablespaces

Restrictions on Extent Management

Extent management is subject to the following restrictions:

- A permanent locally managed tablespace can contain only permanent objects. If you need a locally managed tablespace to store temporary objects, for example, if you will assign it as a user's temporary tablespace, then use the *`temporary_tablespace_clause`*.

- If you specify this clause, then you cannot specify `DEFAULT` *`storage_clause`*, `MINIMUM EXTENT`, or the *`temporary_tablespace_clause`*.

See Also:

Oracle Database Administrator's Guide for information on changing extent management by migrating tablespaces and "Creating a Locally Managed Tablespace: Example"

segment_management_clause

The *`segment_management_clause`* is relevant only for permanent, locally managed tablespaces. It lets you specify whether Oracle Database should track the used and free space in the segments in the tablespace using free lists or bitmaps. This clause is not valid for a temporary tablespace.

AUTO

Specify `AUTO` if you want the database to manage the free space of segments in the tablespace using a bitmap. If you specify `AUTO`, then the database ignores any specification for `PCTUSED`, `FREELIST`, and `FREELIST GROUPS` in subsequent storage specifications for objects in this tablespace. This setting is called **automatic segment-space management** and is the default.

MANUAL

Specify `MANUAL` if you want the database to manage the free space of segments in the tablespace using free lists. Oracle strongly recommends that you do not use this setting and that you create tablespaces with automatic segment-space management.

To determine the segment management of an existing tablespace, query the `SEGMENT_SPACE_MANAGEMENT` column of the `DBA_TABLESPACES` or `USER_TABLESPACES` data dictionary view.

Note:

If you specify `AUTO` segment management, then:

- If you set extent management to `LOCAL UNIFORM`, then you must ensure that each extent contains at least 5 database blocks.
- If you set extent management to `LOCAL AUTOALLOCATE`, and if the database block size is 16K or greater, then Oracle manages segment space by creating extents with a minimum size of 5 blocks rounded up to 64K.

Restrictions on Automatic Segment-Space Management

This clause is subject to the following restrictions:

- You can specify this clause only for a permanent, locally managed tablespace.
- You cannot specify this clause for the `SYSTEM` tablespace.

See Also:

- *Oracle Automatic Storage Management Administrator's Guide* for information on automatic segment-space management and when to use it
- *Oracle Database Reference* for information on the data dictionary views
- "Specifying Segment Space Management for a Tablespace: Example"

flashback_mode_clause

Use this clause in conjunction with the `ALTER DATABASE FLASHBACK` clause to specify whether the tablespace can participate in `FLASHBACK DATABASE` operations. This clause is useful if you have the database in `FLASHBACK` mode but you do not want Oracle Database to maintain Flashback log data for this tablespace.

This clause is not valid for temporary or undo tablespaces.

FLASHBACK ON

Specify `FLASHBACK ON` to put the tablespace in `FLASHBACK` mode. Oracle Database will save Flashback log data for this tablespace and the tablespace can participate in a `FLASHBACK DATABASE` operation. If you omit the *`flashback_mode_clause`*, then `FLASHBACK ON` is the default.

FLASHBACK OFF

Specify `FLASHBACK OFF` to take the tablespace out of `FLASHBACK` mode. Oracle Database will not save any Flashback log data for this tablespace. You must take the data files in this tablespace offline or drop them prior to any subsequent `FLASHBACK DATABASE` operation. Alternatively, you can take the entire tablespace offline. In either case, the database does not drop existing Flashback logs.

Note:

The `FLASHBACK` mode of a tablespace is independent of the `FLASHBACK` mode of an individual table.

See Also:

- *Oracle Database Backup and Recovery User's Guide* for information on Oracle Flashback Database
- ALTER DATABASE and FLASHBACK DATABASE for information on setting the `FLASHBACK` mode of the entire database and reverting the database to an earlier version
- FLASHBACK TABLE and *flashback_query_clause*

lost_write_protection

Specify the `lost_write_protection` clause to create a storage area for lost write records. This storage area or shadow tablespace must be created, before you can enable lost write protection on datafiles and databases.

You may create as many shadow tablespaces as you need, and name them as you would any other tablespace.

Example: Create a Shadow Tablespace in a Database

This example creates the shadow tablespace `sh_lwp1` for lost write protection:

```
CREATE BIGFILE TABLESPACE sh_lwp1 DATAFILE sh_lwp1.df SIZE 10M BLOCKSIZE 8K
  LOST WRITE PROTECTION;
```

To enable lost write protection on datafiles and databases, you must specify the `lost_write_protection` clause with the `ALTER TABLESPACE`, `ALTER DATABASE`, and `ALTER PLUGGABLE DATABASE` statements.

undo_tablespace_clause

Specify `UNDO` to create an undo tablespace. When you run the database in automatic undo management mode, Oracle Database manages undo space using the undo tablespace instead of rollback segments. This clause is useful if you are now running in automatic undo management mode but your database was not created in automatic undo management mode.

Oracle Database always assigns an undo tablespace when you start up the database in automatic undo management mode. If no undo tablespace has been assigned to this instance, then the database uses the `SYSTEM` rollback segment. You can avoid this by creating an undo tablespace, which the database will implicitly assign to the instance if no other undo tablespace is currently assigned.

The `DATAFILE` clause is described in "DATAFILE | TEMPFILE Clause".

extent_management_clause

It is unnecessary to specify the *`extent_management_clause`* when creating an undo tablespace, because undo tablespaces must be locally managed tablespaces that use `AUTOALLOCATE` extent management. If you do specify this clause, then you must specify `EXTENT MANAGEMENT LOCAL` or `EXTENT MANAGEMENT LOCAL AUTOALLOCATE`, both of which are the same as omitting this clause. Refer to *extent_management_clause* for the full semantics of this clause.

tablespace_retention_clause

This clause is valid only for undo tablespaces.

- `RETENTION GUARANTEE` specifies that Oracle Database should preserve unexpired undo data in all undo segments of *`tablespace`* even if doing so forces the failure of ongoing operations that need undo space in those segments. This setting is useful if you need to issue an Oracle Flashback Query or an Oracle Flashback Transaction Query to diagnose and correct a problem with the data.
- `RETENTION NOGUARANTEE` returns the undo behavior to normal. Space occupied by unexpired undo data in undo segments can be consumed if necessary by ongoing transactions. This is the default.

tablespace_encryption_clause

This clause has the same semantics for undo tablespaces as for permanent tablespaces. Refer to *tablespace_encryption_clause* in the documentation on permanent tablespaces for full information.

Restrictions on Undo Tablespaces

Undo tablespaces are subject to the following restrictions:

- You cannot create database objects in this tablespace. It is reserved for system-managed undo data.
- The only clauses you can specify for an undo tablespace are the `DATAFILE` clause, the *`tablespace_retention_clause`*, the *`tablespace_encryption_clause`*, and the *`extent_management_clause`* to specify local `AUTOALLOCATE` extent management. You cannot specify local `UNIFORM` extent management or dictionary extent management using the *`extent_management_clause`*. All undo tablespaces are created permanent, read/write, and in logging mode. Values for `MINIMUM EXTENT` and `DEFAULT STORAGE` are system generated.

See Also:

- *Oracle Database Administrator's Guide* for information on automatic undo management and undo tablespaces and *Oracle Database Reference* for information on the `UNDO_MANAGEMENT` parameter
- CREATE DATABASE for information on creating an undo tablespace during database creation, and ALTER TABLESPACE and DROP TABLESPACE
- "Creating an Undo Tablespace: Example"

temporary_tablespace_clause

Use this clause to create a temporary tablespace, which is an allocation of space in the database that can contain transient data that persists only for the duration of a session. This transient data cannot be recovered after process or instance failure.

The transient data can be user-generated schema objects such as temporary tables or system-generated data such as temp space used by hash joins and sort operations. When a temporary tablespace, or a tablespace group of which this tablespace is a member, is assigned to a particular user, then Oracle Database uses the tablespace for sorting operations in transactions initiated by that user.

You can create two types of temporary tablespaces:

- You can create a shared temporary tablespace by specifying the `TEMPORARY TABLESPACE` clause. A shared temporary tablespace stores temp files on shared disk, so that the temporary space is accessible to all database instances. 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.
- Starting with Oracle Database 12*c* Release 2 (12.2), you can create a local temporary tablespace by specifying the `LOCAL TEMPORARY TABLESPACE` clause. Local temporary tablespaces are useful in an Oracle Clusterware environment. They store a separate,

nonshared temp files 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.

TEMPFILE

The `TEMPFILE` clause is described in "DATAFILE | TEMPFILE Clause".

tablespace_group_clause

This clause is relevant only for temporary tablespaces. Use this clause to determine whether *`tablespace`* is a member of a tablespace group. A tablespace group lets you assign multiple temporary tablespaces to a single user and increases the addressability of temporary tablespaces.

- Specify a group name to indicate that *`tablespace`* is a member of this tablespace group. The group name cannot be the same as *`tablespace`* or any other existing tablespace. If the tablespace group already exists, then Oracle Database adds the new tablespace to that group. If the tablespace group does not exist, then the database creates the group and adds the new tablespace to that group.
- Specify an empty string (' ') to indicate that *`tablespace`* is not a member of any tablespace group.

Restriction on Tablespace Groups

Tablespace groups support only shared temporary tablespaces. You cannot add a local temporary tablespace to a tablespace group.

extent_management_clause

The *`extent_management_clause`* is described in *extent_management_clause* .

tablespace_encryption_clause

This clause has the same semantics for temporary tablespaces as for permanent tablespaces. Refer to *tablespace_encryption_clause* in the documentation on permanent tablespaces for full information.

See Also:

- ALTER TABLESPACE and "Adding a Temporary Tablespace to a Tablespace Group: Example" for information on adding a tablespace to a tablespace group
- CREATE USER for information on assigning a temporary tablespace to a user
- *Oracle Database Administrator's Guide* for more information on tablespace groups

Restrictions on Temporary Tablespaces

The data stored in temporary tablespaces persists only for the duration of a session. Therefore, only a subset of the `CREATE TABLESPACE` clauses are relevant for temporary tablespaces. The only clauses you can specify for a temporary tablespace are the `TEMPFILE` clause, the *`tablespace_group_clause`*, the *`extent_management_clause`*, and the *`tablespace_encryption_clause`*.

Examples

These examples assume that your database is using 8K blocks.

Creating a Bigfile Tablespace: Example

The following example creates a bigfile tablespace `bigtbs_01` with a data file `bigtbs_f1.dbf` of 20 MB:

```
CREATE BIGFILE TABLESPACE bigtbs_01
  DATAFILE 'bigtbs_f1.dbf'
  SIZE 20M AUTOEXTEND ON;
```

Creating an Undo Tablespace: Example

The following example creates a 10 MB undo tablespace `undots1`:

```
CREATE UNDO TABLESPACE undots1
   DATAFILE 'undotbs_1a.dbf'
   SIZE 10M AUTOEXTEND ON
   RETENTION GUARANTEE;
```

Creating a Temporary Tablespace: Example

This statement shows how the temporary tablespace that serves as the default temporary tablespace for database users in the sample database was created:

```
CREATE TEMPORARY TABLESPACE temp_demo
   TEMPFILE 'temp01.dbf' SIZE 5M AUTOEXTEND ON;
```

Assuming that the default database block size is 2K, and that each bit in the map represents one extent, then each bit maps 2,500 blocks.

The following example sets the default location for data file creation and then creates a tablespace with an Oracle-managed temp file in the default location. The temp file is 100 M and is autoextensible with unlimited maximum size. These are the default values for Oracle Managed Files:

```
ALTER SYSTEM SET DB_CREATE_FILE_DEST = '$ORACLE_HOME/rdbms/dbs';

CREATE TEMPORARY TABLESPACE tbs_05;
```

Adding a Temporary Tablespace to a Tablespace Group: Example

The following statement creates the `tbs_temp_02` temporary tablespace as a member of the `tbs_grp_01` tablespace group. If the tablespace group does not already exist, then Oracle Database creates it during execution of this statement:

```
CREATE TEMPORARY TABLESPACE tbs_temp_02
  TEMPFILE 'temp02.dbf' SIZE 5M AUTOEXTEND ON
  TABLESPACE GROUP tbs_grp_01;
```

Creating Basic Tablespaces: Examples

This statement creates a tablespace named `tbs_01` with one data file:

```
CREATE TABLESPACE tbs_01
   DATAFILE 'tbs_f2.dbf' SIZE 40M
   ONLINE;
```

This statement creates tablespace `tbs_03` with one data file and allocates every extent as a multiple of 500K:

```
CREATE TABLESPACE tbs_03
   DATAFILE 'tbs_f03.dbf' SIZE 20M
   LOGGING;
```

Enabling Autoextend for a Tablespace: Example

This statement creates a tablespace named `tbs_02` with one data file. When more space is required, 500 kilobyte extents will be added up to a maximum size of 100 megabytes:

```
CREATE TABLESPACE tbs_02
   DATAFILE 'diskb:tbs_f5.dbf' SIZE 500K REUSE
   AUTOEXTEND ON NEXT 500K MAXSIZE 100M;
```

Creating a Locally Managed Tablespace: Example

The following statement assumes that the database block size is 2K.

```
CREATE TABLESPACE tbs_04 DATAFILE 'file_1.dbf' SIZE 10M
   EXTENT MANAGEMENT LOCAL UNIFORM SIZE 128K;
```

This statement creates a locally managed tablespace in which every extent is 128K and each bit in the bit map describes 64 blocks.

The following statement creates a locally managed tablespace with uniform extents and shows an example of a table stored in that tablespace:

```
CREATE TABLESPACE lmt1 DATAFILE 'lmt_file2.dbf' SIZE 100m REUSE
  EXTENT MANAGEMENT LOCAL UNIFORM SIZE 1M;

CREATE TABLE lmt_table1 (col1 NUMBER, col2 VARCHAR2(20))
  TABLESPACE lmt1 STORAGE (INITIAL 2m);
```

The initial segment size of the table is 2M.

The following example creates a locally managed tablespace without uniform extents:

```
CREATE TABLESPACE lmt2 DATAFILE 'lmt_file3.dbf' SIZE 100m REUSE
  EXTENT MANAGEMENT LOCAL;

CREATE TABLE lmt_table2 (col1 NUMBER, col2 VARCHAR2(20))
  TABLESPACE lmt2 STORAGE (INITIAL 2m MAXSIZE 100m);
```

The initial segment size of the table is 2M. Oracle Database determines the size of each extent and the total number of extents allocated to satisfy the initial segment size. The segment's maximum size is limited to 100M.

Creating an Encrypted Tablespace: Example

In the following example, the first statement enables encryption for the database by opening the wallet. The second statement creates an encrypted tablespace.

```
ALTER SYSTEM SET ENCRYPTION WALLET OPEN IDENTIFIED BY "wallet_password";

CREATE TABLESPACE encrypt_ts
  DATAFILE '$ORACLE_HOME/dbs/encrypt_df.dbf' SIZE 1M
  ENCRYPTION USING 'AES256' ENCRYPT;
```

The following example creates a tablespace `encts2` using the encryption algoritm `AES256` and cipher mode `XTS`:

```
CREATE TABLESPACE encts2 DATAFILE 'encts2.f' SIZE 1G ENCRYPTION USING AES256 MODE 'XTS'
ENCRYPT;
```

Specifying Segment Space Management for a Tablespace: Example

The following example creates a tablespace with automatic segment-space management:

```
CREATE TABLESPACE auto_seg_ts DATAFILE 'file_2.dbf' SIZE 1M
   EXTENT MANAGEMENT LOCAL
   SEGMENT SPACE MANAGEMENT AUTO;
```

Creating Oracle Managed Files: Examples

The following example sets the default location for data file creation and creates a tablespace with a data file in the default location. The data file is 100M and is autoextensible with an unlimited maximum size:

```
ALTER SYSTEM SET DB_CREATE_FILE_DEST = '$ORACLE_HOME/rdbms/dbs';

CREATE TABLESPACE omf_ts1;
```

The following example creates a tablespace with an Oracle-managed data file of 100M that is not autoextensible:

```
CREATE TABLESPACE omf_ts2 DATAFILE AUTOEXTEND OFF;
```

CREATE TABLESPACE SET

Note:

This SQL statement is valid only if you are using Oracle Sharding. For more information on Oracle Sharding, refer to *Oracle Database Administrator's Guide*.

Purpose

Use the `CREATE TABLESPACE SET` statement to create a tablespace set. A tablespace set can be used in a sharded database as a logical storage unit for one or more sharded tables and indexes.

A tablespace set consists of multiple tablespaces distributed across shards in a shardspace. The database automatically creates the tablespaces in a tablespace set. The number of tablespaces is determined automatically and is equal to the number of chunks in the corresponding shardspace.

All tablespaces in a tablespace set are permanent bigfile tablespaces; a tablespace set does not contain `SYSTEM`, undo, or temporary tablespaces. The database automatically creates one data file for each tablespace. All tablespaces in a tablespace set share the same attributes. You can modify attributes for all tablespaces in a tablespace set with the `ALTER TABLESPACE SET` statement.

See Also:

ALTER TABLESPACE SET and DROP TABLESPACE SET

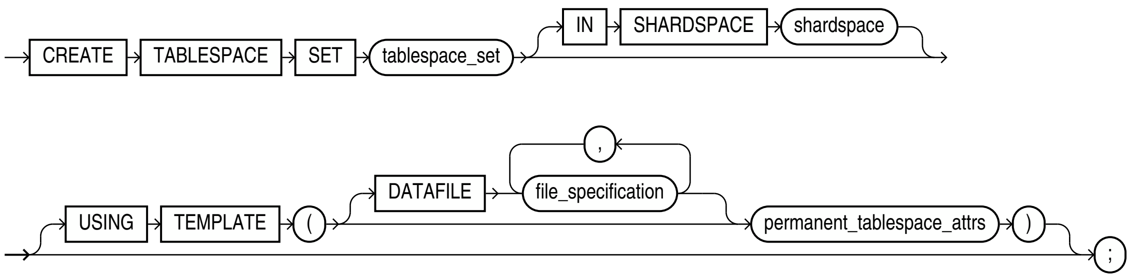
Prerequisites

You must be connected to a shard catalog database as an SDB user.

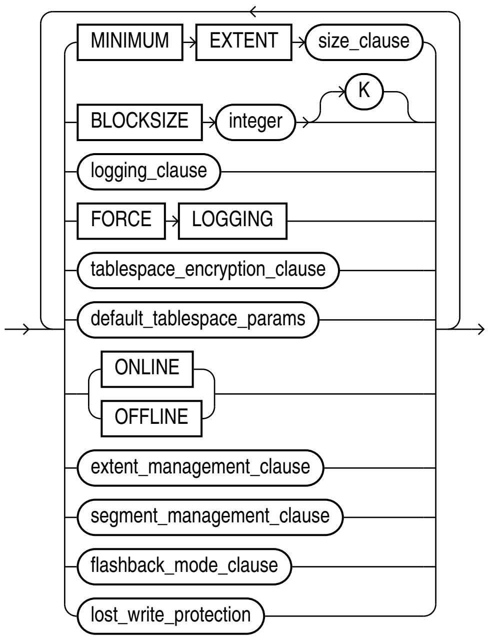
You must have the `CREATE TABLESPACE` system privilege.

Syntax

***create_tablespace_set*::=**



***permanent_tablespace_attrs*::=**



(*file_specification*::=, See the following clauses of `CREATE TABLESPACE`: *logging_clause*::=, *tablespace_encryption_clause*::=, *default_tablespace_params*::=, *extent_management_clause*::=, *segment_management_clause*::=, *flashback_mode_clause*::=)

Semantics

tablespace_set

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

IN SHARDSPACE

Specify this clause if you are using composite sharding. For *shardspace_name*, specify the name of the shardspace in which the tablespace set is to be created.

Omit this clause if you are using system-managed sharding. In this case, the tablespace set is created in the default shardspace for the sharded database.

USING TEMPLATE

The USING TEMPLATE clause allows you to specify attributes for the tablespaces in the tablespace set.

The DATAFILE and *permanent_tablespace_attrs* clauses have the same semantics here as for the CREATE TABLESPACE statement, with the following exceptions:

- For the DATAFILE *file_specification* clause, you can specify only the SIZE clause and the *autoextend_clause*.
- You cannot specify the MINIMUM EXTENT *size_clause*.
- For the *segment_management_clause*, you can specify only SEGMENT SPACE MANAGEMENT AUTO. The MANUAL setting is not supported.

See Also:

file_specification and *permanent_tablespace_attrs* in the documentation on CREATE TABLESPACE for the full semantics of these clauses

Examples

Creating a Tablespace Set: Example

The following statement creates tablespace set ts1:

```
CREATE TABLESPACE SET ts1
  IN SHARDSPACE sgr1
  USING TEMPLATE
  ( DATAFILE SIZE 100m
    EXTENT MANAGEMENT LOCAL
    SEGMENT SPACE MANAGEMENT AUTO
  );
```

CREATE TRIGGER

Purpose

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

Use the `CREATE TRIGGER` statement to create a **database trigger**, which is:

- A stored PL/SQL block associated with a table, a schema, or the database or
- An anonymous PL/SQL block or a call to a procedure implemented in PL/SQL or Java

Oracle Database automatically executes a trigger when specified conditions occur.

See Also:

ALTER TRIGGER and DROP TRIGGER

Prerequisites

To create a trigger in your own schema on a table in your own schema or on your own schema (`SCHEMA`), you must have the `CREATE TRIGGER` system privilege.

To create a trigger in any schema on a table in any schema, or on another user's schema (schema.`SCHEMA`), you must have the schema level `CREATE ANY TRIGGER` privilege both in the schema where the trigger is created and in the schema where table resides, or you must have the `CREATE ANY TRIGGER` system privilege.

In addition to the preceding privileges, to create a trigger on `DATABASE`, you must have the `ADMINISTER DATABASE TRIGGER` system privilege.

To create a trigger on a pluggable database (PDB), the current container must be that PDB and you must have the `ADMINISTER DATABASE TRIGGER` system privilege. For information about PDBs, see *Oracle Database Administrator's Guide*.

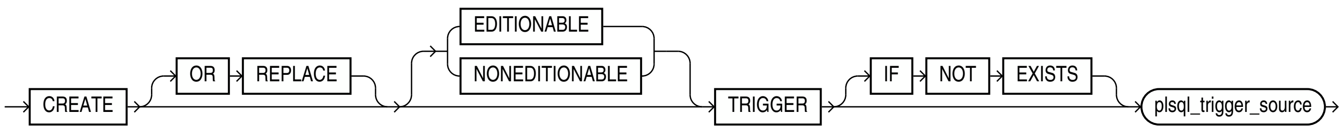
In addition to the preceding privileges, to create a crossedition trigger, you must be enabled for editions. For information about enabling editions for a user, see *Oracle Database Development Guide*.

If the trigger issues SQL statements or calls procedures or functions, then the owner of the trigger must have the privileges necessary to perform these operations. These privileges must be granted directly to the owner rather than acquired through roles.

Syntax

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

***create_trigger*::=**



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

Semantics

OR REPLACE

Specify `OR REPLACE` to re-create the trigger if it already exists. Use this clause to change the definition of an existing trigger without first dropping it.

IF NOT EXISTS

Specifying `IF NOT EXISTS` has the following effects:

- If the trigger does not exist, a new trigger is created at the end of the statement.
- If the trigger exists, this is the trigger 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 trigger is an editioned or noneditioned object if editioning is enabled for the schema object type `TRIGGER` in *`schema`*. The default is `EDITIONABLE`. For information about editioned and noneditioned objects, see *Oracle Database Development Guide*.

Restriction on NONEDITIONABLE

You cannot specify `NONEDITIONABLE` for a crossedition trigger.

plsql_trigger_source

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

CREATE TRUE CACHE

Purpose

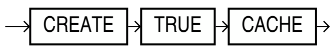
Use `CREATE TRUE CACHE` to internally create and initialize the run-time management files required for True Cache, and also open True Cache for service. The set of run-time management files for True Cache operation include controlfile, `SPFILE` and tempfiles.

Prerequisites

- You must set the initialization parameter `TRUE_CACHE` to `TRUE` to be in a True Cache environment.
- You must start the database in `NOMOUNT` mode.

> **See Also:**
>
> *True Cache User's Guide*

Syntax



Semantics

To drop True Cache use DROP DATABASE.

CREATE TYPE

Purpose

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

> **Note:**
>
> Starting with Oracle Database 23ai, the SQLJ method of embedding SQL statements in Java code is deprecated. Oracle recommends using the Java Database Connectivity (JDBC) APIs instead of SQLJ.

Use the `CREATE TYPE` statement to create the specification of an **object type**, a **SQLJ object type**, a named varying array (**varray**), a **nested table type**, or an **incomplete object type**. You create object types with the `CREATE TYPE` and the `CREATE TYPE BODY` statements. The `CREATE TYPE` statement specifies the name of the object type, its attributes, methods, and other properties. The `CREATE TYPE BODY` statement contains the code for the methods that implement the type.

Note:

- If you create an object type for which the type specification declares only attributes but no methods, then you need not specify a type body.
- If you create a SQLJ object type, then you cannot specify a type body. The implementation of the type is specified as a Java class.

An **incomplete type** is a type created by a forward type definition. It is called "incomplete" because it has a name but no attributes or methods. It can be referenced by other types, and so can be used to define types that refer to each other. However, you must fully specify the type before you can use it to create a table or an object column or a column of a nested table type.

See Also:

- CREATE TYPE BODY for information on creating the member methods of a type
- *Oracle Database Object-Relational Developer's Guide* for more information about objects, incomplete types, varrays, and nested tables

Prerequisites

To create a type in your own schema, you must have the `CREATE TYPE` system privilege. To create a type in another user's schema, you must have the `CREATE ANY TYPE` system privilege. You can acquire these privileges explicitly or be granted them through a role.

To create a subtype, you must have the `UNDER ANY TYPE` system privilege or the `UNDER` object privilege on the supertype.

The owner of the type must be explicitly granted the `EXECUTE` object privilege in order to access all other types referenced within the definition of the type, or the type owner must be granted the `EXECUTE ANY TYPE` system privilege. The owner cannot obtain these privileges through roles.

If the type owner intends to grant other users access to the type, then the owner must be granted the `EXECUTE` object privilege on the referenced types with the `GRANT OPTION` or the `EXECUTE ANY TYPE` system privilege with the `ADMIN OPTION`. Otherwise, the type owner has insufficient privileges to grant access on the type to other users.

User-Defined Data Types Declared as Non-Persistable Data Types

You can specify a user-defined data type as non-persistable when creating the data type. Instances of non-persistable types cannot persist on disk. Perisistable data types include the following:

- ANSI-supported data types, for example `NUMERIC`, `DECIMAL`, `REAL`.
- Oracle built-in data types, for example `NUMBER`, `VARCHAR2`, `TIMESTAMP`.
- Oracle-supplied data types, for example `ANYDATA`, `XML Type`, `ORDImage`.

Rules For SQL User-Defined Data Types

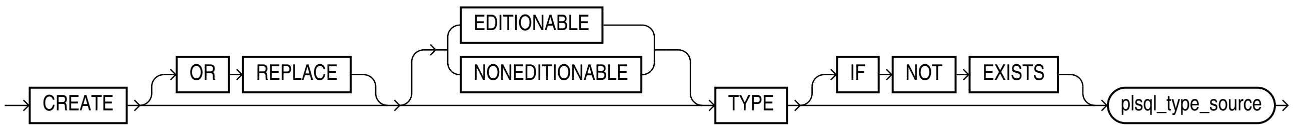
- A persistable type cannot have attributes or elements of non-persistable types.
- A non-persistable type can have attributes or elements of both persistable and non-persistable types.
- A sub-type must inherit the persistence property from its super type.
- A `REF` type is persistable and can hold references only to objects of persistable types.
- You cannot persist instances of non-persistable types on disk. If you create a table with a type that has been declared as non-persistable, the `CREATE TABLE` statement will fail. The following operations will likewise fail:
  - Create or alter a relational table with columns of non-persistable types.
  - Create an object table with columns of non-persistable types.
  - Store instances of non-persistable types in an `ANYDATA` instance which is persisted on disk.

You can specify unique PL/SQL attributes in the `CREATE TYPE` statement in the PL/SQL context only.

Syntax

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

create_type::=



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

Semantics

OR REPLACE

Specify `OR REPLACE` to re-create the type if it already exists. Use this clause to change the definition of an existing type without first dropping it.

Users previously granted privileges on the re-created object type can use and reference the object type without being granted privileges again.

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

IF NOT EXISTS

Specifying `IF NOT EXISTS` has the following effects:

- If the type does not exist, a new type is created at the end of the statement.
- If the type exists, this is the type 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 type is an editioned or noneditioned object if editioning is enabled for the schema object type `TYPE` in *`schema`*.The default is `EDITIONABLE`. For information about editioned and noneditioned objects, see *Oracle Database Development Guide*.

plsql_type_source

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

CREATE TYPE BODY

Purpose

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

Use the `CREATE TYPE BODY` to define or implement the member methods defined in the object type specification. You create object types with the `CREATE TYPE` and the `CREATE TYPE BODY` statements. The `CREATE TYPE` statement specifies the name of the object type, its attributes, methods, and other properties. The `CREATE TYPE BODY` statement contains the code for the methods that implement the type.

For each method specified in an object type specification for which you did not specify the *`call_spec`*, you must specify a corresponding method body in the object type body.

Note:

If you create a SQLJ object type, then specify it as a Java class.

See Also:

- CREATE TYPE for information on creating a type specification
- ALTER TYPE for information on modifying a type specification

Prerequisites

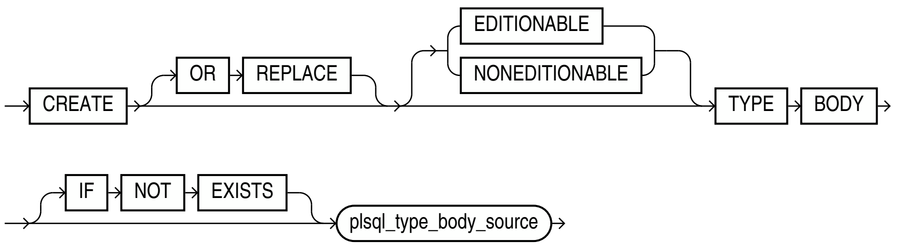
Every member declaration in the `CREATE TYPE` specification for object types must have a corresponding construct in the `CREATE TYPE` or `CREATE TYPE BODY` statement.

To create or replace a type body in your own schema, you must have the `CREATE TYPE` or the `CREATE ANY TYPE` system privilege. To create an object type in another user's schema, you must have the `CREATE ANY TYPE` system privilege. To replace an object type in another user's schema, you must have the `DROP ANY TYPE` system privilege.

Syntax

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

create_type_body::=



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

Semantics

OR REPLACE

Specify `OR REPLACE` to re-create the type body if it already exists. Use this clause to change the definition of an existing type body without first dropping it.

Users previously granted privileges on the re-created object type body can use and reference the object type body without being granted privileges again.

You can use this clause to add new member subprogram definitions to specifications added with the `ALTER TYPE ... REPLACE` statement.

IF NOT EXISTS

Specifying `IF NOT EXISTS` has the following effects:

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

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

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

[EDITIONABLE | NONEDITIONABLE]

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

plsql_type_body_source

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

CREATE USER

Purpose

Use the `CREATE USER` statement to create and configure a database **user**, which is an account through which you can log in to the database, and to establish the means by which Oracle Database permits access by the user.

You can issue this statement in an Oracle Automatic Storage Management (Oracle ASM) cluster to add a user and password combination to the password file that is local to the Oracle ASM instance of the current node. Each node's Oracle ASM instance can use this statement to update its own password file. The password file itself must have been created by the `ORAPWD` utility.

You can enable a user to connect to the database through a proxy application or application server. For syntax and discussion, refer to ALTER USER .

Prerequisites

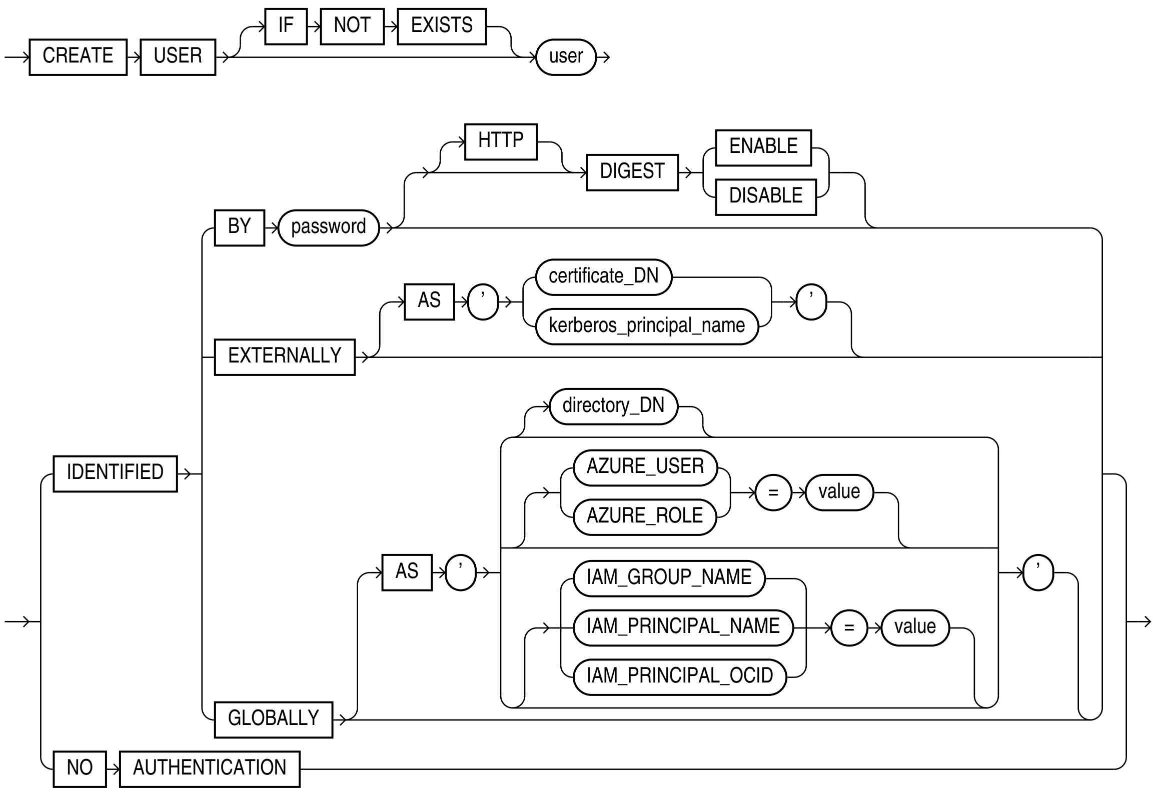
You must have the `CREATE USER` system privilege. When you create a user with the `CREATE USER` statement, the user's privilege domain is empty. To log on to Oracle Database, a user must have the `CREATE SESSION` system privilege. Therefore, after creating a user, you should grant the user at least the `CREATE SESSION` system privilege. Refer to GRANT for more information.

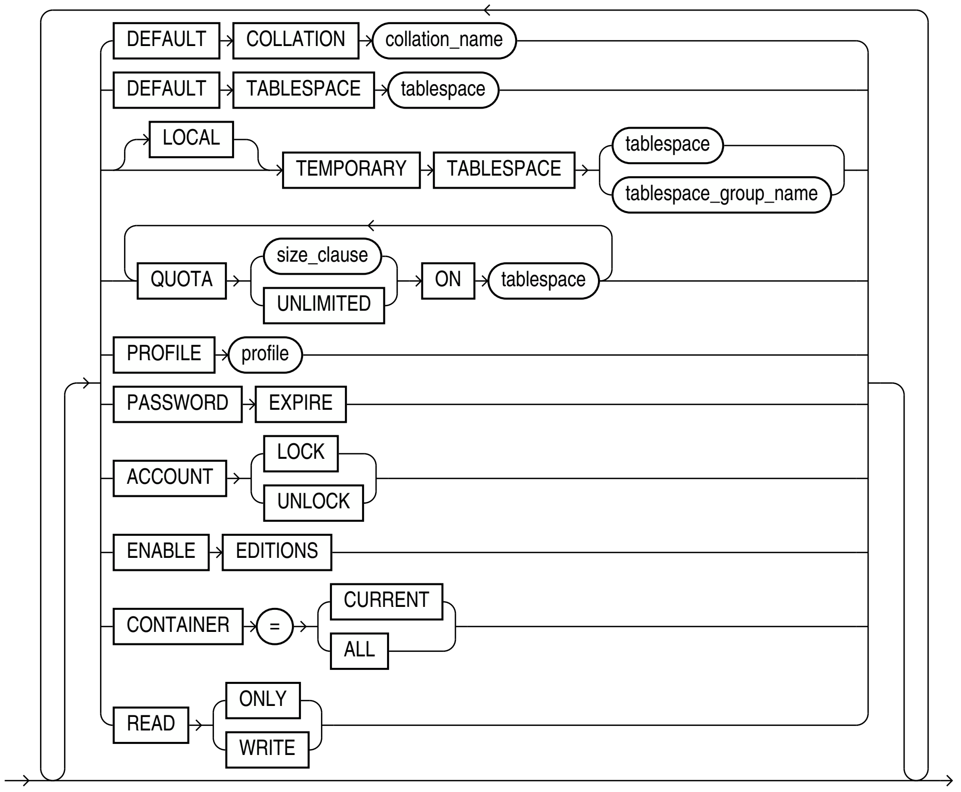
Only a user authenticated `AS SYSASM` can issue this command to modify the Oracle ASM instance password file.

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

Syntax

***create_user*::=**





(*size_clause*::=)

Semantics

IF NOT EXISTS

Specifying `IF NOT EXISTS` has the following effects:

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

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

user

Specify the name of the user to be created. This name can contain only characters from your database character set and must follow the rules described in the section "Database Object Naming Rules ". Oracle recommends that the user name contain at least one single-byte character regardless of whether the database character set also contains multibyte characters.

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

Note:

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

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

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

Note:

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

Note:

Oracle recommends that user names and passwords be encoded in ASCII or EBCDIC characters only, depending on your platform.

See Also:

"Creating a Database User: Example"

IDENTIFIED Clause

The `IDENTIFIED` clause lets you indicate how Oracle Database authenticates the user.

BY *password*

The `BY` *`password`* clause lets you creates a **local user** and indicates that the user must specify *`password`* to log on to the database. Passwords are case sensitive and their maximum length is 1024 bytes. Any subsequent `CONNECT` string used to connect this user to the database must specify the password using the same case (upper, lower, or mixed) that is used in this `CREATE USER` statement or a subsequent `ALTER USER` statement. Passwords can contain any single-byte, multibyte, or special characters, or any combination of these, from your database character set, with the exception of the double quotation mark (") and the return character . If a password starts with a non-alphabetic character, or contains a character other than an alphanumeric character, the underscore (_), dollar sign ($), or pound sign (#), then it must be enclosed in double quotation marks. Otherwise, enclosing a password in double quotation marks is optional.

See Also:

Oracle Database Security Guide for more information about case-sensitive passwords, password complexity, and other password guidelines

Passwords must follow the rules described in the section "Database Object Naming Rules ", unless you are using one of the three Oracle Database password complexity verification routines. These routines requires a more complex combination of characters than the normal naming rules permit. You implement these routines with the `UTLPWDMG.SQL` script, which is further described in *Oracle Database Security Guide*.

Note:

Oracle recommends that user names and passwords be encoded in ASCII or EBCDIC characters only, depending on your platform.

See Also:

Oracle Database Security Guide to for a detailed discussion of password management and protection

[HTTP] DIGEST Clause

This clause lets you `ENABLE` or `DISABLE` HTTP Digest Access Authentication for the user. The default is `DISABLE`.

The `HTTP` keyword is optional and is provided for semantic clarity.

Restriction on the [HTTP] DIGEST Clause

You cannot specify this clause for external or global users.

EXTERNALLY Clause

Specify `EXTERNALLY` to create an **external user.** Such a user must be authenticated by an external service, such as an operating system or a third-party service. In this case, Oracle Database relies on authentication by the operating system or third-party service to ensure that a specific external user has access to a specific database user.

The `IDENTIFIED EXTERNALLY` clause setting is unique to the user that is specified in the `CREATE USER` statement. This means that you cannot create another user with the same name used in a previous user creation statement.

The following example creates the external user `jsmith`:

```
CREATE USER jsmith IDENTIFIED EXTERNALLY AS "CN=foo,DNQ=123,SERIAL=234";
```

Now create another external user `tjones` with the same name `CN=foo,dnQualifier=123,SERIALNUMER=234` :

```
CREATE USER tjones IDENTIFIED EXTERNALLY AS "CN=foo,dnQualifier=123,SERIALNUMER=234";
```

The user `tjones` is not created because the command fails with the error: `User with same external name already exists` because the `CN=foo,dnQualifier=123,SerialNumber=234` setting has already been used. This happens because `dnq=` is converted to `dnqualifier=` and `serial=` is converted to `serialnumber=` internally, and therefore, `CN=foo,DNQ=123,SERIAL=234` and `CN=foo,dnQualifier=123,SerialNumber=234` are treated as the same user.

AS '*certificate_DN*'

This clause is required for and used for SSL-authenticated external users only. The *`certificate_DN`* is the distinguished name in the user's PKI certificate in the user's wallet. The maximum length of *`certificate_DN`* is 1024 characters.

AS '*kerberos_principal_name*'

This clause is required for and used for Kerberos-authenticated external users only. The maximum length of *`kerberos_principal_name`* is 1024 characters.

Note:

Oracle strongly recommends that you do not use `IDENTIFIED EXTERNALLY` with operating systems that have inherently weak login security.

Restriction on Creating External Users

Oracle ASM does not support the creation of external users.

See Also:

- *Oracle Database Enterprise User Security Administrator's Guide* for more information on externally identified users
- "Creating External Database Users: Examples"

GLOBALLY Clause

The `GLOBALLY` clause lets you create a **global user**. Such a user must be authorized by the enterprise directory service (Oracle Internet Directory).

The *`directory_DN`* string can take one of two forms:

- The X.509 name at the enterprise directory service that identifies this user. It should be of the form `CN=`*`username,other_attributes`*, where *`other_attributes`* is the rest of the user's distinguished name (DN) in the directory. This form uses the LDAP Data Interchange Format (LDIF) and creates a **private global schema**.
- A null string (' ') indicating that the enterprise directory service will map authenticated global users to this database schema with the appropriate roles. This form is the same as specifying the `GLOBALLY` keyword alone and creates a **shared global schema**.

The maximum length of *`directory_DN`* is 1024 characters.

You can control the ability of an application server to connect as the specified user and to activate that user's roles using the `ALTER USER` statement.

You can exclusively map an Oracle Database schema to a Microsoft Azure AD user using `GLOBALLY AS AZURE_USER`. You must log in to the Oracle Autonomous Database instance as a user who has been granted the `CREATE USER` or `ALTER USER` system privilege.

Example: Map Oracle Database Schema to a Microsoft Azure AD User

The example creates a new database schema user named *`peter_fitch`* and maps this user to an existing Azure AD user named *`peter.fitch@example.com`*:

```
CREATE USER peter_fitch IDENTIFIED GLOBALLY AS 'AZURE_USER=peter.fitch@example.com';
```

Example: Map a Shared Oracle Database Schema to an App Role

The example creates a new database global user account (schema) named *`dba_azure`* and maps it to an existing Azure AD application role named `AZURE_DBA`:

```
CREATE USER dba_azure IDENTIFIED GLOBALLY AS 'AZURE_ROLE=AZURE_DBA';
```

Restriction on Creating Global Users

Oracle ASM does not support the creation of global users.

> **See Also:**
>
> - *Oracle Database Security Guide* for more information on global users
> - *Authenticating and Authorizing Microsoft Azure Active Directory Users for Oracle Autonomous Databases*
> - ALTER USER
> - "Creating a Global Database User: Example"

NO AUTHENTICATION Clause

Use the `NO AUTHENTICATION` clause to create a schema that does not have a password and cannot be logged into. This is intended for schema only accounts and reduces maintenance by removing default passwords and any requirement to rotate the password.

DEFAULT COLLATION Clause

This clause lets you specify the default collation for the schema owned by the user. The default collation is assigned to tables, views, and materialized views that are subsequently created in the schema.

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

If you omit this clause, then the default collation for the schema owned by the user is set to the `USING_NLS_COMP` pseudo-collation.

You can override this clause and assign a different default collation to a particular table, materialized view, or view by specifying the `DEFAULT COLLATION` clause of the `CREATE` or `ALTER` statement for the table, materialized view, or view. You can also override the default collations of all schemas for the duration of a database session by setting the default collation for the session. See the DEFAULT_COLLATION clause of `ALTER SESSION` for more details.

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

DEFAULT TABLESPACE Clause

Specify the default tablespace for objects that are created in the user's schema. If you omit this clause, then the user's objects are stored in the database default tablespace. If no default tablespace has been specified for the database, then the user's objects are stored in the `SYSTEM` tablespace.

Restriction on Default Tablespaces

You cannot specify a locally managed temporary tablespace, including an undo tablespace, or a dictionary-managed temporary tablespace, as a user's default tablespace.

See Also:

- CREATE TABLESPACE for more information on tablespaces in general and undo tablespaces in particular
- *Oracle Database Security Guide* for more information on assigning default tablespaces to users

[LOCAL] TEMPORARY TABLESPACE Clause

Specify the tablespace or tablespace group for the user's temporary segments. If you omit this clause, then the user's temporary segments are stored in the database default temporary tablespace or, if none has been specified, in the `SYSTEM` tablespace.

- Specify *`tablespace`* to indicate the user's temporary tablespace. Specify `TEMPORARY TABLESPACE` to indicate a shared temporary tablespace. Specify `LOCAL TEMPORARY TABLESPACE` to indicate a local temporary tablespace. If you are connected to a CDB, then you can specify `CDB$DEFAULT` to use the CDB-wide default temporary tablespace.
- Specify *`tablespace_group_name`* to indicate that the user can save temporary segments in any tablespace in the tablespace group specified by *`tablespace_group_name`*. Local temporary tablespaces cannot be part of a tablespace group.

Restrictions on Temporary Tablespace

This clause is subject to the following restrictions:

- The tablespace must be a temporary tablespace and must have a standard block size.
- The tablespace cannot be an undo tablespace or a tablespace with automatic segment-space management.

See Also:

- *Oracle Database Administrator's Guide* for information about tablespace groups and *Oracle Database Security Guide* for information on assigning temporary tablespaces to users
- CREATE TABLESPACE for more information on undo tablespaces and segment management
- "Assigning a Tablespace Group: Example"

QUOTA Clause

Use the `QUOTA` clause to specify the maximum amount of space the user can allocate in the tablespace.

A `CREATE USER` statement can have multiple `QUOTA` clauses for multiple tablespaces.

`UNLIMITED` lets the user allocate space in the tablespace without bound.

The maximum amount of space that you can specify is 2 terabytes (TB). If you need more space, then specify `UNLIMITED`.

Restriction on the QUOTA Clause

You cannot specify this clause for a temporary tablespace.

See Also:

size_clause for information on that clause and *Oracle Database Security Guide* for more information on assigning tablespace quotas

PROFILE Clause

Specify the profile you want to assign to the user. The profile limits the amount of database resources the user can use. If you omit this clause, then Oracle Database assigns the `DEFAULT` profile to the user.

You can use the `CREATE USER` statement to create a new user, and associate the user with a profile that has the `PASSWORD_ROLLOVER_TIME` configured.

You must first set the password rollover period using `CREATE PROFILE` or `ALTER PROFILE`.

In the example `u1` is the user, with password `p1`. `prof1` is the profile with `PASSWORD_ROLLOVER_TIME` set.

```
CREATE USER u1 IDENTIFIED BY p1 PROFILE prof1 ;
```

Note:

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

See Also:

- GRANT and CREATE PROFILE
- Configuring Authentication

PASSWORD EXPIRE Clause

Specify `PASSWORD EXPIRE` if you want the user's password to expire. This setting forces the user or the DBA to change the password before the user can log in to the database.

ACCOUNT Clause

Specify `ACCOUNT LOCK` to lock the user's account and disable access. Specify `ACCOUNT UNLOCK` to unlock the user's account and enable access to the account. The default is `ACCOUNT UNLOCK`.

ENABLE EDITIONS

This clause is not reversible. Specify `ENABLE EDITIONS` to allow the user to create multiple versions of editionable objects in this schema using editions. Editionable objects in schemas that are not editions-enabled cannot be editioned.

Note the following before enabling editions with `ALTER USER`:

- Enabling editions is not a live operation.
- When a database is upgraded from Release 11.2 to Release 12.1, users who were enabled for editions in the pre-upgrade database are enabled for editions in the post-upgrade database and the default schema object types are editionable in their schemas. The default schema object types are displayed by the static data dictionary view `DBA_EDITIONED_TYPES` . Users who were not enabled for editions in the pre-upgrade database are not enabled for editions in the post-upgrade database and no schema object types are editionable in their schemas.
- To see which users already have editions enabled, see the `EDITIONS_ENABLED` column of the static data dictionary view `DBA_USERS` or `USER_USERS` .

Restriction on Enabling Editions

The `FOR` clause is ignored when used with `ENABLE EDITIONS`. This only applies to the `CREATE USER` statement, not the `ALTER USER` statement.

You cannot enable editions for any schemas supplied by Oracle.

See Also:

- *Enabling Editions for a User*
- *Oracle Database Reference* for more information about the `V$EDITIONABLE_TYPES` dynamic performance view

CONTAINER Clause

The `CONTAINER` clause applies when you are connected to a CDB. However, it is not necessary to specify the `CONTAINER` clause because its default values are the only allowed values.

- To create a common user, you must be connected to the root. You can optionally specify `CONTAINER = ALL`, which is the default when you are connected to the root.
- To create a local user, you must be connected to a PDB. You can optionally specify `CONTAINER = CURRENT`, which is the default when you are connected to a PDB.

While creating a common user, any default tablespace, temporary tablespace, or profile specified using the following clauses must exist in all the containers belonging to the CDB:

- `DEFAULT TABLESPACE`
- `TEMPORARY TABLESPACE`
- `QUOTA`
- `PROFILE`

If these objects do not exist in all the containers, the `CREATE USER` statement fails.

READ ONLY | READ WRITE

Use this clause to set `READ ONLY` access to a local PDB user.

With read-only access, the local PDB user is not permitted to execute any write operations on the PDB they connect to. The session operates as if the database is open in read-only mode.

Specify `READ WRITE` to set `READ WRITE` access to a local user.

You must have the `CREATE USER` privilege to execute this statement.

You can view the state of a local user in the `*_USERS` view.

Examples

All of the following examples use the `example` tablespace, which exists in the seed database and is accessible to the sample schemas.

Creating a Database User: Example

If you create a new user with `PASSWORD EXPIRE`, then the user's password must be changed before the user attempts to log in to the database. You can create the user `sidney` by issuing the following statement:

```
CREATE USER sidney
    IDENTIFIED BY out_standing1
    DEFAULT TABLESPACE example
    QUOTA 10M ON example
    TEMPORARY TABLESPACE temp
    QUOTA 5M ON system
    PROFILE app_user
    PASSWORD EXPIRE;
```

The user `sidney` has the following characteristics:

- The password `out_standing1`
- Default tablespace `example`, with a quota of 10 megabytes
- Temporary tablespace `temp`
- Access to the tablespace `SYSTEM`, with a quota of 5 megabytes
- Limits on database resources defined by the profile `app_user` (which was created in "Creating a Profile: Example")
- An expired password, which must be changed before `sidney` can log in to the database

Creating External Database Users: Examples

The following example creates an external user, who must be identified by an external source before accessing the database:

```
CREATE USER app_user1
   IDENTIFIED EXTERNALLY
   DEFAULT TABLESPACE example
   QUOTA 5M ON example
   PROFILE app_user;
```

The user `app_user1` has the following additional characteristics:

- Default tablespace `example`

- Default temporary tablespace `example`
- 5M of space on the tablespace `example` and unlimited quota on the temporary tablespace of the database
- Limits on database resources defined by the `app_user` profile

To create another user accessible only by an operating system account, prefix the user name with the value of the initialization parameter `OS_AUTHENT_PREFIX`. For example, if this value is "`ops$`", then you can create the externally identified user `external_user` with the following statement:

```
CREATE USER ops$external_user
   IDENTIFIED EXTERNALLY
   DEFAULT TABLESPACE example
   QUOTA 5M ON example
   PROFILE app_user;
```

Creating a Global Database User: Example

The following example creates a global user. When you create a global user, you can specify the X.509 name that identifies this user at the enterprise directory server:

```
CREATE USER global_user
   IDENTIFIED GLOBALLY AS 'CN=analyst, OU=division1, O=oracle, C=US'
   DEFAULT TABLESPACE example
   QUOTA 5M ON example;
```

Creating a Common User in a CDB

The following example creates a common user called c##`comm_user` in a CDB. Before you run this `CREATE USER` statement, ensure that the tablespaces `example` and `temp_tbs` exist in all of the containers in the CDB.

```
CREATE USER c##comm_user
   IDENTIFIED BY comm_pwd
   DEFAULT TABLESPACE example
   QUOTA 20M ON example
   TEMPORARY TABLESPACE temp_tbs;
```

The user `comm_user` has the following additional characteristics:

- The password `comm_pwd`
- Default tablespace `example`, with a quota of 20 megabytes
- Temporary tablespace `temp_tbs`

CREATE VECTOR INDEX

Purpose

Vector indexes speed up vector searches and are either exact search indexes or approximate search indexes. An exact search gives 100% accuracy at the cost of heavy compute resources. Approximate search indexes, also called vector indexes, trade accuracy for performance.

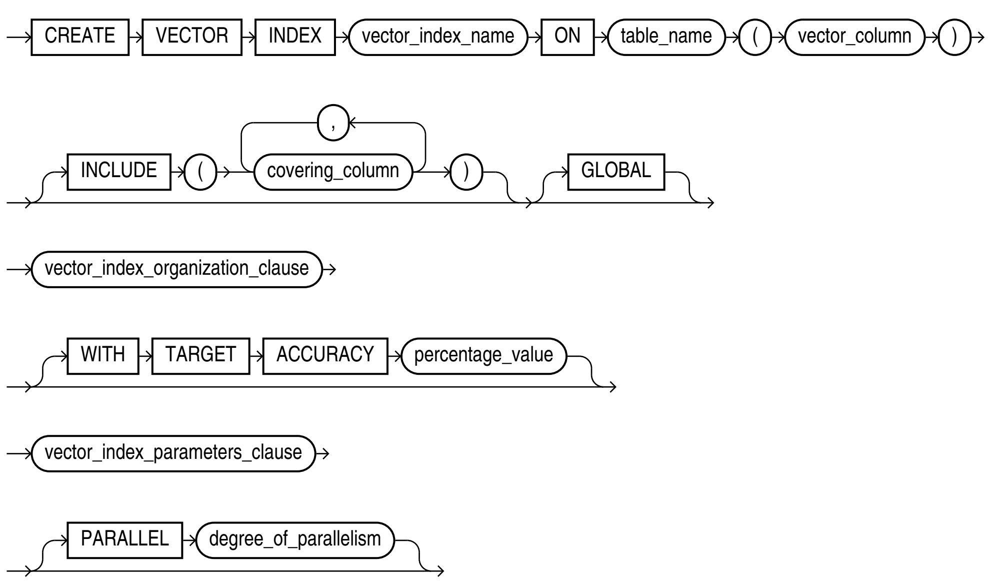
Vectors are grouped or connected together based on similarity, where similarity is determined by their relative distance to each other. Greedy searches are done across these groups and connections to find the best, closest match to the query vector being searched for. A search using a vector index is called an approximate search.

There are two vector indexes supported in vector search: IVF (Inverted File) Flat index and HNSW (Hierarchical Navigable Small Worlds) index. IVF Flat (also simply called IVF) is a partitioned-based index, while HNSW is a graph-based index. All partition based indexes are classifed as Neighbor Partition Vector Index, and all graph-based indexes are classifed as In-Memory Neighbor Graph Vector Index.

See Also:

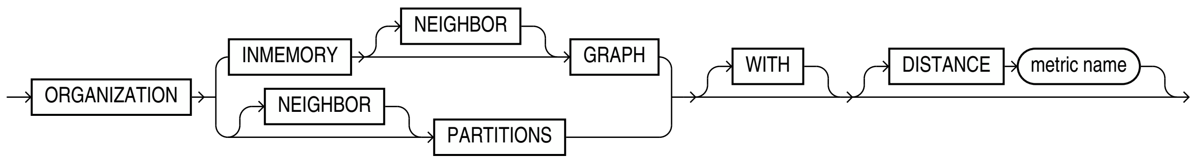
Create Vector Indexes of the *AI Vector Search User's Guide*

Syntax

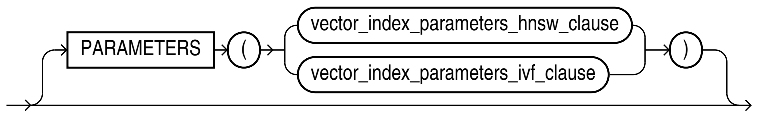


(*vector_index_organization_clause*::=,*vector_index_parameters_clause*::=,*vector_index_parameters_hnsw_clause*::=,*vector_index_parameters_ivf_clause*::=)

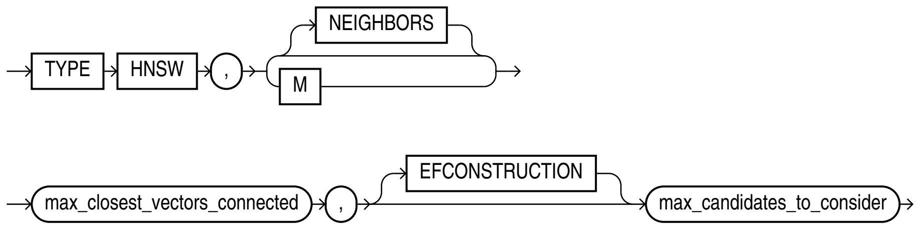
***vector_index_organization_clause*::=**



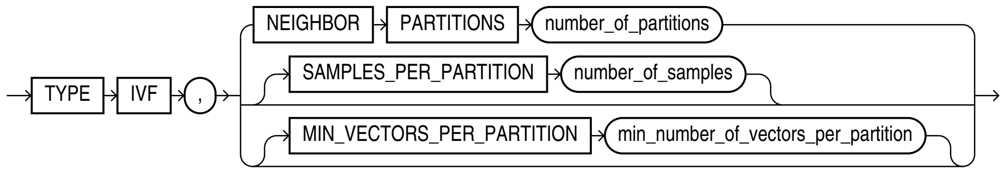
***vector_index_parameters_clause*::=**


PARAMETERS
(
vector_index_parameters_hnsw_clause
vector_index_parameters_ivf_clause
)

***vector_index_parameters_hnsw_clause*::=**



***vector_index_parameters_ivf_clause*::=**



Semantics

INCLUDE

See *Included Columns* of the *AI Vector Search User's Guide* for semantics.

vector_index_parameters_hnsw_clause

HNSW Specific Parameters

`NEIGHBORS` and `M` are equivalent and represent the maximum number of neighbors a vector can have on any layer. The last vertex has one additional flexibility that it can have up to 2M neighbors.

`EFCONSTRUCTION` represents the maximum number of closest vector candidates considered at each step of the search during insertion.

The valid range for HNSW vector index parameters are:

- `ACCURACY`: > 0 and <= 100
- `DISTANCE`: `EUCLIDEAN`, `L2_SQUARED` (aka `EUCLIDEAN_SQUARED`), `COSINE`, `DOT`, `MANHATTAN`, `HAMMING`
- `TYPE` : `HNSW`
- `NEIGHBORS`: >= 2 and <= 2048
- `EFCONSTRUCTION`: > 0 and <= 65535

vector_index_parameters_ivf_clause

IVF Parameters

`NEIGHBOR PARTITIONS` determines the number of centroid partitions that are created by the index.

`SAMPLE_PER_PARTITION` decides the total number of vectors that are passed to the clustering algorithm (number of samples per partition times the number of neighbor partitions). Note, that passing all the vectors would significantly increase the total time to create the index. Instead, aim to pass a subset of vectors that can capture the data distribution.

`MIN_VECTORS_PER_PARTITION` represents the target minimum number of vectors per partition. Aim to trim out any partition that can end up with fewer than 100 vectors. This may result in lesser number of centroids. Its values can range from 0 (no trimming of centroids) to num_vectors (would result in 1 neighbor partition).

The valid range for IVF vector index parameters are:

- `ACCURACY`: > 0 and <= 100
- `DISTANCE`: `EUCLIDEAN`, `L2_SQUARED` (aka `EUCLIDEAN_SQUARED`), `COSINE`, `DOT`, `MANHATTAN`, `HAMMING`
- `TYPE` : `IVF`
- `NEIGHBOR PARTITIONS`: >= 1 and <= 10000000
- `SAMPLE_PER_PARTITION`: from 1 to (num_vectors/neighbor_partitions)
- `MIN_VECTORS_PER_PARTITION`: from 0 (no trimming of centroid partitions) to total number of vectors (would result in 1 centroid partition)

Examples

```
CREATE VECTOR INDEX galaxies_hnsw_idx ON galaxies (embedding) ORGANIZATION INMEMORY
NEIGHBOR GRAPH
DISTANCE COSINE
WITH TARGET ACCURACY 95;
```

```
CREATE VECTOR INDEX galaxies_hnsw_idx ON galaxies (embedding) ORGANIZATION INMEMORY
NEIGHBOR GRAPH
DISTANCE COSINE
WITH TARGET ACCURACY 90 PARAMETERS (type HNSW, neighbors 40, efconstruction 500);
```

```
CREATE VECTOR INDEX galaxies_ivf_idx ON galaxies (embedding) ORGANIZATION NEIGHBOR
PARTITIONS
DISTANCE COSINE
WITH TARGET ACCURACY 95;
```

```
CREATE VECTOR INDEX galaxies_ivf_idx ON galaxies (embedding) ORGANIZATION NEIGHBOR
PARTITIONS
DISTANCE COSINE
WITH TARGET ACCURACY 90 PARAMETERS (type IVF, neighbor partitions 10);
```

CREATE VIEW

Purpose

Use the `CREATE VIEW` statement to define a **view**, which is a logical table based on one or more tables or views. A view contains no data itself. The tables upon which a view is based are called **base tables**.

You can create an **object view** or a relational view that supports LOBs, object types, `REF` data types, nested table, or varray types on top of the existing view mechanism. An object view is a view of a user-defined type, where each row contains objects, each object with a unique object identifier.

You can create a `XMLType` view, which is similar to an object view but displays data from XMLSchema-based tables of `XMLType`.

See Also:

- *Oracle Database Concepts*, *Oracle Database Development Guide*, and *Oracle Database Administrator's Guide* for information on various types of views and their uses
- *Oracle XML DB Developer's Guide* for information on `XMLType` views
- ALTER VIEW and DROP VIEW for information on modifying a view and removing a view from the database

Prerequisites

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

To create a subview, you must have the `UNDER ANY VIEW` system privilege or the `UNDER` object privilege on the superview.

The owner of the schema containing the view must have the privileges necessary to either select (`READ` or `SELECT` privilege), insert, update, or delete rows from all the tables or views on which the view is based. The owner must be granted these privileges directly, rather than through a role.

To use the basic constructor method of an object type when creating an object view, one of the following must be true:

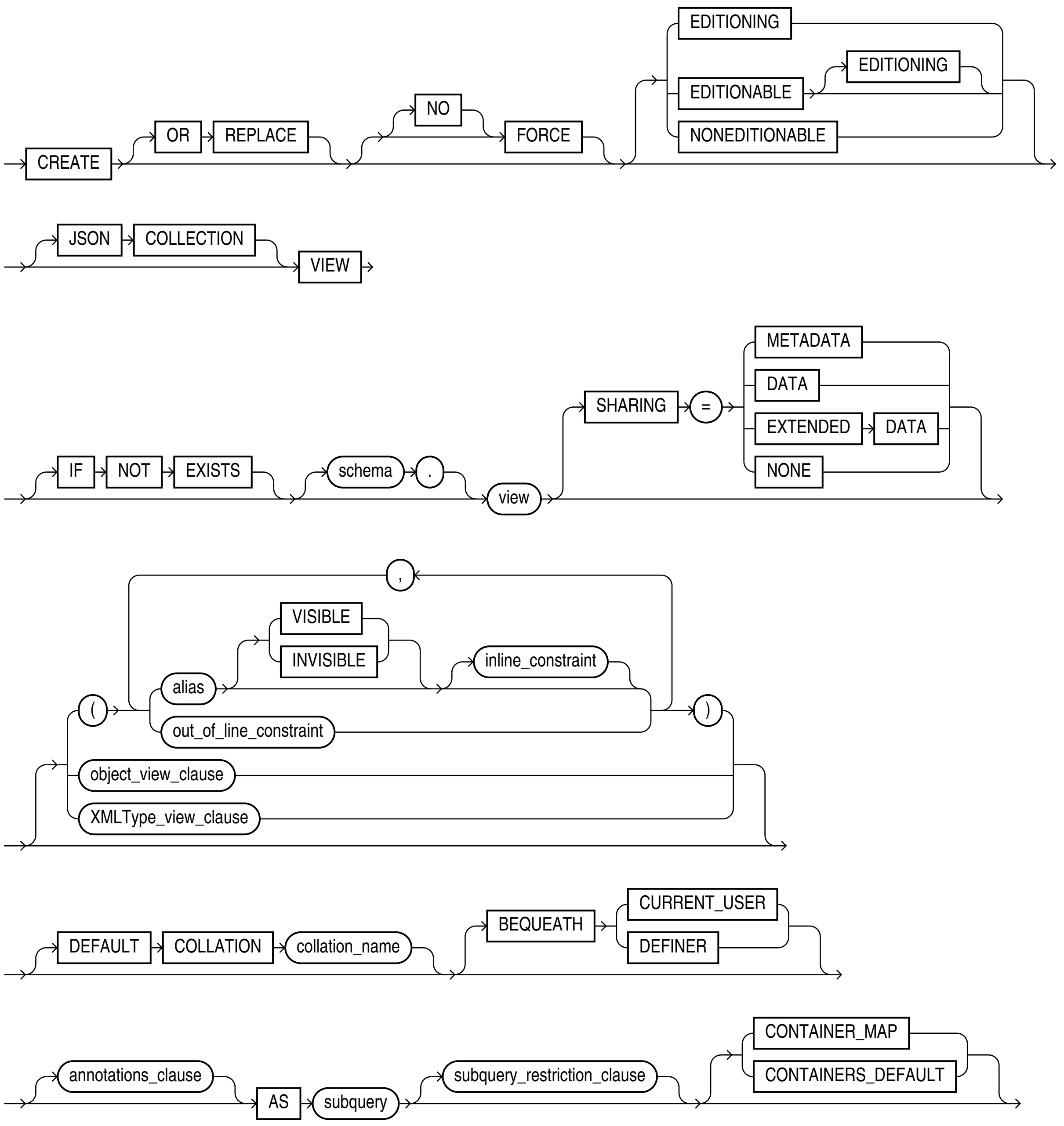
- The object type must belong to the same schema as the view to be created.
- You must have the `EXECUTE ANY TYPE` system privileges.
- You must have the `EXECUTE` object privilege on that object type.

See Also:

SELECT , INSERT , UPDATE , and DELETE for information on the privileges required by the owner of a view on the base tables or views of the view being created

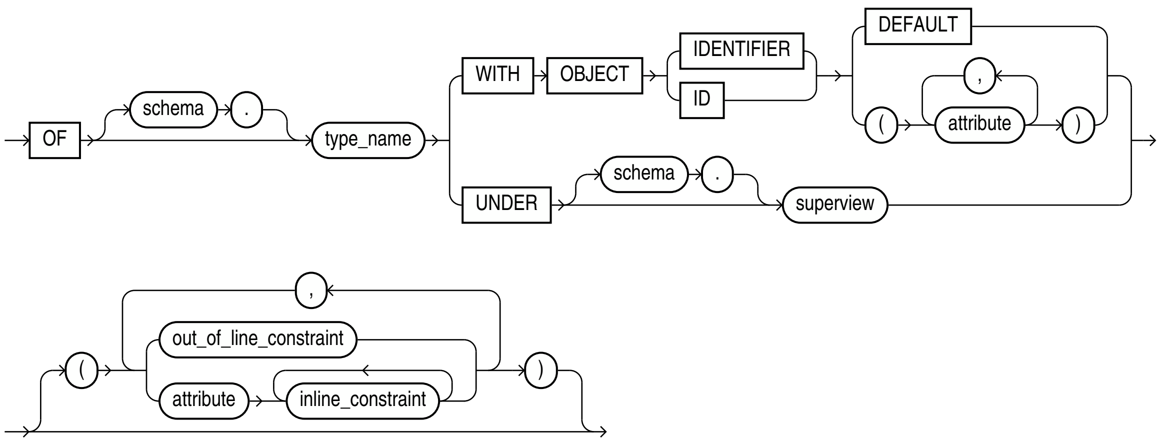
Syntax

***create_view*::=**



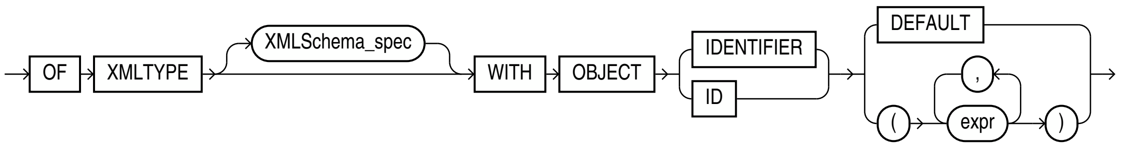
(*inline_constraint*::= and *out_of_line_constraint*::=, *object_view_clause*::=, *XMLType_view_clause*::=, *subquery*::=—part of `SELECT`, *subquery_restriction_clause*::=, *annotations_clause*)

***object_view_clause*::=**

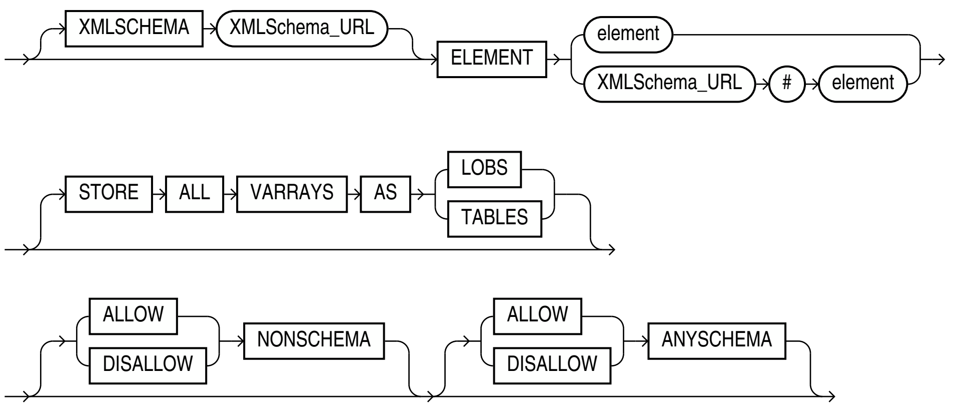


(*inline_constraint*::= and *out_of_line_constraint*::=)

***XMLType_view_clause*::=**



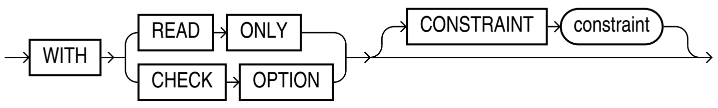
***XMLSchema_spec*::=**



***annotations_clause*::=**

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

***subquery_restriction_clause*::=**



Semantics

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.

INSTEAD OF triggers defined on a conventional view are dropped when the view is re-created. DML triggers defined on an editioning view are retained when an editioning view is re-created. However, such triggers can be rendered permanently invalid if the editioning view has changed so that it can no longer be compiled—for example if an editioning view column referenced in the trigger definition has been dropped.

If any materialized views are dependent on *view*, then those materialized views will be marked UNUSABLE and will require a full refresh to restore them to a usable state. Invalid materialized views cannot be used by query rewrite and cannot be refreshed until they are recompiled.

You cannot replace a conventional view with an editioning view or an editioning view with a conventional view. See *Oracle Database Development Guide* for more information on editioning views.

See Also:

- ALTER MATERIALIZED VIEW for information on refreshing invalid materialized views
- *Oracle Database Concepts* for information on materialized views in general
- CREATE TRIGGER for more information about the INSTEAD OF clause

FORCE

Specify FORCE if you want to create the view regardless of whether the base tables of the view or the referenced object types exist or the owner of the schema containing the view has privileges on them. These conditions must be true before any SELECT, INSERT, UPDATE, or DELETE statements can be issued against the view.

If the view definition contains any constraints, CREATE VIEW ... FORCE fails if the base table does not exist or the referenced object type does not exist. CREATE VIEW ... FORCE also fails if the view definition names a constraint that does not exist.

NO FORCE

Specify NOFORCE if you want to create the view only if the base tables exist and the owner of the schema containing the view has privileges on them. This is the default.

EDITIONING

Use this clause to create an **editioning view**. An editioning view is a single-table view that selects all rows from the base table and displays a subset of the base table columns. You can use an editioning view to isolate an application from DDL changes to the base table during administrative operations such as upgrades. You can obtain information about the relationship of existing editioning view to their base tables by querying the `USER_`, `ALL_`, and `DBA_EDITIONING_VIEW` data dictionary views.

The owner of an editioning view must be editions-enabled. Refer to ENABLE EDITIONS for more information.

Notes on Editioning Views

Editioning views differ from conventional views in several important ways:

- Editioning views are intended only to select and provide aliases for a subset of columns in a table. Therefore, the syntax for creating an editioning view is more limited than the syntax for creating a conventional view. Any violation of the restrictions that follow causes the creation of the view to fail, even if you specify `FORCE`.
- You can create DML triggers on editioning views. In this case, the database considers the editioning view to be the base object of the trigger. Such triggers fire when a DML operation target the editioning view itself. They do not fire if the DML operation targets the base table.
- You cannot create `INSTEAD OF` triggers on editioning views.

Restrictions on Editioning Views

Editioning views are subject to the following restrictions:

- Within any edition, you can create only one editioning view for any single table.
- You cannot specify the *`object_view_clause`*, *`XMLType_view_clause`*, or `BEQUEATH` clause.
- You cannot define a constraint `WITH CHECK OPTION` on an editioning view.
- In the select list of the defining subquery, you can specify only simple references to the columns of the base table, and you can specify each column of the base table only once in the select list. The asterisk wildcard symbol `*` and *`t_alias`*`.*` are supported to designate all columns of a base table.
- The `FROM` clause of the defining subquery of the view can reference only a single existing database table. Joins are not permitted. The base table must be in the same schema as the view being created. You cannot use a synonym to identify the table, but you can specify a table alias.
- The following clauses of the defining subquery are not valid for editioning views: *`subquery_factoring_clause`*, `DISTINCT` or `UNIQUE`, *`where_clause`*, *`hierarchical_query_clause`*, *`group_by_clause`*, `HAVING` condition, *`model_clause`*, or the set operators (`UNION`, `INTERSECT`, or `MINUS`)

See Also:

- *Oracle Database Development Guide* for detailed information about editioning views
- CREATE EDITION for information about editions, including an example of an editioning view

EDITIONABLE | NONEDITIONABLE

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

JSON Collection

You can create a `JSON COLLECTION` view that maps JSON documents to underlying relational data.

A `JSON COLLECTION` view is a special view that provides JSON documents in a single JSON-type object column named `DATA`. You cannot specify more than one column, otherwise you will raise an error.

`JSON COLLECTION` views are read only.

It is recommended, but not mandatory to define an `_id` column pointing to the unique identifier of the view to follow the standard format of JSON collections in Oracle.

Example

```
CREATE OR REPLACE JSON COLLECTION VIEW
AS
SELECT JSON { '_id': deptno, 'deptName': dname } DATA
FROM dept ;
```

For more information on JSON relational duality views, fully updateable views on top of relational tables, see CREATE JSON RELATIONAL DUALITY VIEW .

For more on JSON collections, see JSON Collections of the *JSON Developer's Guide*.

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

view

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

Restriction on Views

If a view has `INSTEAD OF` triggers, then any views created on it must have `INSTEAD OF` triggers, even if the views are inherently updatable.

See Also:

"Creating a View: Example"

SHARING

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

- `METADATA` - A metadata link shares the view's metadata, but its data is unique to each container. This type of view is referred to as a **metadata-linked application common object**.
- `DATA` - A data link shares the view, and its data is the same for all containers in the application container. Its data is stored only in the application root. This type of view is referred to as a **data-linked application common object**.
- `EXTENDED DATA` - An extended data link shares the view, and its data in the application root is the same for all containers in the application container. However, each application PDB in the application container can store data that is unique to the application PDB. For this type of view, data is stored in the application root and, optionally, in each application PDB. This type of view is referred to as an **extended data-linked application common object**.
- `NONE` - The view 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 view. If the `DEFAULT_SHARING` initialization parameter does not have a value, then the default is `METADATA`.

When creating a conventional view, you can specify `METADATA`, `DATA`, `EXTENDED DATA`, or `NONE`.

When creating an object view or an `XMLTYPE` view, you can specify only `METADATA` or `NONE`.

You cannot change the sharing attribute of a view 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

alias

Specify names for the expressions selected by the defining query of the view. The number of aliases must match the number of expressions selected by the view. Aliases must follow the rules for naming Oracle Database schema objects. Aliases must be unique within the view.

If you omit the aliases, then the database derives them from the columns or column aliases in the query. For this reason, you must use aliases if the query contains expressions rather than only column names. Also, you must specify aliases if the view definition includes constraints and/or column annotations.

Restriction on View Aliases

You cannot specify an alias when creating an object view.

See Also:

"Syntax for Schema Objects and Parts in SQL Statements"

VISIBLE | INVISIBLE

Use this clause to specify whether a view column is `VISIBLE` or `INVISIBLE`. By default, view columns are `VISIBLE` regardless of their visibility in the base tables, unless you specify `INVISIBLE`. This applies to conventional views and editioning views. For complete information on these clauses, refer to "VISIBLE | INVISIBLE" in the documentation on `CREATE TABLE`.

inline_constraint* and *out_of_line_constraint

You can specify constraints on views and object views. You define the constraint at the view level using the *out_of_line_constraint* clause. You define the constraint as part of column or attribute specification using the *inline_constraint* clause after the appropriate alias.

Oracle Database does not enforce view constraints. For a full discussion of view constraints, including restrictions, refer to "View Constraints ".

See Also:

"Creating a View with Constraints: Example"

object_view_clause

The *object_view_clause* lets you define a view on an object type.

See Also:

"Creating an Object View: Example"

OF *type_name* Clause

Use this clause to explicitly create an **object view** of type *type_name*. The columns of an object view correspond to the top-level attributes of type *type_name*. Each row will contain an object instance and each instance will be associated with an object identifier as specified in the `WITH OBJECT IDENTIFIER` clause. If you omit *schema*, then the database creates the object view in your own schema.

Object tables, as well as `XMLType` tables, object views, and `XMLType` views, do not have any column names specified for them. Therefore, Oracle Database defines a system-generated pseudocolumn `OBJECT_ID`. You can use this column name in queries and to create object views with the `WITH OBJECT IDENTIFIER` clause.

WITH OBJECT IDENTIFIER Clause

Use the `WITH OBJECT IDENTIFIER` clause to specify a top-level (root) object view. This clause lets you specify the attributes of the object type that will be used as a key to identify each row in the object view. In most cases these attributes correspond to the primary key columns of the base table. You must ensure that the attribute list is unique and identifies exactly one row in the view. The `WITH OBJECT IDENTIFIER` and `WITH OBJECT ID` clauses can be used interchangeably and are provided for semantic clarity.

Restrictions on Object Views

Object views are subject to the following restrictions:

- If you try to dereference or pin a primary key `REF` that resolves to more than one instance in the object view, then the database returns an error.
- You cannot specify this clause if you are creating a subview, because subviews inherit object identifiers from superviews.

If the object view is defined on an object table or an object view, then you can omit this clause or specify `DEFAULT`.

DEFAULT

Specify `DEFAULT` if you want the database to use the intrinsic object identifier of the underlying object table or object view to uniquely identify each row.

attribute

For *`attribute`*, specify an attribute of the object type from which the database should create the object identifier for the object view.

UNDER Clause

Use the `UNDER` clause to specify a subview based on an object superview.

Restrictions on Subviews

Subviews are subject to the following restrictions:

- You must create a subview in the same schema as the superview.
- The object type *`type_name`* must be the immediate subtype of *`superview`*.
- You can create only one subview of a particular type under the same superview.

See Also:

- CREATE TYPE for information about creating objects
- *Oracle Database Reference* for information on data dictionary views

XMLType_view_clause

Use this clause to create an `XMLType` view, which displays data from an XMLSchema-based table of type `XMLType`. The *`XMLSchema_spec`* indicates the XMLSchema to be used to map the XML data to its object-relational equivalents. The XMLSchema must already have been created before you can create an `XMLType` view.

The `WITH OBJECT IDENTIFIER` and `WITH OBJECT ID` clauses can be used interchangeably and are provided for semantic clarity.

Object tables, as well as `XMLType` tables, object views, and `XMLType` views, do not have any column names specified for them. Therefore, Oracle Database defines a system-generated pseudocolumn `OBJECT_ID`. You can use this column name in queries and to create object views with the `WITH OBJECT IDENTIFIER` clause.

See Also:

- *Oracle XML DB Developer's Guide* for information on `XMLType` views and XMLSchemas
- "Creating an XMLType View: Example" and "Creating a View on an XMLType Table: Example"

annotations_clause

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

DEFAULT COLLATION

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

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

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

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

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

Restriction on the Default Collation for Views

If the defining query of the view contains the `WITH` *`plsql_declarations`* clause, then the default collation of the view must be `USING_NLS_COMP`.

BEQUEATH

Use the `BEQUEATH` clause to specify whether functions referenced in the view are executed using the view invoker's rights or the view definer's rights.

CURRENT_USER

If you specify `BEQUEATH CURRENT_USER`, then functions referenced by the view are executed using the view invoker's rights as long as one of the following conditions is met:

- The view owner has the `INHERIT PRIVILEGES` object privilege on the invoking user.
- The view owner has the `INHERIT ANY PRIVILEGES` system privilege.

If a query of the view invokes an identity- or privilege-sensitive SQL function, or an invoker's rights PL/SQL or Java function, then the current schema, current user, and currently enabled roles within the operation's execution are inherited from the querying user's environment, rather than from the owner of the view.

This clause does not turn the view itself into an invoker's rights object. Name resolution within the view is still handled using the view owner's schema, and privilege checking for the view is done using the view owner's privileges.

DEFINER

If you specify `BEQUEATH DEFINER`, then functions referenced by the view are executed using the view definer's rights. If a query on the view invokes an identity- or privilege-sensitive SQL function, or an invoker's rights PL/SQL or Java function, then the current schema, current user, and currently enabled roles within the operation's execution are inherited from the owner of the view.

Name resolution within the view is handled using the view owner's schema, and privilege checking for the view is done using the view owner's privileges.

This is the default.

Restriction on the BEQUEATH Clause

You cannot specify this clause with the `EDITIONING` clause.

See Also:

Oracle Database Security Guide for more information on controlling invoker's rights and definer's rights in views

AS *subquery*

Specify a subquery that identifies columns and rows of the table(s) that the view is based on. The select list of the subquery can contain up to 1000 expressions.

If you create views that refer to remote tables and views, then the database links you specify must have been created using the `CONNECT TO` clause of the `CREATE DATABASE LINK` statement, and you must qualify them with a schema name in the view subquery.

If you create a view with the *`flashback_query_clause`* in the defining query, then the database does not interpret the `AS OF` expression at create time but rather each time a user subsequently queries the view.

See Also:

"Creating a Join View: Example" and *Oracle Database Development Guide* for more information on Oracle Flashback Query

Restrictions on the Defining Query of a View

The view query is subject to the following restrictions:

- The subquery cannot select the `CURRVAL` or `NEXTVAL` pseudocolumns.
- If the subquery selects the `ROWID`, `ROWNUM`, or `LEVEL` pseudocolumns, then those columns must have aliases in the view subquery.
- If the subquery uses an asterisk (*) to select all columns of a table, and you later add new columns to the table, then the view will not contain those columns until you re-create the view by issuing a `CREATE OR REPLACE VIEW` statement.
- For object views, the number of elements in the subquery select list must be the same as the number of top-level attributes for the object type. The data type of each of the selecting elements must be the same as the corresponding top-level attribute.
- You cannot specify the `SAMPLE` clause.

The preceding restrictions apply to materialized views as well.

Notes on Updatable Views

The following notes apply to updatable views:

An updatable view is one you can use to insert, update, or delete base table rows. You can create a view to be inherently updatable, or you can create an `INSTEAD OF` trigger on any view to make it updatable.

To learn whether and in what ways the columns of an inherently updatable view can be modified, query the `USER_UPDATABLE_COLUMNS` data dictionary view. The information displayed by this view is meaningful only for inherently updatable views. For a view to be inherently updatable, the following conditions must be met:

- Each column in the view must map to a column of a single table. For example, if a view column maps to the output of a `TABLE` clause (an unnested collection), then the view is not inherently updatable.
- The view must not contain any of the following constructs:

  A set operator
  A `DISTINCT` operator
  An aggregate or analytic function
  A `GROUP BY`, `ORDER BY`, `MODEL`, `CONNECT BY`, or `START WITH` clause
  A collection expression in a `SELECT` list
  A subquery in a `SELECT` list
  A subquery designated `WITH READ ONLY`
  Joins, with some exceptions, as documented in *Oracle Database Administrator's Guide*
- In addition, if an inherently updatable view contains pseudocolumns or expressions, then you cannot update base table rows with an `UPDATE` statement that refers to any of these pseudocolumns or expressions.
- If you want a join view to be updatable, then all of the following conditions must be true:

- The DML statement must affect only one table underlying the join.
- For an `INSERT` statement, the view must not be created `WITH CHECK OPTION`, and all columns into which values are inserted must come from a **key-preserved table**. A key-preserved table is one for which every primary key or unique key value in the base table is also unique in the join view.
- For an `UPDATE` statement, the view must not be created `WITH CHECK OPTION`, and update must be deterministic (updates each row only once).
- For a `DELETE` statement, if the join results in more than one key-preserved table, then Oracle Database deletes from the first table named in the `FROM` clause, whether or not the view was created `WITH CHECK OPTION`.

See Also:

- *Oracle Database Administrator's Guide* for more information on updatable views
- "Creating an Updatable View: Example", "Creating a Join View: Example" for an example of updatable join views and key-preserved tables, and *Oracle Database PL/SQL Language Reference* for an example of an `INSTEAD OF` trigger on a view that is not inherently updatable

subquery_restriction_clause

Use the *`subquery_restriction_clause`* to restrict the defining query of the view in one of the following ways:

WITH READ ONLY

Specify `WITH READ ONLY` to indicate that the table or view cannot be updated.

WITH CHECK OPTION

Specify `WITH CHECK OPTION` to indicate that Oracle Database prohibits any changes to the table or view that would produce rows that are not included in the subquery. When used in the subquery of a DML statement, you can specify this clause in a subquery in the `FROM` clause but not in subquery in the `WHERE` clause.

CONSTRAINT *constraint*

Specify the name of the `READ ONLY` or `CHECK OPTION` constraint. If you omit this identifier, then Oracle automatically assigns the constraint a name of the form `SYS_C`*`n`*, where *n* is an integer that makes the constraint name unique within the database.

Note:

For tables, `WITH CHECK OPTION` guarantees that inserts and updates result in tables that the defining table subquery can select. For views, `WITH CHECK OPTION` cannot make this guarantee if:

- There is a subquery within the defining query of this view or any view on which this view is based or
- `INSERT`, `UPDATE`, or `DELETE` operations are performed using `INSTEAD OF` triggers.

Restriction on the *subquery_restriction_clause*

You cannot specify this clause if you specify an `ORDER BY` clause.

See Also:

"Creating a Read-Only View: Example"

CONTAINER_MAP

Specify the `CONTAINER_MAP` clause to enable the view to be queried using a container map.

CONTAINERS_DEFAULT

Specify the `CONTAINERS_DEFAULT` clause to enable the view for the `CONTAINERS` clause.

Examples

Creating a View: Example

The following statement creates a view of the sample table `employees` named `emp_view`. The view shows the employees in department 20 and their annual salary:

```
CREATE VIEW emp_view AS
   SELECT last_name, salary*12 annual_salary
   FROM employees
   WHERE department_id = 20;
```

The view declaration need not define a name for the column based on the expression `salary*12`, because the subquery uses a column alias (`annual_salary`) for this expression.

Creating an Editioning View: Example

The following statement creates an editioning view of the `orders` table:

```
CREATE EDITIONING VIEW ed_orders_view (o_id, o_date, o_status)
  AS SELECT order_id, order_date, order_status FROM orders
  WITH READ ONLY;
```

You can use this view to isolate an application from DDL changes to the `orders` table during an administrative operation such as an upgrade. You can create a DML trigger on this view, so that the trigger fires when a DML operation targets the view itself, but does not fire if the DML operation targets the `orders` table.

Creating a View with Constraints: Example

The following statement creates a restricted view of the sample table `hr.employees` and defines a unique constraint on the `email` view column and a primary key constraint for the view on the `emp_id` view column:

```
CREATE VIEW emp_sal (emp_id, last_name,
      email UNIQUE RELY DISABLE NOVALIDATE,
   CONSTRAINT id_pk PRIMARY KEY (emp_id) RELY DISABLE NOVALIDATE)
   AS SELECT employee_id, last_name, email FROM employees;
```

Creating an Updatable View: Example

The following statement creates an updatable view named `clerk` of all clerks in the `employees` table. Only the employees' IDs, last names, department numbers, and jobs are visible in this view, and these columns can be updated only in rows where the employee is a kind of clerk:

```
CREATE VIEW clerk AS
   SELECT employee_id, last_name, department_id, job_id
   FROM employees
   WHERE job_id = 'PU_CLERK'
      or job_id = 'SH_CLERK'
      or job_id = 'ST_CLERK';
```

This view lets you change the `job_id` of a purchasing clerk to purchasing manager (`PU_MAN`):

```
UPDATE clerk SET job_id = 'PU_MAN' WHERE employee_id = 118;
```

The next example creates the same view `WITH CHECK OPTION`. You cannot subsequently insert a new row into `clerk` if the new employee is not a clerk. You can update an employee's `job_id` from one type of clerk to another type of clerk, but the update in the preceding statement would fail, because the view cannot access employees with non-clerk `job_id`.

```
CREATE VIEW clerk AS
   SELECT employee_id, last_name, department_id, job_id
   FROM employees
   WHERE job_id = 'PU_CLERK'
      or job_id = 'SH_CLERK'
      or job_id = 'ST_CLERK'
   WITH CHECK OPTION;
```

Creating a Join View: Example

A join view is one whose view subquery contains a join. If at least one column in the join has a unique index, then it may be possible to modify one base table in a join view. You can query `USER_UPDATABLE_COLUMNS` to see whether the columns in a join view are updatable. For example:

```
CREATE VIEW locations_view AS
   SELECT d.department_id, d.department_name, l.location_id, l.city
   FROM departments d, locations l
   WHERE d.location_id = l.location_id;

SELECT column_name, updatable
   FROM user_updatable_columns
   WHERE table_name = 'LOCATIONS_VIEW'
   ORDER BY column_name, updatable;

COLUMN_NAME                    UPD
------------------------------ ---
DEPARTMENT_ID                  YES
DEPARTMENT_NAME                YES
LOCATION_ID                    NO
CITY                           NO
```

In the preceding example, the primary key index on the `location_id` column of the `locations` table is not unique in the `locations_view` view. Therefore, `locations` is not a key-preserved table and columns from that base table are not updatable.

```
INSERT INTO locations_view VALUES
   (999, 'Entertainment', 87, 'Roma');
INSERT INTO locations_view VALUES
*
ERROR at line 1:
ORA-01776: cannot modify more than one base table through a join view
```

You can insert, update, or delete a row from the departments base table, because all the columns in the view mapping to the departments table are marked as updatable and because the primary key of departments is retained in the view.

```
INSERT INTO locations_view (department_id, department_name)
   VALUES (999, 'Entertainment');

1 row created.
```

Note:

For you to insert into the table using the view, the view must contain all NOT NULL columns of all tables in the join, unless you have specified DEFAULT values for the NOT NULL columns.

See Also:

Oracle Database Administrator's Guide for more information on updating join views

Creating a Read-Only View: Example

The following statement creates a read-only view named customer_ro of the oe.customers table. Only the customers' last names, language, and credit limit are visible in this view:

```
CREATE VIEW customer_ro (name, language, credit)
      AS SELECT cust_last_name, nls_language, credit_limit
      FROM customers
      WITH READ ONLY;
```

Creating an Object View: Example

The following example shows the creation of the type inventory_typ in the oc schema, and the oc_inventories view that is based on that type:

```
CREATE TYPE inventory_typ
 OID '82A4AF6A4CD4656DE034080020E0EE3D'
 AS OBJECT
    ( product_id          NUMBER(6)
    , warehouse           warehouse_typ
    , quantity_on_hand    NUMBER(8)
    ) ;
/
CREATE OR REPLACE VIEW oc_inventories OF inventory_typ
 WITH OBJECT OID (product_id)
 AS SELECT i.product_id,
           warehouse_typ(w.warehouse_id, w.warehouse_name, w.location_id),
           i.quantity_on_hand
    FROM inventories i, warehouses w
    WHERE i.warehouse_id=w.warehouse_id;
```

Creating a View on an XMLType Table: Example

The following example builds a regular view on the XMLType table xwarehouses, which was created in "Examples ":

```
CREATE VIEW warehouse_view AS
   SELECT VALUE(p) AS warehouse_xml
   FROM xwarehouses p;
```

You select from such a view as follows:

```
SELECT e.warehouse_xml.getclobval()
   FROM warehouse_view e
   WHERE EXISTSNODE(warehouse_xml, '//Docks') =1;
```

Creating an XMLType View: Example

In some cases you may have an object-relational table upon which you would like to build an `XMLType` view. The following example creates an object-relational table resembling the `XMLType` column `warehouse_spec` in the sample table `oe.warehouses`, and then creates an `XMLType` view of that table:

```
CREATE TABLE warehouse_table
(
  WarehouseID       NUMBER,
  Area              NUMBER,
  Docks             NUMBER,
  DockType          VARCHAR2(100),
  WaterAccess       VARCHAR2(10),
  RailAccess        VARCHAR2(10),
  Parking           VARCHAR2(20),
  VClearance        NUMBER
);

INSERT INTO warehouse_table
   VALUES(5, 103000,3,'Side Load','false','true','Lot',15);

CREATE VIEW warehouse_view OF XMLTYPE
 XMLSCHEMA "http://www.example.com/xwarehouses.xsd"
    ELEMENT "Warehouse"
    WITH OBJECT ID
    (extract(OBJECT_VALUE, '/Warehouse/Area/text()').getnumberval())
 AS SELECT XMLELEMENT("Warehouse",
            XMLFOREST(WarehouseID as "Building",
                      area as "Area",
                      docks as "Docks",
                      docktype as "DockType",
                      wateraccess as "WaterAccess",
                      railaccess as "RailAccess",
                      parking as "Parking",
                      VClearance as "VClearance"))
  FROM warehouse_table;
```

You query this view as follows:

```
SELECT VALUE(e) FROM warehouse_view e;
```

DELETE

Purpose

Use the `DELETE` statement to remove rows from:

- An unpartitioned or partitioned table
- The unpartitioned or partitioned base table of a view

- The unpartitioned or partitioned container table of a writable materialized view
- The unpartitioned or partitioned master table of an updatable materialized view

Prerequisites

For you to delete rows from a table, the table must be in your own schema or you must have the `DELETE` object privilege on the table.

For you to delete rows from an updatable materialized view, the materialized view must be in your own schema or you must have the `DELETE` object privilege on the materialized view.

For you to delete rows from the base table of a view, the owner of the schema containing the view must have the `DELETE` object privilege on the base table. Also, if the view is in a schema other than your own, then you must have the `DELETE` object privilege on the view.

The `DELETE ANY TABLE` system privilege also allows you to delete rows from any table or table partition or from the base table of any view.

To delete rows from an object on a remote database, you must also have the `READ` or `SELECT` object privilege on the object.

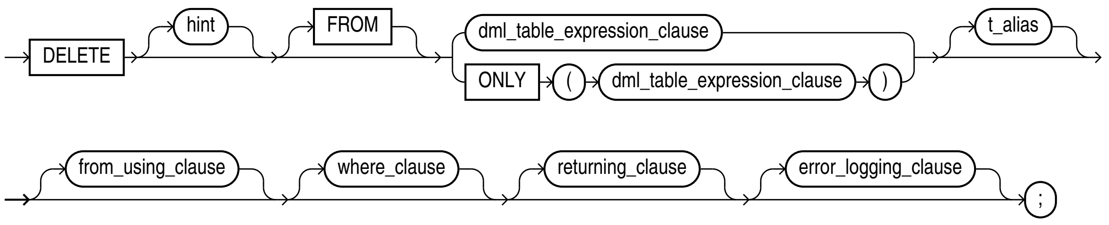
To specify the *`returning_clause`*, you must have the `READ` or `SELECT` object privilege on the object.

If the `SQL92_SECURITY` initialization parameter is set to `TRUE` and the `DELETE` operation references table columns, such as the columns in a *`where_clause`* or *`returning_clause`* , then you must have the `SELECT` object privilege on the object from which you want to delete rows.

You cannot delete rows from a table if a function-based index on the table has become invalid. You must first validate the function-based index.

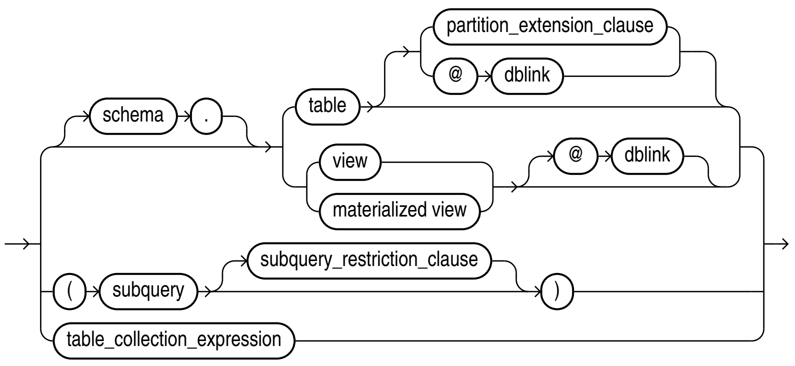
Syntax

delete::=



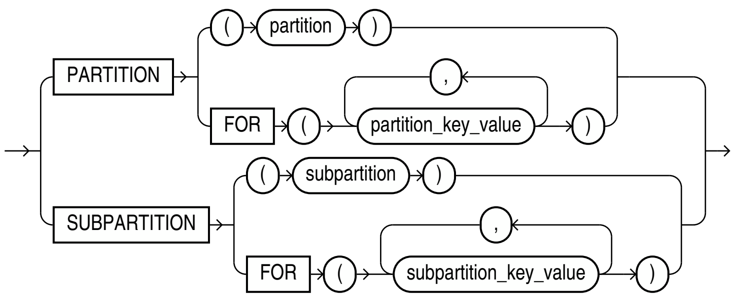
(*DML_table_expression_clause*::=, *where_clause*::=, *returning_clause*::=, *error_logging_clause*::=, *from_using_clause*::=)

***DML_table_expression_clause*::=**

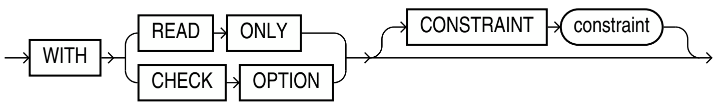

partition_extension_clause
@
dblink
schema
.
table
view
materialized view
@
dblink
subquery_restriction_clause
(
subquery
)
table_collection_expression

(*partition_extension_clause*::=, *subquery*::=, *subquery_restriction_clause*::=, *table_collection_expression*::=)

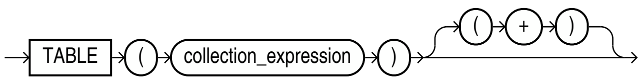
***partition_extension_clause*::=**



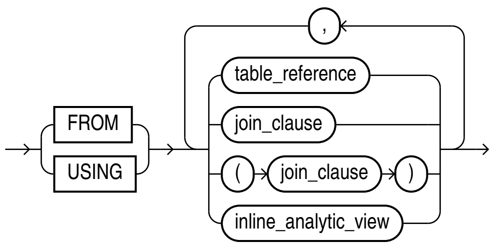
***subquery_restriction_clause*::=**



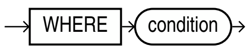
***table_collection_expression*::=**



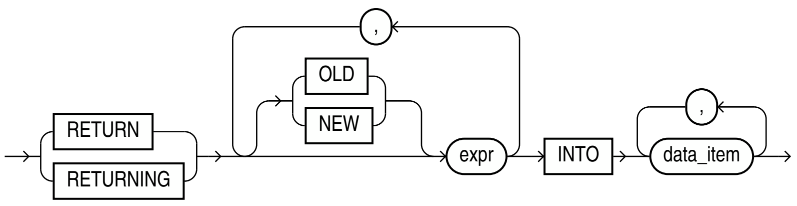
***from_using_clause*::=**



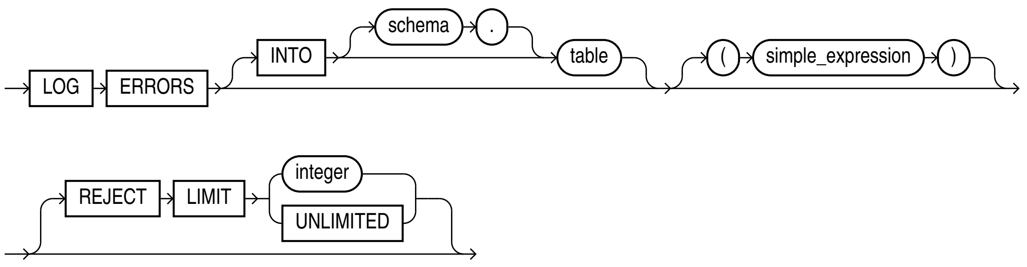
***where_clause*::=**



***returning_clause*::=**



***error_logging_clause*::=**



Semantics

hint

Specify a comment that passes instructions to the optimizer on choosing an execution plan for the statement.

See Also:

"Hints " for the syntax and description of hints

from_clause

Use the `FROM` clause to specify the database objects from which you are deleting rows.

The `ONLY` syntax is relevant only for views. Use the `ONLY` clause if the view in the `FROM` clause belongs to a view hierarchy and you do not want to delete rows from any of its subviews.

DML_table_expression_clause

Use this clause to specify the objects from which data is being deleted.

schema

Specify the schema containing the table or view. If you omit *schema*, then Oracle Database assumes the table or view is in your own schema.

table | view | materialized view | subquery

Specify the name of a table, view, materialized view, or the column or columns resulting from a subquery, from which the rows are to be deleted.

When you delete rows from an updatable view, Oracle Database deletes rows from the base table.

You cannot delete rows from a read-only materialized view. If you delete rows from a writable materialized view, then the database removes the rows from the underlying container table. However, the deletions are overwritten at the next refresh operation. If you delete rows from an updatable materialized view that is part of a materialized view group, then the database also removes the corresponding rows from the master table.

If *table* or the base table of *view* or the master table of *materialized_view* contains one or more domain index columns, then this statement executes the appropriate indextype delete routine.

See Also:

Oracle Database Data Cartridge Developer's Guide for more information on these routines

Issuing a `DELETE` statement against a table fires any `DELETE` triggers defined on the table.

All table or index space released by the deleted rows is retained by the table and index.

partition_extension_clause

Specify the name or partition key value of the partition or subpartition targeted for deletes within the object.

You need not specify the partition name when deleting values from a partitioned object. However, in some cases, specifying the partition name is more efficient than a complicated *where_clause*.

See Also:

"References to Partitioned Tables and Indexes " and "Deleting Rows from a Partition: Example"

dblink

Specify the complete or partial name of a database link to a remote database where the object is located. You can delete rows from a remote object only if you are using Oracle Database distributed functionality.

Note:

Starting with Oracle Database 12*c* Release 2 (12.2), the `DELETE` statement accepts remote LOB locators as bind variables. Refer to the “Distributed LOBs” chapter in *Oracle Database SecureFiles and Large Objects Developer's Guide* for more information.

See Also:

"References to Objects in Remote Databases " for information on referring to database links and "Deleting Rows from a Remote Database: Example"

If you omit *`dblink`*, then the database assumes that the object is located on the local database.

subquery_restriction_clause

The *`subquery_restriction_clause`* lets you restrict the subquery in one of the following ways:

WITH READ ONLY

Specify `WITH READ ONLY` to indicate that the table or view cannot be updated.

WITH CHECK OPTION

Specify `WITH CHECK OPTION` to indicate that Oracle Database prohibits any changes to the table or view that would produce rows that are not included in the subquery. When used in the subquery of a DML statement, you can specify this clause in a subquery in the `FROM` clause but not in subquery in the `WHERE` clause.

CONSTRAINT *constraint*

Specify the name of the `CHECK OPTION` constraint. If you omit this identifier, then Oracle automatically assigns the constraint a name of the form `SYS_C`*`n`*, where n is an integer that makes the constraint name unique within the database.

See Also:

"Using the WITH CHECK OPTION Clause: Example"

table_collection_expression

The *`table_collection_expression`* lets you inform Oracle that the value of *`collection_expression`* should be treated as a table for purposes of query and DML operations. The *`collection_expression`* can be a subquery, a column, a function, or a collection constructor. Regardless of its form, it must return a collection value—that is, a value whose type is nested table or varray. This process of extracting the elements of a collection is called **collection unnesting**.

The optional plus (+) is relevant if you are joining the `TABLE` collection expression with the parent table. The + creates an outer join of the two, so that the query returns rows from the outer table even if the collection expression is null.

Note:

In earlier releases of Oracle, when *collection_expression* was a subquery, *table_collection_expression* was expressed as THE *subquery*. That usage is now deprecated.

You can use a *table_collection_expression* in a correlated subquery to delete rows with values that also exist in another table.

See Also:

"Table Collections: Examples"

collection_expression

Specify a subquery that selects a nested table column from the object from which you are deleting.

Restrictions on the *dml_table_expression_clause* Clause

This clause is subject to the following restrictions:

- You cannot execute this statement if *table* or the base or master table of *view* or *materialized_view* contains any domain indexes marked IN_PROGRESS or FAILED.
- You cannot insert into a partition if any affected index partitions are marked UNUSABLE.
- You cannot specify the ORDER BY clause in the subquery of the *DML_table_expression_clause*.
- You cannot delete from a view except through INSTEAD OF triggers if the defining query of the view contains one of the following constructs:

  A set operator
  A DISTINCT operator
  An aggregate or analytic function
  A GROUP BY, ORDER BY, MODEL, CONNECT BY, or START WITH clause
  A collection expression in a SELECT list
  A subquery in a SELECT list
  A subquery designated WITH READ ONLY
  Joins, with some exceptions, as documented in *Oracle Database Administrator's Guide*

If you specify an index, index partition, or index subpartition that has been marked UNUSABLE, then the DELETE statement will fail unless the SKIP_UNUSABLE_INDEXES initialization parameter has been set to true.

See Also:

ALTER SESSION

t_alias

Provide a **correlation name** for the table, view, materialized view, subquery, or collection value to be referenced elsewhere in the statement. This alias is required if the *DML_table_expression_clause* references any object type attributes or object type methods. Table aliases are generally used in DELETE statements with correlated queries.

from_using_clause

Use this clause to filter the rows DELETE removes. Specify the join conditions in the *where_clause*. You can outer join source tables to the target with (+). The target table cannot be the outer table in the join.

You can join many tables, views, and inline views. Specify the join conditions in the *where_clause* or use the *join_clause* to join these to each other with ANSI join syntax.

You can specify the same table in the *dml_table_expression_clause* and *from_using_clause*. When you do so they must have unique aliases.

Example: Delete with Direct-Join

In this example, the join condition between table tand table s determines which rows of t are deleted:

```
DELETE FROM t
FROM s
WHERE t.t1 = s.s1;
```

If the join condition results in the same target row being selected more than once, the DELETE will succeed, and the deletion count will correctly reflect the number of rows deleted.

In a DELETE of a join view, one of the tables must be key-preserving. That table is used as the delete target. If there is more than one table that is key-preserving, the first key-preserved table encountered in the FROM clause is used as the delete target. If no such table exists, error ORA-01752 is raised. There is no such restriction in direct join syntax, since it is clear what the delete target is.

Direct joins for DELETE have the same semantics and restrictions as SELECT in the *from_clause* and *where_clause*. Triggers on the target table fire as normal.

Restrictions

- You cannot specify ANSI join syntax using the *dml_table_expression_clause*. However, you can specify ANSI join syntax between the tables specified in the FROM clause. Right and full outer joins are not allowed.
- You can use a lateral view in the FROM clause, but it cannot reference a column from the delete target. It may be outer-joined.
- You can only specify one table, view, or materialized view in *dml_table_expression_clause* when the *from_using_clause* is present.
- The *hint* clause can be used to specify instructions to the optimizer for joins involving the *from_using_clause*

where_clause

Use the *where_clause* to delete only rows that satisfy the condition. The condition can reference the object from which you are deleting and can contain a subquery. You can delete rows from a remote object only if you are using Oracle Database distributed functionality. Refer to Conditions for the syntax of `condition`.

If this clause contains a `subquery` that refers to remote objects, then the `DELETE` operation can run in parallel as long as the reference does not loop back to an object on the local database. However, if the `subquery` in the `DML_table_expression_clause` refers to any remote objects, then the `DELETE` operation will run serially without notification. Refer to the *parallel_clause* in the `CREATE TABLE` documentation for additional information.

If you omit `dblink`, then the database assumes that the table or view is located on the local database.

If you omit the `where_clause`, then the database deletes all rows of the object.

returning_clause

This clause lets you return values from deleted columns, and thereby eliminate the need to issue a `SELECT` statement following the `DELETE` statement.

The returning clause retrieves the rows affected by a DML statement. You can specify this clause for tables and materialized views and for views with a single base table.

When operating on a single row, a DML statement with a `returning_clause` can retrieve column expressions using the affected row, rowid, and `REF`s to the affected row and store them in host variables or PL/SQL variables.

When operating on multiple rows, a DML statement with the `returning_clause` stores values from expressions, rowids, and `REF`s involving the affected rows in bind arrays.

expr

Each item in the `expr` list must be a valid expression syntax.

INTO

The `INTO` clause indicates that the values of the changed rows are to be stored in the variable(s) specified in `data_item` list.

data_item

Each `data_item` is a host variable or PL/SQL variable that stores the retrieved `expr` value.

For each expression in the `RETURNING` list, you must specify a corresponding type-compatible PL/SQL variable or host variable in the `INTO` list.

Restrictions on the RETURNING Clause

The following restrictions apply to the `RETURNING` clause:

- The `expr` is restricted as follows:
  - For `UPDATE` and `DELETE` statements each `expr` must be a simple expression or a single-set aggregate function expression. You cannot combine simple expressions and single-set aggregate function expressions in the same `returning_clause`. For `INSERT` statements, each `expr` must be a simple expression. Aggregate functions are not supported in an `INSERT` statement `RETURNING` clause.
  - Single-set aggregate function expressions cannot include the `DISTINCT` keyword.
- If the `expr` list contains a primary key column or other `NOT NULL` column, then the update statement fails if the table has a `BEFORE UPDATE` trigger defined on it.

- You cannot specify the *returning_clause* for a multitable insert.
- You cannot use this clause with parallel DML or with remote objects.
- You cannot retrieve LONG types with this clause.
- You cannot specify this clause for a view on which an INSTEAD OF trigger has been defined.

See Also:

- *Oracle Database PL/SQL Language Reference* for information on using the BULK COLLECT clause to return multiple values to collection variables
- "Using the RETURNING Clause: Example"

error_logging_clause

The *error_logging_clause* has the same behavior in DELETE statement as it does in an INSERT statement. Refer to the INSERT statement *error_logging_clause* for more information.

See Also:

"Inserting Into a Table with Error Logging: Example"

Examples

Deleting Rows: Examples

The following statement deletes all rows from the sample table oe.product_descriptions where the value of the language_id column is AR:

```
DELETE FROM product_descriptions
   WHERE language_id = 'AR';
```

The following statement deletes from the sample table hr.employees purchasing clerks whose commission rate is less than 10%:

```
DELETE FROM employees
   WHERE job_id = 'SA_REP'
   AND commission_pct < .2;
```

The following statement has the same effect as the preceding example, but uses a subquery:

```
DELETE FROM (SELECT * FROM employees)
   WHERE job_id = 'SA_REP'
   AND commission_pct < .2;
```

Deleting Rows from a Remote Database: Example

The following statement deletes specified rows from the locations table owned by the user hr on a database accessible by the database link remote:

```
DELETE FROM hr.locations@remote
   WHERE location_id > 3000;
```

Deleting Nested Table Rows: Example

For an example that deletes nested table rows, refer to "Table Collections: Examples".

Deleting Rows from a Partition: Example

The following example removes rows from partition `sales_q1_1998` of the `sh.sales` table:

```
DELETE FROM sales PARTITION (sales_q1_1998)
   WHERE amount_sold > 1000;
```

Using the RETURNING Clause: Example

The following example returns column `salary` from the deleted rows and stores the result in bind variable `:bnd1`. The bind variable must already have been declared.

```
DELETE FROM employees
   WHERE job_id = 'SA_REP'
   AND hire_date + TO_YMINTERVAL('01-00') < SYSDATE
   RETURNING salary INTO :bnd1;
```

Deleting Data from a Table: Example

The following statements create a table named product_price_history and insert data into it:

```
CREATE TABLE product_price_history (
  product_id          INTEGER NOT NULL,
  price               INTEGER NOT NULL,
  currency_code       VARCHAR2(3 CHAR) NOT NULL,
  effective_from_date DATE NOT NULL,
  effective_to_date   DATE,
  CONSTRAINT product_price_history_pk
    PRIMARY KEY (product_id, currency_code, effective_from_date)
) PARTITION BY RANGE (effective_from_date) (
    PARTITION p0 VALUES less than (DATE'2015-01-02'),
    PARTITION p1 VALUES less than (DATE'2015-01-03'),
    PARTITION p2 VALUES less than (DATE'2015-01-04')
);

INSERT INTO product_price_history
  WITH prices AS (
    SELECT 1, 100, 'USD', DATE'2015-01-01', DATE'2015-01-02'
    FROM   dual UNION ALL
    SELECT 1, 60, 'GBP', DATE'2015-01-01', DATE'2015-01-02'
    FROM   dual UNION ALL
    SELECT 1, 110, 'EUR', DATE'2015-01-01', DATE'2015-01-02'
    FROM   dual UNION ALL
    SELECT 1, 101, 'USD', DATE'2015-01-02', DATE'2015-01-03'
    FROM   dual UNION ALL
    SELECT 1, 62, 'GBP', DATE'2015-01-02', DATE'2015-01-03'
    FROM   dual UNION ALL
    SELECT 1, 109, 'EUR', DATE'2015-01-02', DATE'2015-01-03'
    FROM   dual UNION ALL
    SELECT 1, 105, 'USD', DATE'2015-01-03', NULL
    FROM   dual UNION ALL
    SELECT 1, 61, 'GBP', DATE'2015-01-03', NULL
    FROM   dual UNION ALL
    SELECT 1, 107, 'EUR', DATE'2015-01-03', NULL
    FROM   dual UNION ALL
    SELECT 2, 30, 'USD', DATE'2015-01-01', DATE'2015-01-03'
    FROM   dual UNION ALL
    SELECT 2, 33, 'USD', DATE'2015-01-03', NULL
    FROM   dual UNION ALL
    SELECT 3, 100, 'GBP', DATE'2015-01-03', NULL
    FROM   dual
  )
SELECT *
FROM   prices;
```

The following statement deletes the rows from the table product_price_history where product_id is 3:

```
DELETE FROM product_price_history WHERE product_id = 3;
```

The following procedure deletes the rows from the product_price_history where product_id is 2 and effective_to_date is NULL:

```
DECLARE
  currency product_price_history.currency_code%TYPE;
BEGIN
  DELETE product_price_history
  WHERE  product_id = 2
  AND    effective_to_date IS NULL
  returning currency_code INTO currency;

  dbms_output.Put_line(currency);
END;

USD
```

The following statement deletes the rows from the table product_price_history where currency_code is ‘EUR’:

```
DELETE (SELECT * FROM product_price_history) WHERE  currency_code = 'EUR';
```

The following statement uses a subquery to delete rows from product_price_history:

```
DELETE product_price_history pp
WHERE  (product_id, currency_code, effective_from_date)
   IN (SELECT product_id, currency_code, Max(effective_from_date)
       FROM   product_price_history
       GROUP BY product_id, currency_code);
```

The following statement uses partitions to delete rows from product_price_history:

```
DELETE product_price_history partition (p1);
```

The following statement displays the table information:

```
SELECT * FROM product_price_history;

PRODUCT_ID    PRICE CUR EFFECTIVE EFFECTIVE
---------- ---------- --- --------- ---------
     1      100 USD 01-JAN-15 02-JAN-15
     1       60 GBP 01-JAN-15 02-JAN-15
```

The following statement deletes all rows from product_price_history:

```
DELETE product_price_history;
```

DISASSOCIATE STATISTICS

Purpose

Use the `DISASSOCIATE STATISTICS` statement to disassociate default statistics or a statistics type from columns, standalone functions, packages, types, domain indexes, or indextypes.

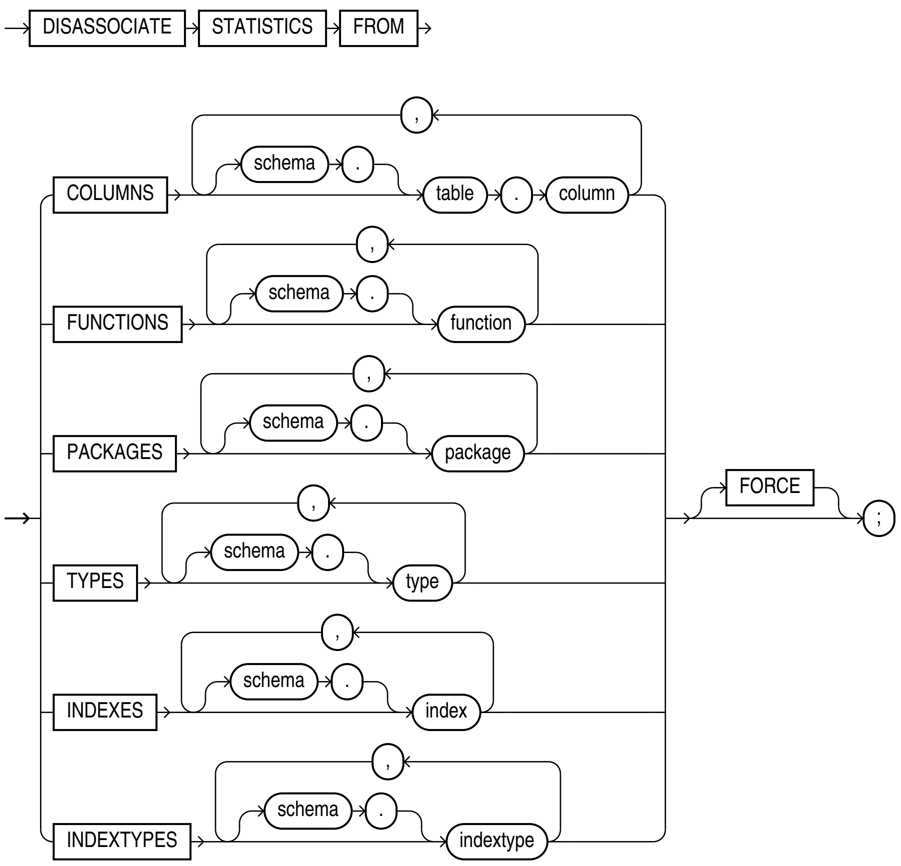
> **See Also:**
>
> ASSOCIATE STATISTICS for more information on statistics type associations

Prerequisites

To issue this statement, you must have the appropriate privileges to alter the underlying table, function, package, type, domain index, or indextype.

Syntax

disassociate_statistics::=



Semantics

FROM COLUMNS | FUNCTIONS | PACKAGES | TYPES | INDEXES | INDEXTYPES

Specify one or more columns, standalone functions, packages, types, domain indexes, or indextypes from which you are disassociating statistics.

If you do not specify `schema`, then Oracle Database assumes the object is in your own schema.

If you have collected user-defined statistics on the object, then the statement fails unless you specify `FORCE`.

FORCE

Specify `FORCE` to remove the association regardless of whether any statistics exist for the object using the statistics type. If statistics do exist, then the statistics are deleted before the association is deleted.

Note:

When you drop an object with which a statistics type has been associated, Oracle Database automatically disassociates the statistics type with the `FORCE` option and drops all statistics that have been collected with the statistics type.

Examples

Disassociating Statistics: Example

This statement disassociates statistics from the `emp_mgmt` package. See *Oracle Database PL/SQL Language Reference* for the example that creates this package in the `hr` schema.

```
DISASSOCIATE STATISTICS FROM PACKAGES hr.emp_mgmt;
```

DROP ANALYTIC VIEW

Purpose

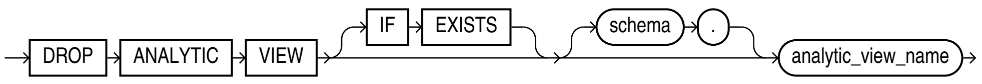
Use the `DROP ANALYTIC VIEW` statement to drop an analytic view. An `ANALYTIC VIEW` object is a component of analytic views.

Prerequisites

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

Syntax

***drop_analytic_view*::=**



Semantics

IF EXISTS

Specify `IF EXISTS` to drop an existing object.

Specifying `IF NOT EXISTS` with `DROP` results in `ORA-11544: Incorrect IF EXISTS clause for ALTER/DROP statement`.

schema

Specify the schema in which the analytic view exists. If you do not specify a schema, then Oracle Database looks for the analytic view in your own schema.

analytic_view_name

Specify the name of the analytic view to drop.

Example

The following statement drops the specified analytic view object:

```
DROP ANALYTIC VIEW sales_av;
```

DROP ATTRIBUTE DIMENSION

Purpose

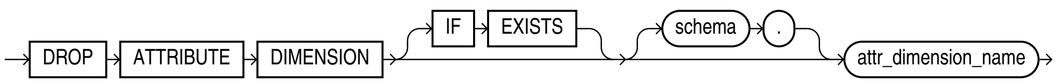
Use the `DROP ATTRIBUTE DIMENSION` statement to drop an attribute dimension. An `ATTRIBUTE DIMENSION` object is a component of analytic views.

Prerequisites

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

Syntax

***drop_attribute_dimension*::=**



Semantics

IF EXISTS

Specify `IF EXISTS` to drop an existing object.

Specifying `IF NOT EXISTS` with `DROP` results in `ORA-11544: Incorrect IF EXISTS clause for ALTER/DROP statement`.

schema

Specify the schema in which the attribute dimension exists. If you do not specify a schema, then Oracle Database looks for the attribute dimension in your own schema.

attr_dimension_name

Specify the name of the attribute dimension to drop.

Example

The following statement drops the specified attribute dimension object:

```
DROP ATTRIBUTE DIMENSION product_attr_dim;
```

DROP AUDIT POLICY (Unified Auditing)

This section describes the `DROP 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 `DROP AUDIT POLICY` statement to remove a unified audit policy from the database.

See Also:

- CREATE AUDIT POLICY (Unified Auditing)
- ALTER AUDIT POLICY (Unified Auditing)
- AUDIT (Unified Auditing)
- NOAUDIT (Unified Auditing)

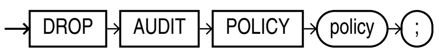
Prerequisites

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

To drop a common unified audit policy, the current container must be the root and you must have the commonly granted `AUDIT SYSTEM` privilege or the `AUDIT_ADMIN` common role. To drop a local unified audit policy, the current container must be the container in which the audit policy was created and 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.

Syntax

***drop_audit_policy*::=**



Semantics

policy

Specify the name of the unified audit policy you want to drop. The policy must have been created using the `CREATE AUDIT POLICY` statement.

You can find the names of all unified audit policies by querying the `AUDIT_UNIFIED_POLICIES` view and the names of all *enabled* unified audit policies by querying the `AUDIT_UNIFIED_ENABLED_POLICIES` view

Restriction on Dropping Unified Audit Policies

You cannot drop an enabled unified audit policy. You must first disable the policy using the `NOAUDIT` statement.

> **See Also:**
>
> - CREATE AUDIT POLICY (Unified Auditing)
> - *Oracle Database Reference* for more information on the `AUDIT_UNIFIED_POLICIES` and `AUDIT_UNIFIED_ENABLED_POLICIES` views

Examples

Dropping a Unified Audit Policy: Example

The following statement drops unified audit policy `table_pol`:

```
DROP AUDIT POLICY table_pol;
```

DROP CLUSTER

Purpose

Use the `DROP CLUSTER` clause to remove a cluster from the database.

> **Note:**
>
> When you drop a cluster, any tables in the recycle bin that were once part of that cluster are purged from the recycle bin and can no longer be recovered with a `FLASHBACK TABLE` operation.

You cannot uncluster an individual table. Instead you must perform these steps:

1. Create a new table with the same structure and contents as the old one, but with no `CLUSTER` clause.
2. Drop the old table.
3. Use the `RENAME` statement to give the new table the name of the old one.

4. Grant privileges on the new unclustered table. Grants on the old clustered table do not apply.

See Also:

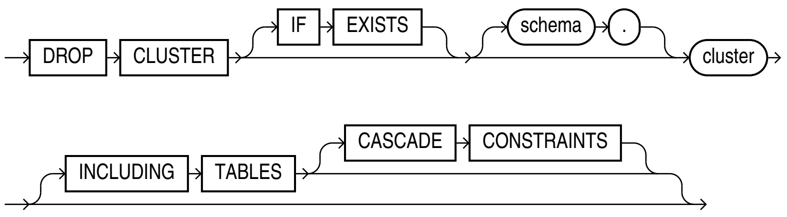
CREATE TABLE, DROP TABLE , RENAME , GRANT for information on these steps

Prerequisites

The cluster must be in your own schema or you must have the `DROP ANY CLUSTER` system privilege.

Syntax

***drop_cluster*::=**



Semantics

IF EXISTS

Specify `IF EXISTS` to drop an existing object.

Specifying `IF NOT EXISTS` with `DROP` results in `ORA-11544: Incorrect IF EXISTS clause for ALTER/DROP statement`.

schema

Specify the schema containing the cluster. If you omit *`schema`*, then the database assumes the cluster is in your own schema.

cluster

Specify the name of the cluster to be dropped. Dropping a cluster also drops the cluster index and returns all cluster space, including data blocks for the index, to the appropriate tablespace(s).

INCLUDING TABLES

Specify `INCLUDING TABLES` to drop all tables that belong to the cluster.

CASCADE CONSTRAINTS

Specify `CASCADE CONSTRAINTS` to drop all referential integrity constraints from tables outside the cluster that refer to primary and unique keys in tables of the cluster. If you omit this clause and such referential integrity constraints exist, then the database returns an error and does not drop the cluster.

Examples

Dropping a Cluster: Examples

The following examples drop the clusters created in the "Examples" section of `CREATE CLUSTER`.

The following statements drops the `language` cluster:

```
DROP CLUSTER language;
```

The following statement drops the `personnel` cluster as well as tables `dept_10` and `dept_20` and any referential integrity constraints that refer to primary or unique keys in those tables:

```
DROP CLUSTER personnel
   INCLUDING TABLES
   CASCADE CONSTRAINTS;
```