12 SQL Statements: ALTER SYNONYM to COMMENT

This chapter contains the following SQL statements:

ALTER SYNONYM

Purpose

Use the `ALTER SYNONYM` statement to modify an existing synonym.

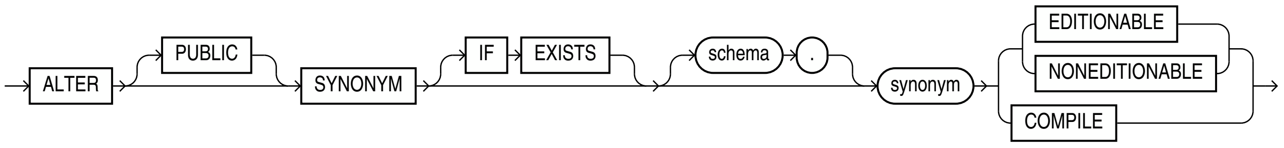
Prerequisites

To modify a private synonym in another user's schema, you must have the `CREATE ANY SYNONYM` and `DROP ANY SYNONYM` system privileges.

To modify a `PUBLIC` synonym, you must have the `CREATE PUBLIC SYNONYM` and `DROP PUBLIC SYNONYM` system privileges.

Syntax

***alter_synonym*::=**



Semantics

PUBLIC

Specify `PUBLIC` if *`synonym`* is a public synonym. You cannot use this clause to change a public synonym to a private synonym, or vice versa.

IF EXISTS

Specify `IF EXISTS` to alter an existing table.

Specifying `IF NOT EXISTS` with `ALTER VIEW` results in `ORA-11544: Incorrect IF EXISTS clause for ALTER/DROP statement`.

schema

Specify the schema containing the synonym. If you omit *`schema`*, then Oracle Database assumes the synonym is in your own schema.

synonym

Specify the name of the synonym to be altered.

EDITIONABLE | NONEDITIONABLE

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

Restriction on EDITIONABLE | NONEDITIONABLE

You cannot specify these clauses for a public synonym because editioning is always enabled for the object type `SYNONYM` in the `PUBLIC` schema.

COMPILE

Use this clause to compile *`synonym`*. A synonym places a dependency on its target object and becomes invalid if the target object is changed or dropped. When you compile an invalid synonym, it becomes valid again.

Note:

You can determine if a synonym is valid or invalid by querying the `STATUS` column of the `ALL_`, `DBA_`, and `USER_OBJECTS` data dictionary views.

Examples

The following examples modify synonyms that were created in the `CREATE SYNONYM` "Examples".

The following statement compiles synonym `offices`:

```
ALTER SYNONYM offices COMPILE;
```

The following statement compiles public synonym `emp_table`:

```
ALTER PUBLIC SYNONYM emp_table COMPILE;
```

The following statement causes synonym `offices` to remain a noneditioned object if editioning is later enabled for schema object type `SYNONYM` in the schema that contains the synonym `offices`:

```
ALTER SYNONYM offices NONEDITIONABLE;
```

ALTER SYSTEM

Purpose

Use the `ALTER SYSTEM` statement to dynamically alter your Oracle Database instance. The settings stay in effect as long as the database is mounted.

When you use the `ALTER SYSTEM` statement in a multitenant container database (CDB), you can specify some clauses to alter the CDB as a whole and other clauses to alter a specific pluggable database (PDB).

> **See Also:**
>
> *Oracle Database Administrator's Guide* for complete information on using the `ALTER SYSTEM` statement in a CDB

Prerequisites

To specify the `RELOCATE CLIENT` clause, you must be authenticated `AS SYSASM`.

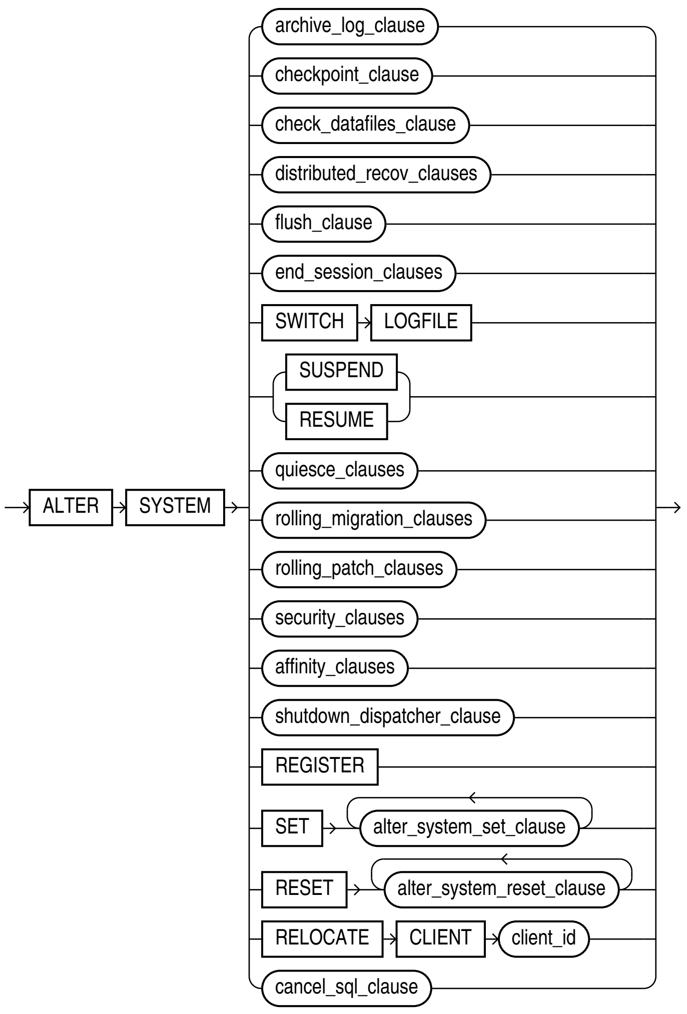
To specify all other clauses, you must have the `ALTER SYSTEM` system privilege.

If you are connected to a CDB:

- To alter the CDB as a whole, the current container must be the root and you must have the commonly granted `ALTER SYSTEM` privilege.
- To alter a PDB, the current container must be the PDB and you must have the `ALTER SYSTEM` privilege, either granted commonly or granted locally in the PDB.

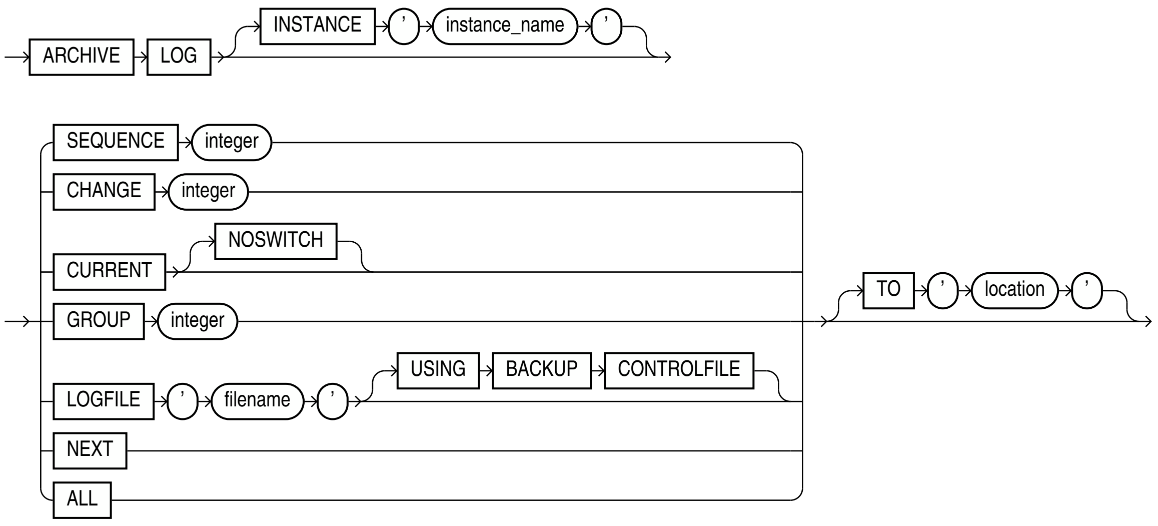
Syntax

alter_system::=

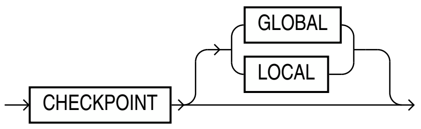


(*archive_log_clause*::=, *checkpoint_clause*::=, *check_datafiles_clause*::=, *distributed_recov_clauses*::=, *end_session_clauses*::=, *quiesce_clauses*::=, *rolling_migration_clauses*::=, *rolling_patch_clauses*::=, *security_clauses*::=, *shutdown_dispatcher_clause*::=, *alter_system_set_clause*::=, *alter_system_reset_clause*::=, cancel_sql_clause)

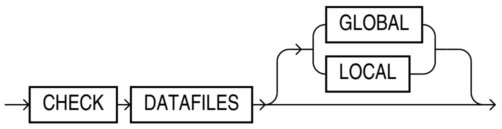
***archive_log_clause*::=**



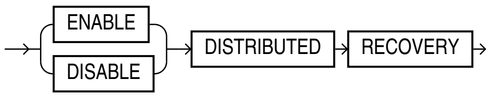
***checkpoint_clause*::=**



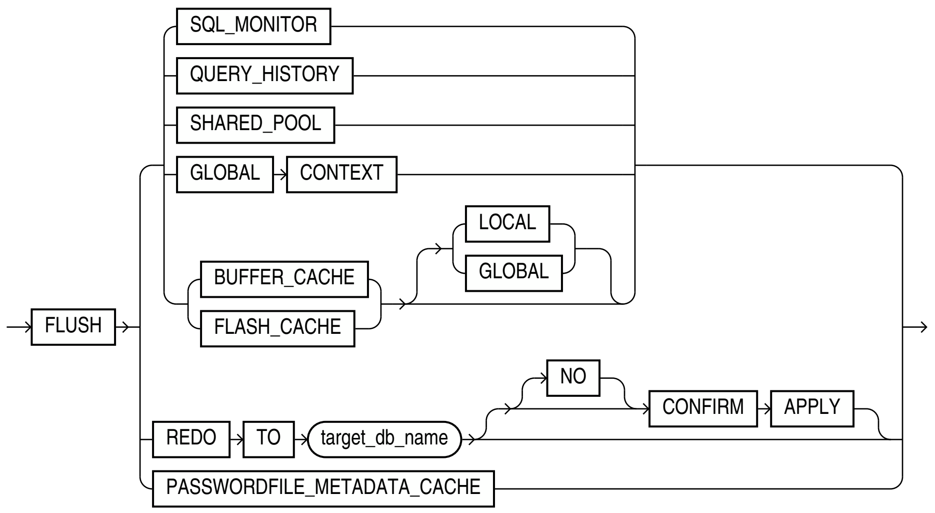
***check_datafiles_clause*::=**



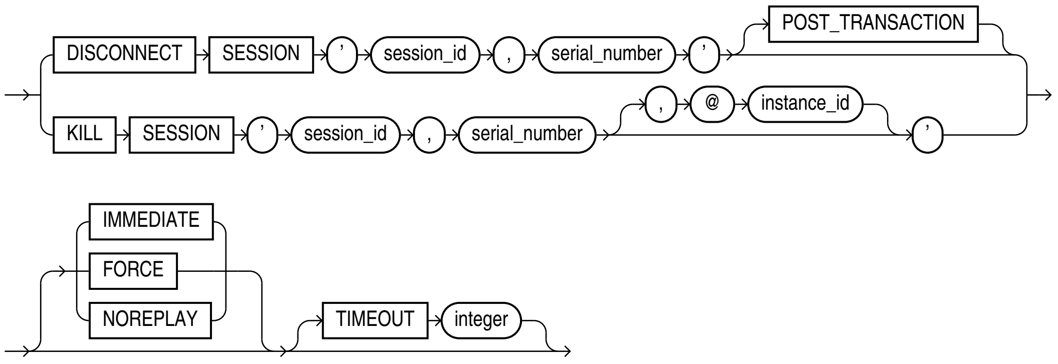
***distributed_recov_clauses*::=**



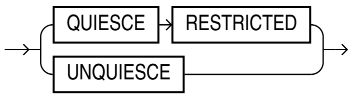
flush_clause



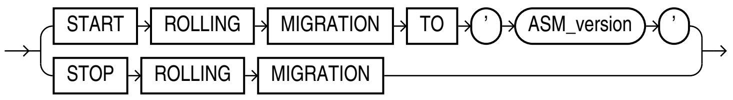
***end_session_clauses*::=**



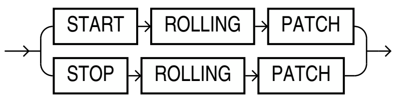
***quiesce_clauses*::=**



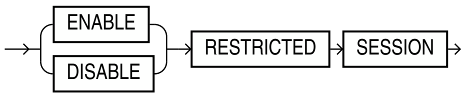
***rolling_migration_clauses*::=**



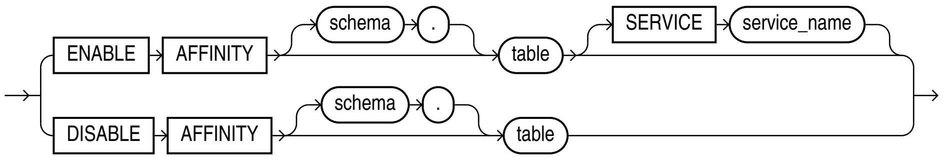
***rolling_patch_clauses*::=**



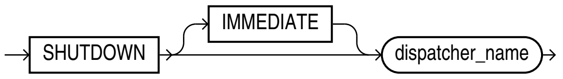
***security_clauses*::=**



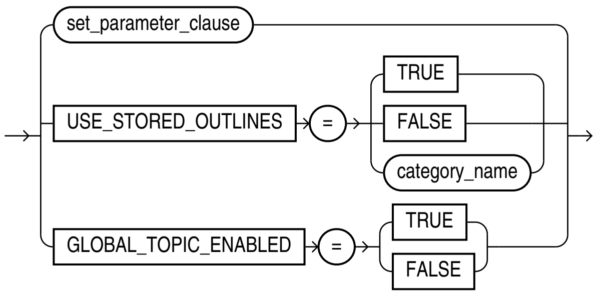
***affinity_clauses*::=**



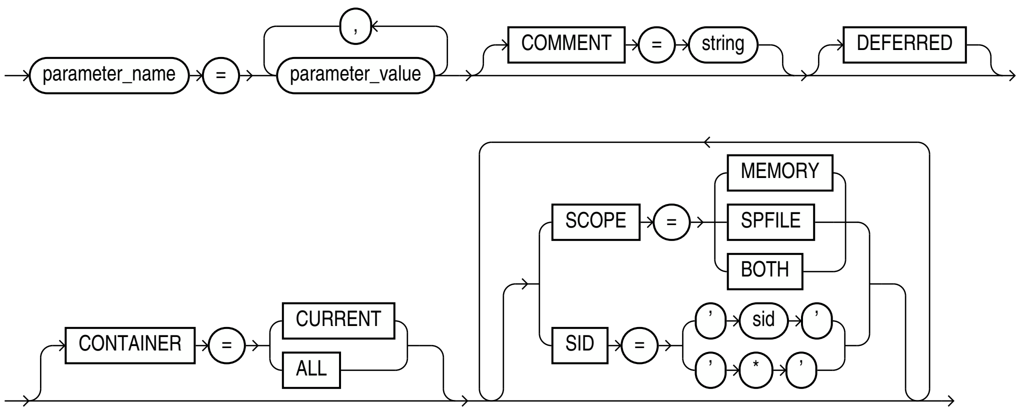
***shutdown_dispatcher_clause*::=**



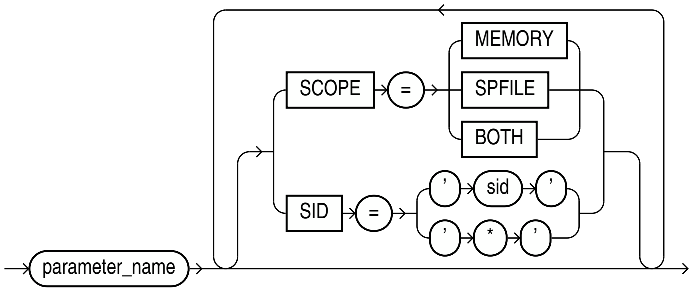
***alter_system_set_clause*::=**



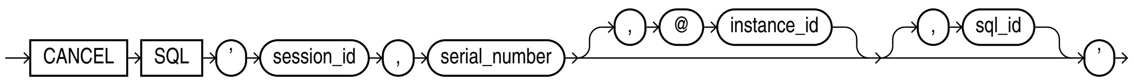
***set_parameter_clause*::=**



***alter_system_reset_clause*::=**



***cancel_sql_clause*::=**



Semantics

archive_log_clause

The `archive_log_clause` manually archives redo log files or enables or disables automatic archiving. To use this clause, your instance must have the database mounted. The database can be either open or closed unless otherwise noted.

INSTANCE Clause

This clause is relevant only if you are using Oracle Real Application Clusters (Oracle RAC). Specify the name of the instance for which you want the redo log file group to be archived. The instance name is a string of up to 80 characters. Oracle Database automatically determines the thread that is mapped to the specified instance and archives the corresponding redo log file group. If no thread is mapped to the specified instance, then Oracle Database returns an error.

SEQUENCE Clause

Specify `SEQUENCE` to manually archive the online redo log file group identified by the log sequence number *`integer`* in the specified thread. If you omit the `THREAD` parameter, then Oracle Database archives the specified group from the thread assigned to your instance.

CHANGE Clause

Specify `CHANGE` to manually archive the online redo log file group containing the redo log entry with the system change number (SCN) specified by *`integer`* in the specified thread. If the SCN is in the current redo log file group, then Oracle Database performs a log switch. If you omit the `THREAD` parameter, then Oracle Database archives the groups containing this SCN from all enabled threads.

You can use this clause only when your instance has the database open.

CURRENT Clause

Specify `CURRENT` to manually archive the current redo log file group of the specified thread, forcing a log switch. If you omit the `THREAD` parameter, then Oracle Database archives all redo log file groups from all enabled threads, including logs previous to current logs. You can specify `CURRENT` only when the database is open.

NOSWITCH

Specify `NOSWITCH` if you want to manually archive the current redo log file group without forcing a log switch. This setting is used primarily with standby databases to prevent data divergence when the primary database shuts down. Divergence implies the possibility of data loss in case of primary database failure.

You can use the `NOSWITCH` clause only when your instance has the database mounted but not open. If the database is open, then this operation closes the database automatically. You must then manually shut down the database before you can reopen it.

GROUP Clause

Specify `GROUP` to manually archive the online redo log file group with the `GROUP` value specified by *`integer`*. You can determine the `GROUP` value for a redo log file group by querying the dynamic performance view `V$LOG`. If you specify both the `THREAD` and `GROUP` parameters, then the specified redo log file group must be in the specified thread.

LOGFILE Clause

Specify `LOGFILE` to manually archive the online redo log file group containing the redo log file member identified by '*`filename`*'. If you specify both the `THREAD` and `LOGFILE` parameters, then the specified redo log file group must be in the specified thread.

If the database was mounted with a backup control file, then specify `USING BACKUP CONTROLFILE` to permit archiving of all online logfiles, including the current logfile.

Restriction on the LOGFILE clause

You must archive redo log file groups in the order in which they are filled. If you specify a redo log file group for archiving with the `LOGFILE` parameter, and earlier redo log file groups are not yet archived, then Oracle Database returns an error.

NEXT Clause

Specify `NEXT` to manually archive the next online redo log file group from the specified thread that is full but has not yet been archived. If you omit the `THREAD` parameter, then Oracle Database archives the earliest unarchived redo log file group from any enabled thread.

ALL Clause

Specify `ALL` to manually archive all online redo log file groups from the specified thread that are full but have not been archived. If you omit the `THREAD` parameter, then Oracle Database archives all full unarchived redo log file groups from all enabled threads.

TO *location* Clause

Specify `TO` '*`location`*' to indicate the primary location to which the redo log file groups are archived. The value of this parameter must be a fully specified file location following the conventions of your operating system. If you omit this parameter, then Oracle Database archives the redo log file group to the location specified by the initialization parameters `LOG_ARCHIVE_DEST` or `LOG_ARCHIVE_DEST_`*`n`*.

checkpoint_clause

Specify `CHECKPOINT` to explicitly force Oracle Database to perform a checkpoint, ensuring that all changes made by committed transactions are written to data files on disk. You can specify this clause only when your instance has the database open. Oracle Database does not return control to you until the checkpoint is complete.

GLOBAL

In an Oracle Real Application Clusters (Oracle RAC) environment, this setting causes Oracle Database to perform a checkpoint for all instances that have opened the database. This is the default.

LOCAL

In an Oracle RAC environment, this setting causes Oracle Database to perform a checkpoint only for the thread of redo log file groups for the instance from which you issue the statement.

> **See Also:**
>
> "Forcing a Checkpoint: Example"

check_datafiles_clause

In a distributed database system, such as an Oracle RAC environment, this clause updates an instance's SGA from the database control file to reflect information on all online data files.

- Specify `GLOBAL` to perform this synchronization for all instances that have opened the database. This is the default.
- Specify `LOCAL` to perform this synchronization only for the local instance.

Your instance should have the database open.

distributed_recov_clauses

The `DISTRIBUTED RECOVERY` clause lets you enable or disable distributed recovery. To use this clause, your instance must have the database open.

ENABLE

Specify `ENABLE` to enable distributed recovery. In a single-process environment, you must use this clause to initiate distributed recovery.

You may need to issue the `ENABLE DISTRIBUTED RECOVERY` statement more than once to recover an in-doubt transaction if the remote node involved in the transaction is not accessible. In-doubt transactions appear in the data dictionary view `DBA_2PC_PENDING`.

See Also:

"Enabling Distributed Recovery: Example"

DISABLE

Specify `DISABLE` to disable distributed recovery.

FLUSH SHARED_POOL Clause

The `FLUSH SHARED_POOL` clause lets you clear data from the shared pool in the system global area (SGA). The shared pool stores:

- Cached data dictionary information and
- Shared SQL and PL/SQL areas for SQL statements, stored procedures, functions, packages, and triggers.

This statement does not clear global application context information, nor does it clear shared SQL and PL/SQL areas for items that are currently being executed. You can use this clause regardless of whether your instance has the database dismounted or mounted, open or closed.

See Also:

"Clearing the Shared Pool: Example"

FLUSH GLOBAL CONTEXT Clause

The `FLUSH GLOBAL CONTEXT` clause lets you flush all global application context information from the shared pool in the system global area (SGA). You can use this clause regardless of whether your instance has the database dismounted or mounted, open or closed.

FLUSH BUFFER_CACHE Clause

The `FLUSH BUFFER_CACHE` clause lets you clear all data from the buffer cache in the system global area (SGA), including the `KEEP`, `RECYCLE`, and `DEFAULT` buffer pools.

Specify `LOCAL` if you only want to flush the local instance. To flush the buffer cache of all instances, specify `GLOBAL`. `GLOBAL` is the default.

Note:

This clause is intended for use only on a test database. Do not use this clause on a production database, because as a result of this statement, subsequent queries will have no hits, only misses.

This clause is useful if you need to measure the performance of rewritten queries or a suite of queries from identical starting points.

FLUSH FLASH_CACHE Clause

Use the `FLUSH FLASH_CACHE` clause to flush the Database Smart Flash Cache. This clause can be useful if you need to measure the performance of rewritten queries or a suite of queries from identical starting points, or if there might be corruption in the cache.

Specify `LOCAL` if you only want to flush the local instance. To flush the flash cache of all instances, specify `GLOBAL`. `GLOBAL` is the default.

FLUSH REDO Clause

Use the `FLUSH REDO` clause to flush redo data from a primary database to a standby database and to optionally wait for the flushed redo data to be applied to a physical or logical standby database.

This clause can allow a failover to be performed on the target standby database without data loss, even if the primary database is not in a zero data loss data protection mode, provided that all redo data that has been generated by the primary database can be flushed to the standby database.

The `FLUSH REDO` clause must be issued on a mounted, but not open, primary database.

target_db_name

For *`target_db_name`*, specify the `DB_UNIQUE_NAME` of the standby database that is to receive the redo data flushed from the primary database.

The value of the `LOG_ARCHIVE_DEST_`*`n`* database initialization parameter that corresponds to the target standby database must contain the `DB_UNIQUE_NAME` attribute, and the value of that attribute must match the `DB_UNIQUE_NAME` of the target standby database.

NO CONFIRM APPLY

If you specify this clause, then the `ALTER SYSTEM` statement will not complete until the standby database has received all of the flushed redo data. You must specify this clause if the target standby database is a snapshot standby database.

CONFIRM APPLY

If you specify this clause, then the `ALTER SYSTEM` statement will not complete until the target standby database has received and applied all flushed redo data. This is the default behavior unless you specify `NO CONFIRM APPLY`. You cannot specify this clause if the target standby database is a snapshot standby database.

See Also:

Oracle Data Guard Concepts and Administration for more information about the `FLUSH REDO` clause and failovers

FLUSH PASSWORDFILE_METADATA_CACHE

If the location or the name of the password file changes, you must notify the database that a change has occurred. The command `ALTER SYSTEM FLUSH PASSWORDFILE_METADATA_CACHE` flushes the password file metadata cache stored in the SGA and informs the database that a change has occurred.

The command also flushes the cache from all the RAC instances if it is run in a cluster environment. Note the delay in propagating the change across all instances. Until the flush is fully propagated, some instances might continue to use the old password file.

end_session_clauses

The `end_session_clauses` give you several ways to end the current session.

DISCONNECT SESSION Clause

Use the `DISCONNECT SESSION` clause to disconnect the current session by destroying the dedicated server process (or virtual circuit if the connection was made by way of a Shared Server). To use this clause, your instance must have the database open. You must identify the session with both of the following values from the `V$SESSION` view:

- For *`session_id`*, specify the value of the `SID` column.
- For *`serial_number`*, specify the value of the `SERIAL#` column.

If system parameters are appropriately configured, then application failover will take effect.

- The `POST_TRANSACTION` setting allows ongoing transactions to complete before the session is disconnected. If the session has no ongoing transactions, then this clause has the same effect described for as `KILL SESSION`.
- The `IMMEDIATE` setting disconnects the session and recovers the entire session state immediately, without waiting for ongoing transactions to complete.
  - If you also specify `POST_TRANSACTION` and the session has ongoing transactions, then the `IMMEDIATE` keyword is ignored.
  - If you do not specify `POST_TRANSACTION`, or you specify `POST_TRANSACTION` but the session has no ongoing transactions, then this clause has the same effect as described for `KILL SESSION IMMEDIATE`.

See Also:

"Disconnecting a Session: Example"

KILL SESSION Clause

The `KILL SESSION` clause lets you mark a session as terminated, roll back ongoing transactions, release all session locks, and partially recover session resources. To use this clause, your instance must have the database open. Your session and the session to be terminated must be on the same instance unless you specify *`integer3`*.You must identify the session with the following values from the `V$SESSION` view:

- For *`session_id`*, specify the value of the `SID` column.
- For *`serial_number`*, specify the value of the `SERIAL#` column.
- For the optional *`instance_id`*, specify the ID of the instance where the target session to be killed exists. You can find the instance ID by querying the GV$ tables.

If the session is performing some activity that must be completed, such as waiting for a reply from a remote database or rolling back a transaction, then Oracle Database waits for this activity to complete, marks the session as terminated, and then returns control to you. If the waiting lasts a minute, then Oracle Database marks the session to be terminated and returns control to you with a message that the session is marked to be terminated. The `PMON` background process then marks the session as terminated when the activity is complete.

Whether or not the session has an ongoing transaction, Oracle Database does not recover the entire session state until the session user issues a request to the session and receives a message that the session has been terminated.

See Also:

"Terminating a Session: Example"

IMMEDIATE

Specify `IMMEDIATE` to instruct Oracle Database to roll back ongoing transactions, release all session locks, recover the entire session state, and return control to you immediately.

Note that `IMMEDIATE` only returns control immediately, if `TIMEOUT` is not specified.

`IMMEDIATE` is similar to the case when the session is deleted without a modifier in that it waits until the activity completes. Once the activity completes, the full session is deleted without waiting for the session user and the connection is closed.

FORCE

`FORCE` is similar to `IMMEDIATE` except that `FORCE` will forcefully terminate the connection if a timeout occurs.

Example

```
ALTER SYSTEM KILL SESSION '20,1' FORCE;
```

NOREPLAY

This clause is valid if you are using Application Continuity. When connected to a service with Application Continuity enabled (that is, `FAILOVER_TYPE = TRANSACTION`), the session is recovered after the session fails or is killed. If you do not want to recover a session after it is terminated, then specify `NOREPLAY`.

TIMEOUT

Specify `TIMEOUT` to set the maximum amount of time (in seconds) to wait before terminating the session. It overrides the default timeout.

The current default timeout values are:

- 60 seconds when no modifier is specified
- 0 seconds when the modifier `IMMEDIATE` is specified
- 5 seconds when the modifier `FORCE` is specified

The action that occurs at `TIMEOUT` is different for `IMMEDIATE`, which marks the session for termination and `FORCE` , which forcefully terminates the session.

Example

```
ALTER SYSTEM KILL SESSION '20,1' TIMEOUT 20;
```

SWITCH LOGFILE Clause

The `SWITCH LOGFILE` clause lets you explicitly force Oracle Database to begin writing to a new redo log file group, regardless of whether the files in the current redo log file group are full. When you force a log switch, Oracle Database begins to perform a checkpoint but returns control to you immediately rather than when the checkpoint is complete. To use this clause, your instance must have the database open.

> **See Also:**
>
> "Forcing a Log Switch: Example"

SUSPEND | RESUME

The `SUSPEND` clause lets you suspend all I/O (data file, control file, and file header) as well as queries, in all instances, enabling you to make copies of the database without having to handle ongoing transactions.

Restrictions on SUSPEND and RESUME

`SUSPEND` and `RESUME` are subject to the following restrictions:

- Do not use this clause unless you have put the database tablespaces in hot backup mode.
- Do not terminate the session that issued the `ALTER SYSTEM SUSPEND` statement. An attempt to reconnect while the system is suspended may fail because of recursive SQL that is running during the `SYS` login.
- If you start a new instance while the system is suspended, then that new instance will not be suspended.

The `RESUME` clause lets you make the database available once again for queries and I/O.

quiesce_clauses

Use the `QUIESCE RESTRICTED` and `UNQUIESCE` clauses to put the database in and take it out of the **quiesced state**. This state enables database administrators to perform administrative operations that cannot be safely performed in the presence of concurrent transactions, queries, or PL/SQL operations.

> **Note:**
>
> The `QUIESCE RESTRICTED` clause is valid only if the Database Resource Manager is installed and only if the Resource Manager has been on continuously since database startup in any instances that have opened the database.

If multiple `QUIESCE RESTRICTED` or `UNQUIESCE` statements issue at the same time from different sessions or instances, then all but one will receive an error.

QUIESCE RESTRICTED

Specify `QUIESCE RESTRICTED` to put the database in the quiesced state. For all instances with the database open, this clause has the following effect:

- Oracle Database instructs the Database Resource Manager in all instances to prevent all inactive sessions (other than `SYS` and `SYSTEM`) from becoming active. No user other than `SYS` and `SYSTEM` can start a new transaction, a new query, a new fetch, or a new PL/SQL operation.
- Oracle Database waits for all existing transactions in all instances that were initiated by a user other than `SYS` or `SYSTEM` to finish (either commit or abort). Oracle Database also waits for all running queries, fetches, and PL/SQL procedures in all instances that were initiated by users other than `SYS` or `SYSTEM` and that are not inside transactions to finish. If a query is carried out by multiple successive OCI fetches, then Oracle Database does not wait for all fetches to finish. It waits for the current fetch to finish and then blocks the next fetch. Oracle Database also waits for all sessions (other than those of `SYS` or `SYSTEM`) that hold any shared resources (such as enqueues) to release those resources. After all these operations finish, Oracle Database places the database into quiesced state and finishes executing the `QUIESCE RESTRICTED` statement.
- If an instance is running in shared server mode, then Oracle Database instructs the Database Resource Manager to block logins (other than `SYS` or `SYSTEM`) on that instance. If an instance is running in non-shared-server mode, then Oracle Database does not impose any restrictions on user logins in that instance.

During the quiesced state, you cannot change the Resource Manager plan in any instance.

UNQUIESCE

Specify `UNQUIESCE` to take the database out of quiesced state. Doing so permits transactions, queries, fetches, and PL/SQL procedures that were initiated by users other than `SYS` or `SYSTEM` to be undertaken once again. The `UNQUIESCE` statement does not have to originate in the same session that issued the `QUIESCE RESTRICTED` statement.

rolling_migration_clauses

Use these clauses in a clustered Oracle Automatic Storage Management (Oracle ASM) environment to migrate one node at a time to a different Oracle ASM version without affecting the overall availability of the Oracle ASM cluster or the database clusters using Oracle ASM for storage.

START ROLLING MIGRATION

When starting rolling upgrade, for *`ASM_version`*, you must specify the following string:

```
'<version_num>, <release_num>, <update_num>,<port_release_num>,<port_update_num>'
```

`ASM_version` must be equal to or greater than 11.1.0.0.0. The surrounding single quotation marks are required. Oracle ASM first verifies that the current release is compatible for migration to the specified release, and then goes into limited functionality mode. Oracle ASM then determines whether any rebalance operations are under way anywhere in the cluster. If there are any such operations, then the statement fails and must be reissued after the rebalance operations are complete.

Rolling upgrade mode is a cluster-wide In-Memory persistent state. The cluster continues to be in this state until there is at least one Oracle ASM instance running in the cluster. Any new instance joining the cluster switches to migration mode immediately upon startup. If all the instances in the cluster terminate, then subsequent startup of any Oracle ASM instance will not be in rolling upgrade mode until you reissue this statement to restart rolling upgrade of the Oracle ASM instances.

STOP ROLLING MIGRATION

Use this clause to stop rolling upgrade and bring the cluster back into normal operation. Specify this clause only after all instances in the cluster have migrated to the same software version. The statement will fail if the cluster is not in rolling upgrade mode.

When you specify this clause, the Oracle ASM instance validates that all the members of the cluster are at the same software version, takes the instance out of rolling upgrade mode, and returns to full functionality of the Oracle ASM cluster. If any rebalance operations are pending because disks have gone offline, then those operations are restarted if the `ASM_POWER_LIMIT` parameter would not be violated by such a restart.

See Also:

Oracle Automatic Storage Management Administrator's Guide for more information about rolling upgrade

rolling_patch_clauses

Use these clauses in a clustered Oracle Automatic Storage Management (Oracle ASM) environment to update one node at a time to the latest patch level without affecting the overall availability of the Oracle ASM cluster or the database clusters using Oracle ASM for storage.

START ROLLING PATCH

Use this clause to start the rolling patch operation. Oracle ASM first verifies that all live nodes in the cluster are at the same version, and then goes into rolling patch mode, which is a cluster-wide In-Memory persistent state. The cluster continues to be in this state until all live nodes have been patched to the latest patch level.

Any nodes that are down during this operation are not patched. This does not affect the success of the rolling patch operation. However, you must patch these nodes before they are started. Otherwise, they will not be allowed to join the cluster.

STOP ROLLING PATCH

use this clause to stop the rolling patch operation and bring the cluster back into normal operation. Specify this clause only after all live nodes in the cluster have been patched to the latest patch level. The statement will fail if the cluster is not in rolling patch mode.

When you specify this clause, the Oracle ASM instance validates that all members of the cluster are at the same patch level, takes the instance out of rolling patch mode, and returns full functionality of the Oracle ASM cluster. If any members of the cluster are not at the latest patch level, then this operation fails and the cluster goes into limited functionality mode.

The following queries display information about rolling patches. In order to run these queries, you must be connected to the Oracle ASM instance in the Grid home, and the Grid Infrastructure home must be configured with the Oracle Clusterware option for an Oracle RAC environment.

- You can determine whether a cluster is in rolling patch mode with the following query:

  ```
  SELECT SYS_CONTEXT('SYS_CLUSTER_PROPERTIES', 'CLUSTER_STATE') FROM DUAL;
  ```

- You can determine the patch level of a cluster with the following query:

  ```
  SELECT SYS_CONTEXT('SYS_CLUSTER_PROPERTIES', 'CURRENT_PATCHLVL') FROM DUAL;
  ```

- You can display a list of patches applied on the Oracle ASM instance, by querying the `V$PATCHES` dynamic performance view. Refer to *Oracle Database Reference* for more information.

See Also:

Oracle Automatic Storage Management Administrator's Guide for more information about rolling patches

security_clauses

The *security_clauses* let you control access to the instance. They also allow you to enable or disable access to the encrypted data in the instance.

RESTRICTED SESSION

The RESTRICTED SESSION clause lets you restrict logon to Oracle Database. You can use this clause regardless of whether your instance has the database dismounted or mounted, open or closed.

- Specify ENABLE to allow only users with RESTRICTED SESSION system privilege to log on to Oracle Database. Existing sessions are not terminated.

  This clause applies only to the current instance. Therefore, in an Oracle RAC environment, authorized users without the RESTRICTED SESSION system privilege can still access the database by way of other instances.
- Specify DISABLE to reverse the effect of the ENABLE RESTRICTED SESSION clause, allowing all users with CREATE SESSION system privilege to log on to Oracle Database. This is the default.

See Also:

- "Restricting Sessions: Example"
- The description of the CREATE TABLE "*encryption_spec* " for information on using that feature to encrypt table columns
- "Establishing a Wallet and Encryption Key: Examples"

affinity_clauses

Use the affinity clauses to enable data-dependent routing to provide cache affinity on a RAC database. The affinity logically partitions data across RAC instances so that a distinct subset of data is assigned to each instance. When data is accessed with a sharding key, the request will be routed to the instance that holds the corresponding subset of data. The benefits of affinity are:

- Sharded access for shard-aware applications and transparency for non-sharded applications
- Better cache utilization and reduced block pings

shutdown_dispatcher_clause

The SHUTDOWN clause is relevant only if your system is using the shared server architecture of Oracle Database. It shuts down a dispatcher identified by *dispatcher_name*.

Note:

Do not confuse this clause with the SQL*Plus command `SHUTDOWN`, which is used to shut down the entire database.

The *`dispatcher_name`* must be a string of the form '`D`*`xxx`*', where *`xxx`* indicates the number of the dispatcher. For a listing of dispatcher names, query the `NAME` column of the `V$DISPATCHER` dynamic performance view.

- If you specify `IMMEDIATE`, then the dispatcher stops accepting new connections immediately and Oracle Database terminates all existing connections through that dispatcher. After all sessions are cleaned up, the dispatcher process shuts down.
- If you do not specify `IMMEDIATE`, then the dispatcher stops accepting new connections immediately but waits for all its users to disconnect and for all its database links to terminate. Then it shuts down.

REGISTER Clause

Specify `REGISTER` to instruct the `PMON` background process to register the instance with the listeners immediately. If you do not specify this clause, then registration of the instance does not occur until the next time `PMON` executes the discovery routine. As a result, clients may not be able to access the services for as long as 60 seconds after the listener is started.

See Also:

Oracle Database Concepts and *Oracle Database Net Services Administrator's Guide* for information on the `PMON` background process and listeners

alter_system_set_clause

This clause allows you to change parameter values. The *`set_parameter_clause`* allows you to change the value of a specified initialization parameter. The `USE_STORED_OUTLINES` and `GLOBAL_TOPIC_ENABLED` clauses allow you to change the value of those system parameters.

set_parameter_clause

You can change the value of many initialization parameters for the current instance, whether you have started the database with a traditional plain-text parameter file (pfile) or with a server parameter file (spfile). *Oracle Database Reference* indicates these parameters in the "Modifiable" category of each parameter description. If you are using a pfile, then the change will persist only for the duration of the instance. However, if you have started the database with an spfile, then you can change the value of the parameter in the spfile itself, so that the new value will occur in subsequent instances.

Oracle Database Reference documents all initialization parameters in full. The parameters fall into three categories:

- **Basic parameters:** Database administrators should be familiar with and consider the setting for all of the basic parameters.
- **Functional categories:** Oracle Database Reference also lists the initialization parameters by their functional category.

- **Alphabetical listing:** The Table of Contents of *Oracle Database Reference* contains all initialization parameters in alphabetical order.

The ability to change initialization parameter values depends on whether you have started up the database with a traditional plain-text initialization parameter file (pfile) or with a server parameter file (spfile). To determine whether you can change the value of a particular parameter, query the `ISSYS_MODIFIABLE` column of the `V$PARAMETER` dynamic performance view.

If you want to enforce case on parameter values that are string literals, you must enclose them within single quotes.

You can enforce the minimum password length for database user accounts across the entire CDB or individual PDBs by setting the `MANDATORY_USER_PROFILE` parameter in the `init.ora` file.

Example

This statement sets the `MANDATORY_USER_PROFILE` parameter to the mandatory profile `c##cdb_profile` for all the PDBs in the CDB:

```
ALTER SYSTEM SET MANDATORY_USER_PROFILE=c##cdb_profile;
```

Only a common user who has been commonly granted the `ALTER SYSTEM` privilege or has the`SYSDBA` administrative privilege can modify the `MANDTORY_USER_PROFILE` in the `init.ora` file.

See Also:

- CREATE PROFILE
- Managing Security for Oracle Databases

When setting a parameter value, you can specify additional settings as follows:

COMMENT

The `COMMENT` clause lets you associate a comment string with this change in the value of the parameter. The comment string cannot contain control characters or a line break. If you also specify `SPFILE`, then this comment will appear in the parameter file to indicate the most recent change made to this parameter.

DEFERRED

The `DEFERRED` keyword sets or modifies the value of the parameter for future sessions that connect to the database. Current sessions retain the old value.

You must specify `DEFERRED` if the value of the `ISSYS_MODIFIABLE` column of `V$PARAMETER` for this parameter is `DEFERRED`. If the value of that column is `IMMEDIATE`, then the `DEFERRED` keyword in this clause is optional. If the value of that column is `FALSE`, then you cannot specify `DEFERRED` in this `ALTER SYSTEM` statement.

See Also:

Oracle Database Reference for information on the `V$PARAMETER` dynamic performance view

CONTAINER

You can specify the `CONTAINER` clause when you set a parameter value in a CDB. A CDB uses an inheritance model for initialization parameters in which PDBs inherit initialization parameter values from the root. In this case, inheritance means that the value of a particular parameter in the root applies to a particular PDB.

A PDB can override the root's setting for some parameters, which means that a PDB has an inheritance property for each initialization parameter that is either true or false. The inheritance property is true for a parameter when the PDB inherits the root's value for the parameter. The inheritance property is false for a parameter when the PDB does not inherit the root's value for the parameter.

The inheritance property for some parameters must be true. For other parameters, you can change the inheritance property by running the `ALTER SYSTEM SET` statement to set the parameter when the current container is the PDB. If `ISPDB_MODIFIABLE` is `TRUE` for an initialization parameter in the `V$SYSTEM_PARAMETER` view, then the inheritance property can be false for the parameter.

- If you specify `CONTAINER = ALL`, then the parameter setting applies to all containers in the CDB, including the root and all of the PDBs. The current container must be the root.

  Specifying `ALL` sets the inheritance property to true for the parameter in all PDBs.
- If you specify `CONTAINER = CURRENT`, then the parameter setting applies only to the current container. When the current container is the root, the parameter setting applies to the root and to any PDB with an inheritance property of true for the parameter.

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

See Also:

Oracle Database Administrator's Guide for more information on modifying parameters in a CDB

SCOPE

The `SCOPE` clause lets you specify when the change takes effect. The behavior of this clause depends on whether you are connected to a non-CDB, a CDB root, or a PDB.

When you issue the ALTER SYSTEM statement while connected to a non-CDB or a CDB root, the scope depends on whether you started up the database using a traditional plain-text parameter file (pfile) or server parameter file (spfile).

- `MEMORY` indicates that the change is made in memory, takes effect immediately, and persists until the database is shut down. If you started up the database using a parameter file (pfile), then this is the only scope you can specify.

  Note that `MEMORY` makes changes in memory of all the instances and overwrites values set individually on the instance.

- `SPFILE` indicates that the change is made in the server parameter file. The new setting takes effect when the database is next shut down and started up again. You must specify `SPFILE` when changing the value of a static parameter that is described as not modifiable in *Oracle Database Reference*.

  Note that `SPFILE` makes no changes in memory, which means that the instance parameter set individually on the instance takes precedence over global.
- `BOTH` indicates that the change is made in memory and in the server parameter file. The new setting takes effect immediately and persists after the database is shut down and started up again.

  Note that `BOTH` makes changes in memory of all the instances and overwrites values set individually on the instance, until the instance is restarted. When the instance is restarted, the spfile is read and then the instance parameter takes precedence.

If a server parameter file was used to start up the database, then `BOTH` is the default. If a parameter file was used to start up the database, then `MEMORY` is the default, as well as the only scope you can specify.

When you issue the ALTER SYSTEM statement while connected to a PDB, you can modify only initialization parameters for which the `ISPDB_MODIFIABLE` column is `TRUE` in the `V$SYSTEM_PARAMETER` view. The initialization parameter value takes effect only for the PDB. For any initialization parameter that is not set explicitly for a PDB, the PDB inherits the CDB root's parameter value.

- `MEMORY` indicates that the change is made in memory and takes effect immediately in the PDB. The setting reverts to the value set in the CDB root in the any of the following cases:
  - An `ALTER SYSTEM SET` statement sets the value of the parameter in the root with `SCOPE` equal to `BOTH` or `MEMORY`, and the PDB is closed and reopened. The parameter value in the PDB is not changed if `SCOPE` is equal to `SPFILE`, and the PDB is closed and reopened.
  - The PDB is closed and reopened.
  - The CDB is shut down and reopened.
- `SPFILE` indicates that the change is made for the PDB and stored persistently. The new setting affects only the PDB and takes effect in either of the following cases:
  - The PDB is closed and reopened.
  - The CDB is shut down and reopened.
- `BOTH` indicates that the change is made in memory, made for the PDB, and stored persistently. The new setting takes effect immediately in the PDB and persists after the PDB is closed and reopened or the CDB is shut down and reopened. The new setting affects only the PDB.

When a PDB is unplugged from a CDB, the values of the initialization parameters that were specified for the PDB with `SCOPE=BOTH` or `SCOPE=SPFILE` are added to the PDB's XML metadata file. These values are restored for the PDB when it is plugged in to a CDB.

Note:

Oracle may internally adjust the parameter value passed in `ALTER SYSTEM SET` before it is set in memory or the spfile. For example, if you input a non-prime number when the paramenter value should be a prime number, Oracle will adjust the value to the next prime number. You can query the parameter value from parameter views `V$PARAMETER`, `V$SYSTEM_PARAMETER`, and `V$SPPARAMETER`.

SID

The `SID` clause lets you specify the SID of the instance where the value will take effect.

- Specify `SID` = `'*'` if you want Oracle Database to change the value of the parameter for all instances that do not already have an explicit setting for this parameter.
- Specify `SID` = *`'sid'`* if you want Oracle Database to change the value of the parameter only for the instance *`sid`*. This setting takes precedence over previous and subsequent `ALTER SYSTEM SET` statements that specify `SID` = `'*'`.

If you do not specify this clause, then:

- If the instance was started up with a pfile (traditional plain-text initialization parameter file), then Oracle Database assumes the SID of the current instance.
- If the instance was started up with an spfile (server parameter file), then Oracle Database assumes `SID` = `'*'`.

If you specify an instance other than the current instance, then Oracle Database sends a message to that instance to change the parameter value in the memory of that instance.

USE_STORED_OUTLINES Clause

Note:

Stored outlines are deprecated. They are still supported for backward compatibility. However, Oracle recommends that you use SQL plan management instead. Refer to *Oracle Database SQL Tuning Guide* for more information about SQL plan management.

`USE_STORED_OUTLINES` is a system parameter, not an initialization parameter. You cannot set it in a pfile or spfile, but you can set it with an `ALTER SYSTEM` statement. This parameter determines whether the optimizer will use stored public outlines to generate execution plans.

- `TRUE` causes the optimizer to use outlines stored in the `DEFAULT` category when compiling requests.
- `FALSE` specifies that the optimizer should not use stored outlines. This is the default.
- *`category_name`* causes the optimizer to use outlines stored in the *`category_name`* category when compiling requests.

GLOBAL_TOPIC_ENABLED

`GLOBAL_TOPIC_ENABLED` is a system parameter, not an initialization parameter. You cannot set it in a pfile or spfile, but you can set it with an `ALTER SYSTEM` statement. If `GLOBAL_TOPIC_ENABLED`

= TRUE when a queue table is created, altered, or dropped, then the corresponding Lightweight Directory Access Protocol (LDAP) entry is also created, altered or dropped.

The parameter works the same way for the Java Message Service (JMS). If a database has been configured to use LDAP and the GLOBAL_TOPIC_ENABLED parameter has been set to TRUE, then all JMS queues and topics are automatically registered with the LDAP server when they are created. The administrator can also create aliases to the queues and topics registered in LDAP. Queues and topics that are registered in LDAP can be looked up through JNDI using the name or alias of the queue or topic.

Shared Server Parameters

When you start your instance, Oracle Database creates shared server processes and dispatcher processes for the shared server architecture based on the values of the SHARED_SERVERS and DISPATCHERS initialization parameters. You can also set the SHARED_SERVERS and DISPATCHERS parameters with ALTER SYSTEM to perform one of the following operations while the instance is running:

- Create additional shared server processes by increasing the minimum number of shared server processes.
- Terminate existing shared server processes after their current calls finish processing.
- Create more dispatcher processes for a specific protocol, up to a maximum across all protocols specified by the initialization parameter MAX_DISPATCHERS.
- Terminate existing dispatcher processes for a specific protocol after their current user processes disconnect from the instance.

See Also:

- *Oracle Real Application Clusters Administration and Deployment Guide* for information on setting parameter values for an individual instance in an Oracle Real Application Clusters environment
- The following examples of using the ALTER SYSTEM statement: "Changing Licensing Parameters: Examples", "Enabling Query Rewrite: Example", "Enabling Resource Limits: Example", "Shared Server Parameters", and "Changing Shared Server Settings: Examples"

alter_system_reset_clause

This clause lets you reset an initialization parameter.

The semantics of this clause are similar to the *set_parameter_clause*, except instead of changing the value of an initialization parameter, this clause removes the setting of an initialization parameter. Refer to the set_parameter_clause to learn about the parameters you can reset, and for the full semantics of the SCOPE and SID clauses.

RELOCATE CLIENT

This clause is valid only if you are using Oracle Flex ASM. You must issue this clause from within an Oracle ASM instance, not from a normal database instance.

Use this clause to relocate the specified client to the least loaded Oracle ASM instance. When you issue this clause, the connection to the client is terminated and the client fails over to the least loaded instance. If the client is currently connected to the least loaded instance, then the connection to the client is terminated and the client fails over to that same instance.

For *client_id*, specify a string of the following form enclosed in single quotation marks:

```
instance_name:db_name
```

where *instance_name* is the identifier for the client and *db_name* is the database name for the client. You can find these values by querying the `INSTANCE_NAME` and `DB_NAME` columns of the `V$ASM_CLIENT` dynamic performance view.

See Also:

- *Oracle Automatic Storage Management Administrator's Guide* for more information on managing Oracle Flex ASM
- *Oracle Database Reference* for more information on the `V$ASM_CLIENT` dynamic performance view

cancel_sql_clause

Use this clause to terminate a SQL operation that is consuming excessive resources, including parallel servers. You must provide the session id and the session serial number of the session whose active SQL statement you want to cancel. If the session is idle (no actively running SQL statement), the next SQL statement will be canceled. To avoid the next SQL statement from getting canceled, specify the `sql_id` in the arguments to identify the SQL statement to be canceled.

- `session_id` is required and stands for the session identifier.
- `serial_number` is required and stands for the serial number of the session.
- `instance_id` is optional. If this argument is omitted, the instance id of the current session is used.
- `sql_id` is optional. If this argument is specified, the `sql_id` will be matched with the actively-running SQL statement in the session before terminating the SQL. If the session is executing a SQL statement other than the one specified in the `sql_id` argument, an error is raised.

Examples

Archiving Redo Logs Manually: Examples

The following statement manually archives the redo log file group containing the redo log entry with the SCN 9356083:

```
ALTER SYSTEM ARCHIVE LOG CHANGE 9356083;
```

The following statement manually archives the redo log file group containing a member named '`diskl:log6.log`' to an archived redo log file in the location '`diska:[arch$]`':

```
ALTER SYSTEM ARCHIVE LOG
    LOGFILE 'diskl:log6.log'
    TO 'diska:[arch$]';
```

Enabling Query Rewrite: Example

This statement enables query rewrite in all sessions for all materialized views for which query rewrite has not been explicitly disabled:

```
ALTER SYSTEM SET QUERY_REWRITE_ENABLED = TRUE;
```

Restricting Sessions: Example

You might want to restrict sessions if you are performing application maintenance and you want only application developers with `RESTRICTED SESSION` system privilege to log on. To restrict sessions, issue the following statement:

```
ALTER SYSTEM
   ENABLE RESTRICTED SESSION;
```

You can then terminate any existing sessions using the `KILL SESSION` clause of the `ALTER SYSTEM` statement.

After performing maintenance on your application, issue the following statement to allow any user with `CREATE SESSION` system privilege to log on:

```
ALTER SYSTEM
   DISABLE RESTRICTED SESSION;
```

Establishing a Wallet and Encryption Key: Examples

The following statements load information from the server wallet into memory and set the Transparent Data Encryption master key:

```
ALTER SYSTEM SET ENCRYPTION WALLET OPEN IDENTIFIED BY "password";
ALTER SYSTEM SET ENCRYPTION KEY IDENTIFIED BY "password";
```

These statements assume that you have initialized the security module and created a wallet with *`password`*.

Closing a Wallet: Examples

The following statement removes password-based wallet information from memory:

```
ALTER SYSTEM SET ENCRYPTION WALLET CLOSE IDENTIFIED BY "password";
```

The following statement removes password-based wallet information and auto-login information, if present, from memory:

```
ALTER SYSTEM SET ENCRYPTION WALLET CLOSE;
```

Clearing the Shared Pool: Example

You might want to clear the shared pool before beginning performance analysis. To clear the shared pool, issue the following statement:

```
ALTER SYSTEM FLUSH SHARED_POOL;
```

Forcing a Checkpoint: Example

The following statement forces a checkpoint:

```
ALTER SYSTEM CHECKPOINT;
```

Enabling Resource Limits: Example

This `ALTER SYSTEM` statement dynamically enables resource limits:

```
ALTER SYSTEM SET RESOURCE_LIMIT = TRUE;
```

Changing Shared Server Settings: Examples

The following statement changes the minimum number of shared server processes to 25:

```
ALTER SYSTEM SET SHARED_SERVERS = 25;
```

If there are currently fewer than 25 shared server processes, then Oracle Database creates more. If there are currently more than 25, then Oracle Database terminates some of them when they are finished processing their current calls if the load could be managed by the remaining 25.

The following statement dynamically changes the number of dispatcher processes for the TCP/IP protocol to 5 and the number of dispatcher processes for the ipc protocol to 10:

```
ALTER SYSTEM
   SET DISPATCHERS =
      '(INDEX=0)(PROTOCOL=TCP)(DISPATCHERS=5)',
      '(INDEX=1)(PROTOCOL=ipc)(DISPATCHERS=10)';
```

If there are currently fewer than 5 dispatcher processes for TCP, then Oracle Database creates new ones. If there are currently more than 5, then Oracle Database terminates some of them after the connected users disconnect.

If there are currently fewer than 10 dispatcher processes for ipc, then Oracle Database creates new ones. If there are currently more than 10, then Oracle Database terminates some of them after the connected users disconnect.

If there are currently existing dispatchers for another protocol, then the preceding statement does not affect the number of dispatchers for that protocol.

Changing Licensing Parameters: Examples

The following statement dynamically changes the limit on sessions for your instance to 64 and the warning threshold for sessions on your instance to 54:

```
ALTER SYSTEM
   SET LICENSE_MAX_SESSIONS = 64
   LICENSE_SESSIONS_WARNING = 54;
```

If the number of sessions reaches 54, then Oracle Database writes a warning message to the `ALERT` file for each subsequent session. Also, users with `RESTRICTED SESSION` system privilege receive warning messages when they begin subsequent sessions.

If the number of sessions reaches 64, then only users with `RESTRICTED SESSION` system privilege can begin new sessions until the number of sessions falls below 64 again.

The following statement dynamically disables the limit for sessions on your instance. After you issue this statement, Oracle Database no longer limits the number of sessions on your instance.

```
ALTER SYSTEM SET LICENSE_MAX_SESSIONS = 0;
```

The following statement dynamically changes the limit on the number of users in the database to 200. After you issue the preceding statement, Oracle Database prevents the number of users in the database from exceeding 200.

```
ALTER SYSTEM SET LICENSE_MAX_USERS = 200;
```

Forcing a Log Switch: Example

You might want to force a log switch to drop or rename the current redo log file group or one of its members, because you cannot drop or rename a file while Oracle Database is writing to it.

The forced log switch affects only the redo log thread of your instance. The following statement forces a log switch:

```
ALTER SYSTEM SWITCH LOGFILE;
```

Enabling Distributed Recovery: Example

The following statement enables distributed recovery:

```
ALTER SYSTEM ENABLE DISTRIBUTED RECOVERY;
```

You might want to disable distributed recovery for demonstration or testing purposes. You can disable distributed recovery in both single-process and multiprocess mode with the following statement:

```
ALTER SYSTEM DISABLE DISTRIBUTED RECOVERY;
```

When your demonstration or testing is complete, you can then enable distributed recovery again by issuing an `ALTER SYSTEM` statement with the `ENABLE DISTRIBUTED RECOVERY` clause.

Terminating a Session: Example

You might want to terminate the session of a user that is holding resources needed by other users. The user receives an error message indicating that the session has been terminated. That user can no longer make calls to the database without beginning a new session. Consider this data from the `V$SESSION` dynamic performance table, when the users `SYS` and `oe` both have open sessions:

```
SELECT sid, serial#, username
   FROM V$SESSION;

       SID    SERIAL# USERNAME
---------- ---------- ------------------------------
        29         85 SYS
        33          1
        35          8
        39         23 OE
        40          1
. . .
```

The following statement terminates the session of the user `scott` using the `SID` and `SERIAL#` values from `V$SESSION`:

```
ALTER SYSTEM KILL SESSION '39, 23';
```

Disconnecting a Session: Example

The following statement disconnects user `scott`'s session, using the `SID` and `SERIAL#` values from `V$SESSION`:

```
ALTER SYSTEM DISCONNECT SESSION '13, 8' POST_TRANSACTION;
```

ALTER TABLE

Purpose

Use the `ALTER TABLE` statement to alter the definition of a nonpartitioned table, a partitioned table, a table partition, or a table subpartition. For object tables or relational tables with object columns, use `ALTER TABLE` to convert the table to the latest definition of its referenced type after the type has been altered.

Note:

Oracle recommends that you use the `ALTER MATERIALIZED VIEW LOG` statement, rather than `ALTER TABLE`, whenever possible for operations on materialized view log tables.

See Also:

- CREATE TABLE for information on creating tables
- *Oracle Text Reference* for information on `ALTER TABLE` statements in conjunction with Oracle Text

Prerequisites

The table must be in your own schema, or you must have `ALTER` object privilege on the table, or you must have `ALTER ANY TABLE` system privilege.

Additional Prerequisites for Partitioning Operations

If you are not the owner of the table, then you need the `DROP ANY TABLE` privilege in order to use the *`d_table_partition`* or *`truncate_table_partition`* clause.

You must also have space quota in the tablespace in which space is to be acquired in order to use the *`add_table_partition`*, *`modify_table_partition`*, *`move_table_partition`*, and *`split_table_partition`* clauses.

When a partitioning operation cascades to reference-partitioned child tables, privileges are not required on the reference-partitioned child tables.

When using the *`exchange_partition_subpart`* clause, if the table data being exchanged contains an identity column and you are not the owner of both tables involved in the exchange, then you must have the `ALTER ANY SEQUENCE` system privilege.

You cannot partition a non-partitioned table that has an object type.

Additional Prerequisites for Constraints and Triggers

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 enable or disable triggers, the triggers must be in your schema or you must have the `ALTER ANY TRIGGER` system privilege.

See Also:

CREATE INDEX for information on the privileges needed to create indexes

Additional Prerequisites When Using Object Types

To use an object type in a column definition when modifying a table, either that object must belong to the same schema as the table being altered, or you must have either the `EXECUTE ANY TYPE` system privilege or the `EXECUTE` object privilege for the object type.

Additional Prerequisites for Flashback Data Archive Operations

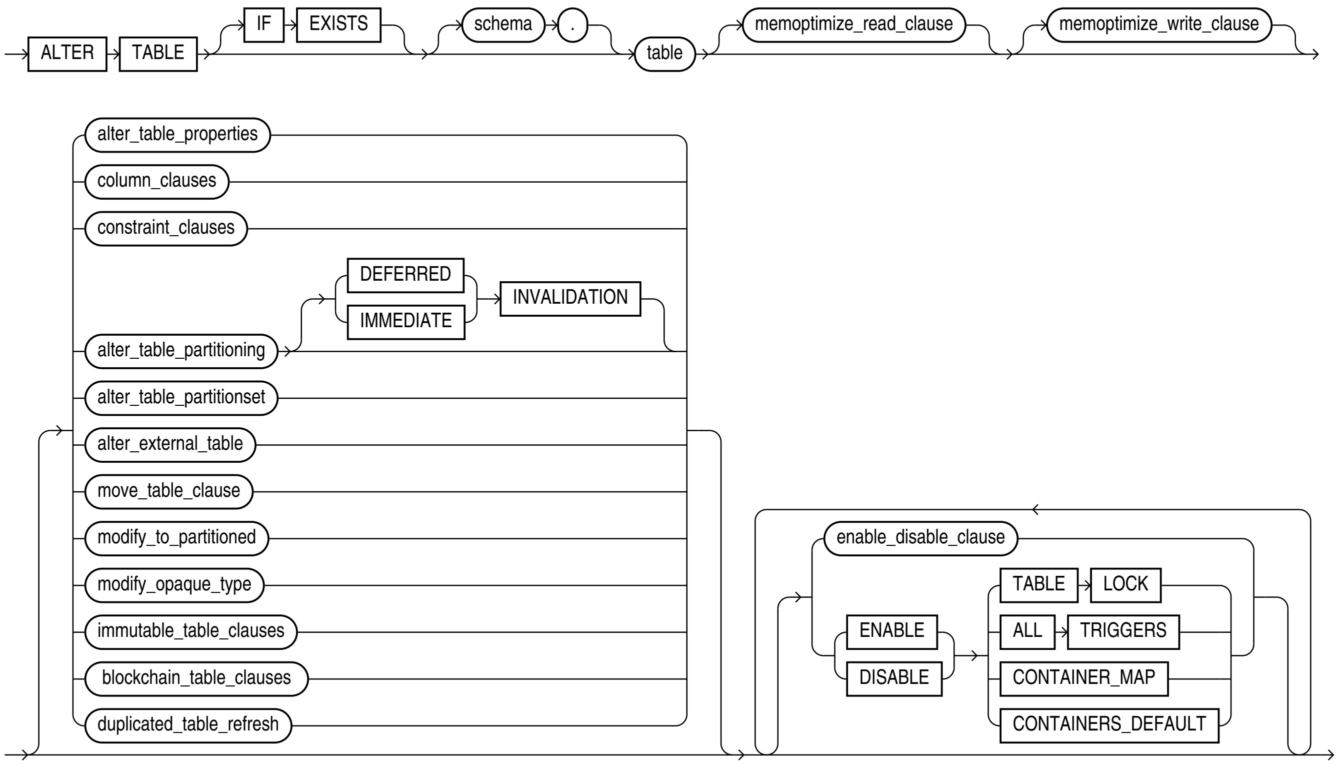
To use the *`flashback_archive_clause`* to *enable* historical tracking for the table, you must have the `FLASHBACK ARCHIVE` object privilege on the flashback data archive that will contain the historical data. To use the *`flashback_archive_clause`* to *disable* historical tracking for the table, you must have the `FLASHBACK ARCHIVE ADMINSTER` system privilege or you must be logged in as `SYSDBA`.

Additional Prerequisite for Referring to Editioned Objects

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

Syntax

alter_table::=



Note:

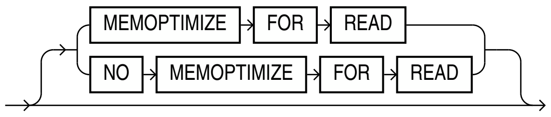
You must specify some clause after *`table`*. None of the clauses after *`table`* are required, but you must specify at least one of them.

Groups of ALTER TABLE syntax:

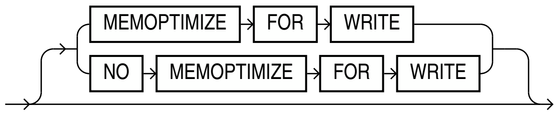
- ***alter_table_properties*::=**
- ***column_clauses*::=**
- ***constraint_clauses*::=**
- ***alter_table_partitioning*::=**
- ***alter_table_partitionset*::=**
- ***alter_external_table*::=**
- ***move_table_clause*::=**
- ***modify_to_partitioned*::=**
- ***modify_opaque_type*::=**
- **immutable_table_clauses**
- **blockchain_table_clauses**
- ***duplicated_table_refresh*::=**
- ***enable_disable_clause*::=**

After each clause you will find links to its component subclauses.

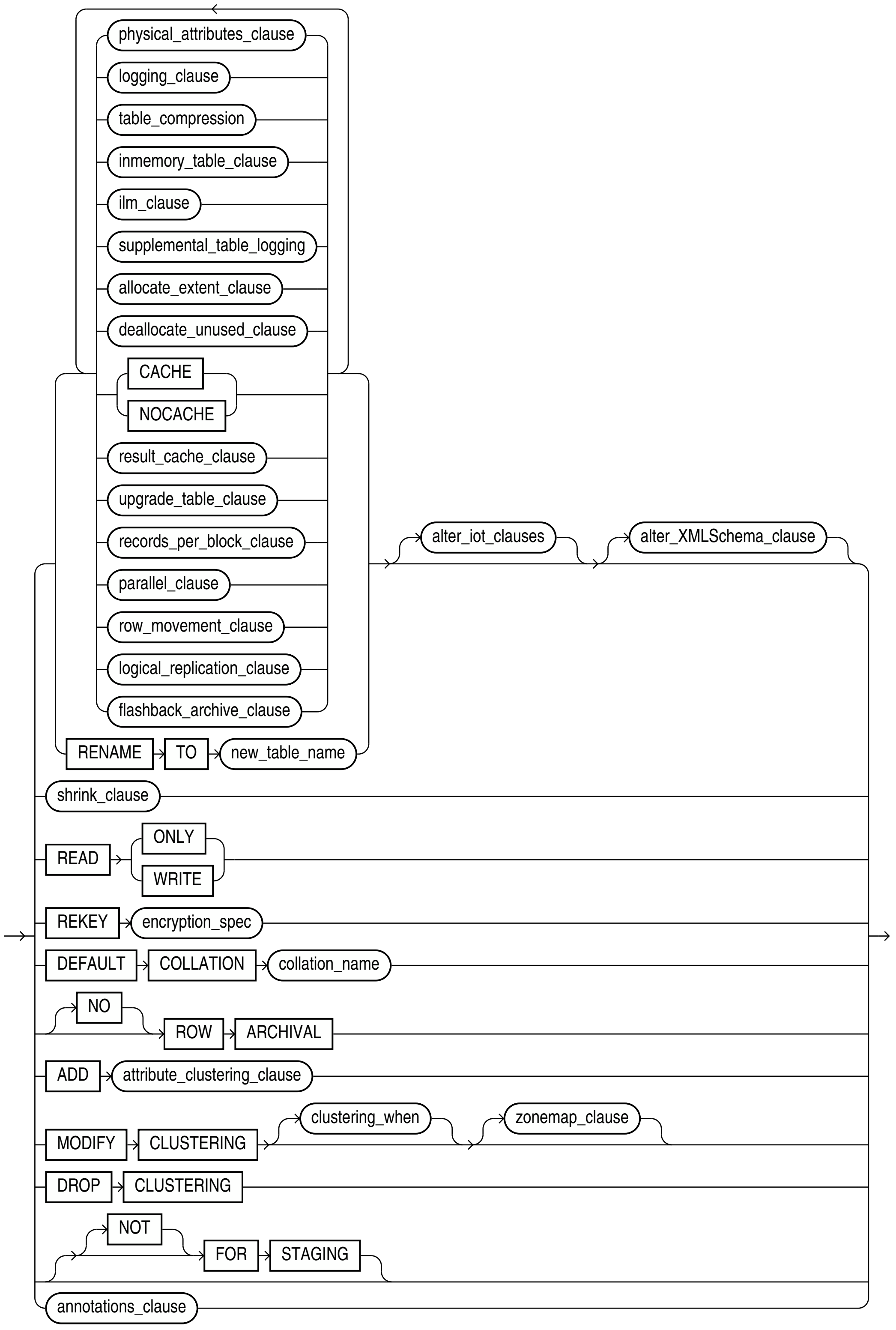
***memoptimize_read_clause*::=**



memoptimize_write_clause



***alter_table_properties*::=**

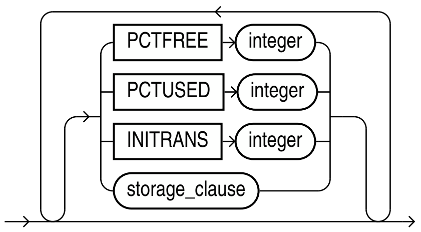


Note:

If you specify the `MODIFY CLUSTERING` clause, then you must specify at least one of the clauses *`clustering_when`* or *`zonemap_clause`*.

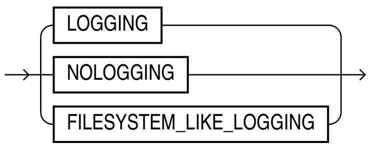
(*physical_attributes_clause*::=, *logging_clause*::=, *table_compression*::=, *inmemory_table_clause*::=, *ilm_clause*::=, *supplemental_table_logging*::=, *allocate_extent_clause*::=, *deallocate_unused_clause*::= , *upgrade_table_clause*::=, *records_per_block_clause*::=, *parallel_clause*::=, *row_movement_clause*::=, *logical_replication_clause*::=, *flashback_archive_clause*::=, *shrink_clause*::=, *attribute_clustering_clause*::=, *clustering_when*::=, *zonemap_clause*::=, *alter_iot_clauses*::=, *alter_XMLSchema_clause*::=, *annotations_clause*::=)

***physical_attributes_clause*::=**

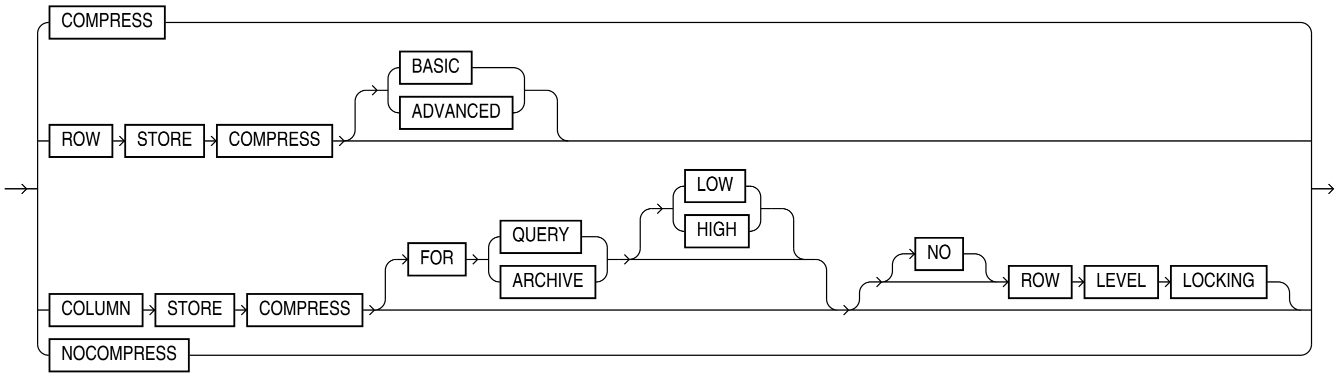


(*storage_clause*::=)

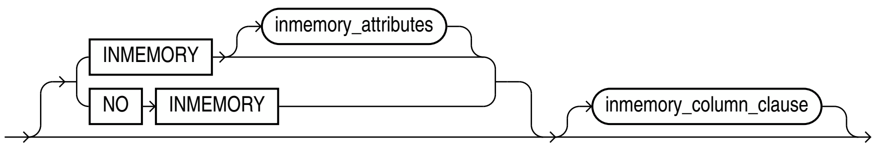
***logging_clause*::=**



***table_compression*::=**

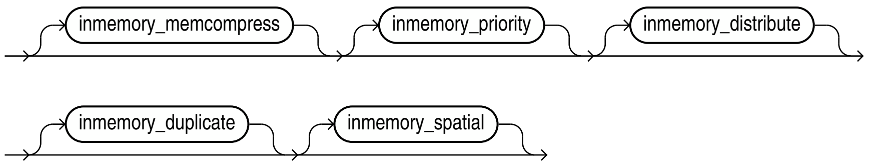


***inmemory_table_clause*::=**



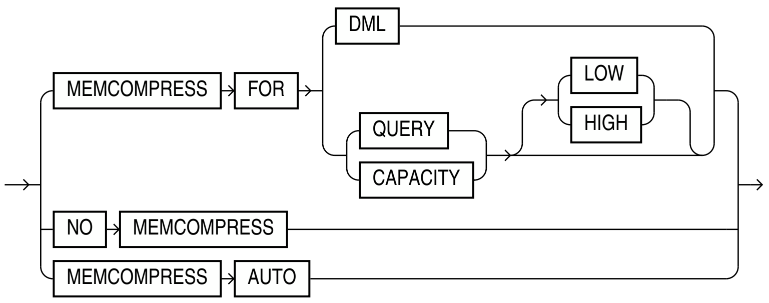
(*inmemory_attributes::=*, *inmemory_column_clause::=*)

***inmemory_attributes*::=**

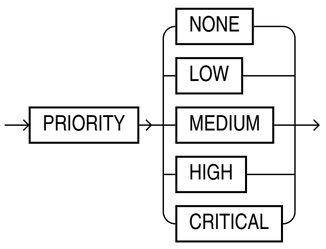


(*inmemory_memcompress::=*, *inmemory_priority::=*, *inmemory_distribute::=*, *inmemory_duplicate::=*)

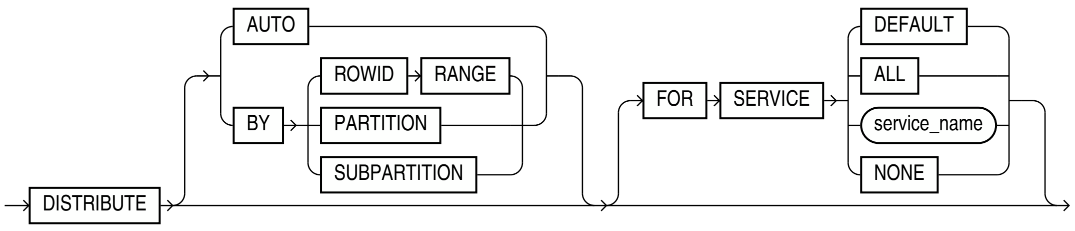
***inmemory_memcompress*::=**



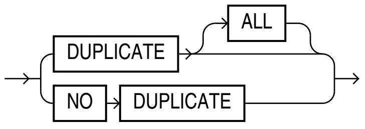
***inmemory_priority*::=**



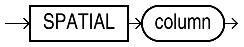
***inmemory_distribute*::=**



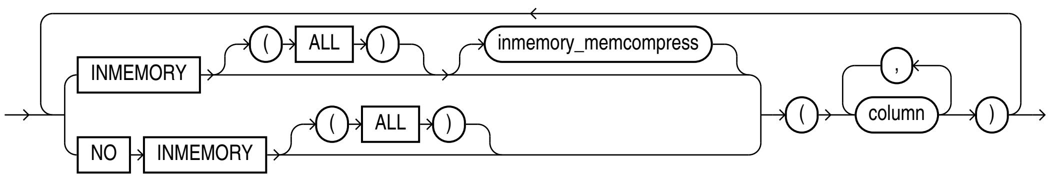
***inmemory_duplicate*::=**



***inmemory_spatial*::=**

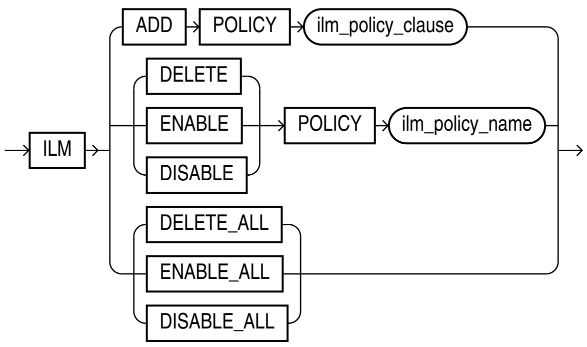


***inmemory_column_clause*::=**

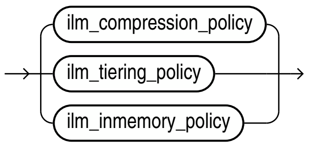


(*inmemory_memcompress*::=)

***ilm_clause*::=**

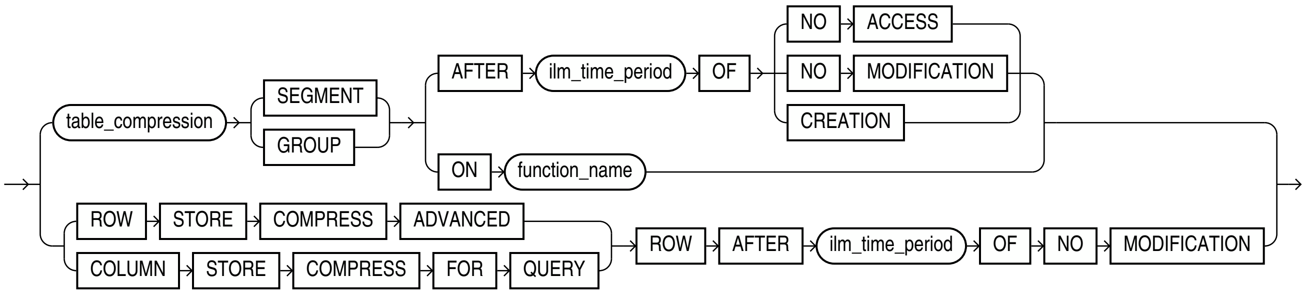


***ilm_policy_clause*::=**



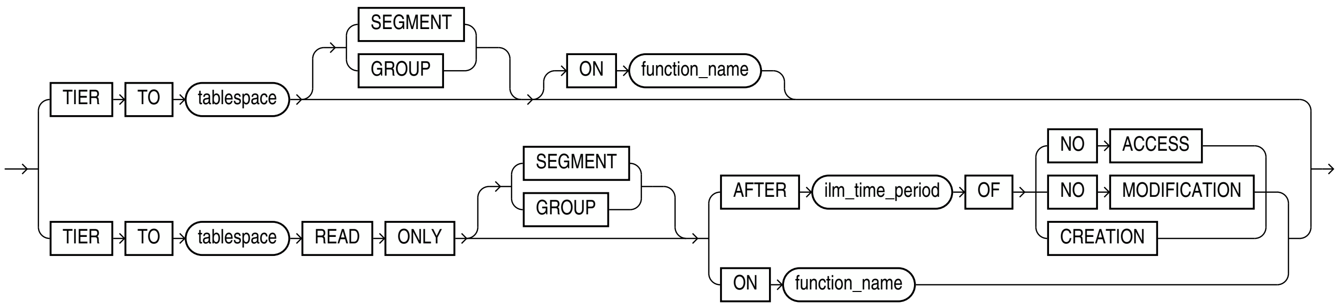
(*ilm_compression_policy::=*, *ilm_tiering_policy::=*, *ilm_inmemory_policy::=*)

***ilm_compression_policy*::=**



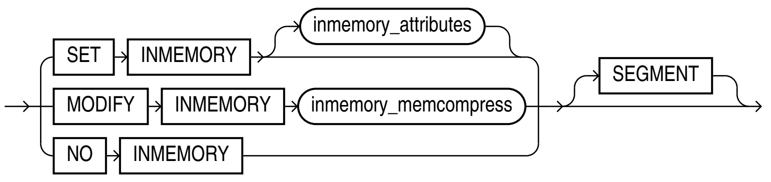
(*table_compression::=*, *ilm_time_period::=*)

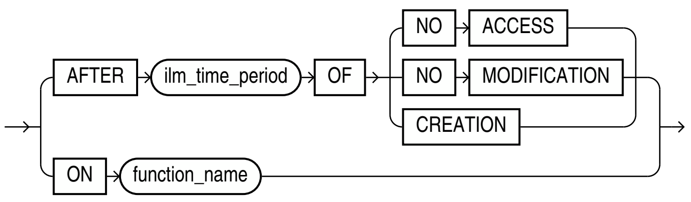
***ilm_tiering_policy*::=**



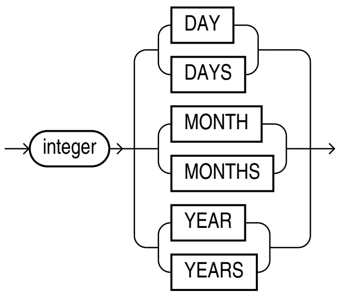
(*ilm_time_period::=*)

***ilm_inmemory_policy*::=**

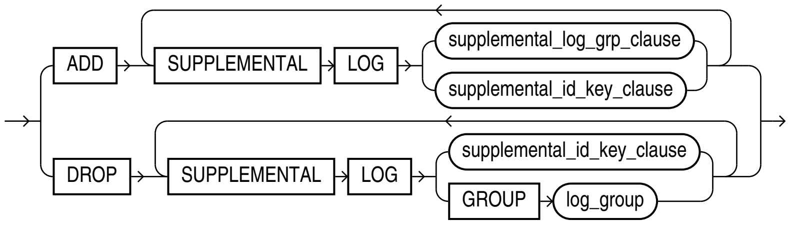




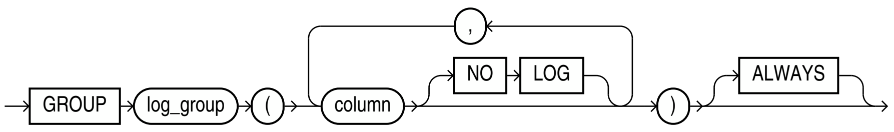
***ilm_time_period*::=**



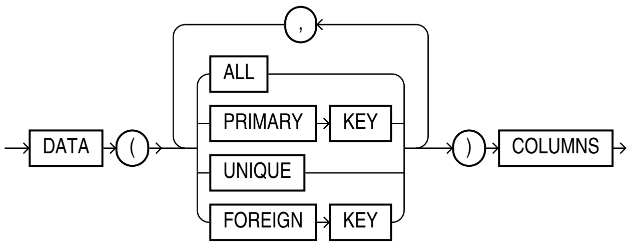
***supplemental_table_logging*::=**


ADD
SUPPLEMENTAL
LOG
supplemental_log_grp_clause
supplemental_id_key_clause
DROP
SUPPLEMENTAL
LOG
supplemental_id_key_clause
GROUP
log_group

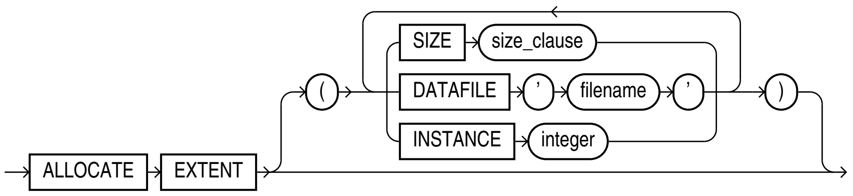
***supplemental_log_grp_clause*::=**



***supplemental_id_key_clause*::=**

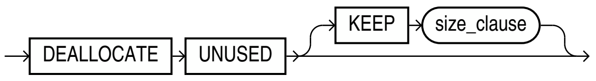


***allocate_extent_clause*::=**



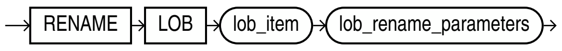
(*size_clause*::=)

***deallocate_unused_clause*::=**

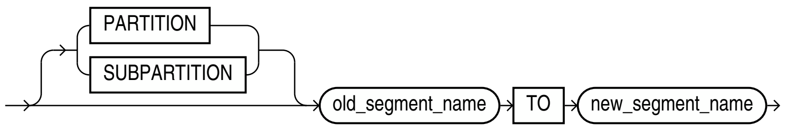


(*size_clause*::=)

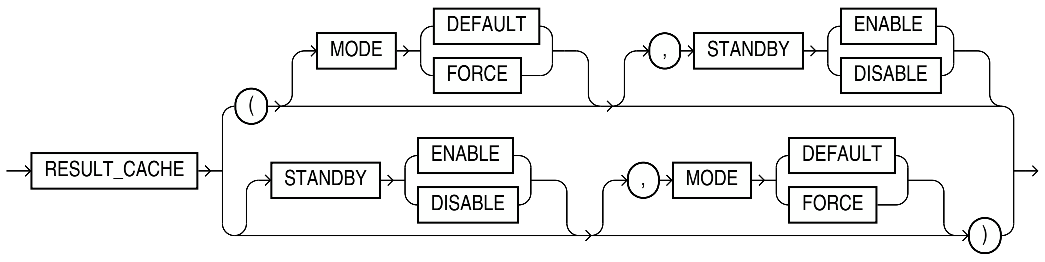
***rename_lob_storage_clause*::=**



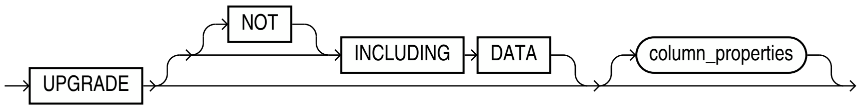
***rename_lob_parameters*::=**



***result_cache_clause*::=**

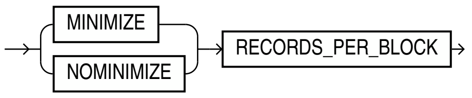


***upgrade_table_clause*::=**

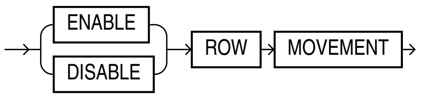


(*column_properties::=*)

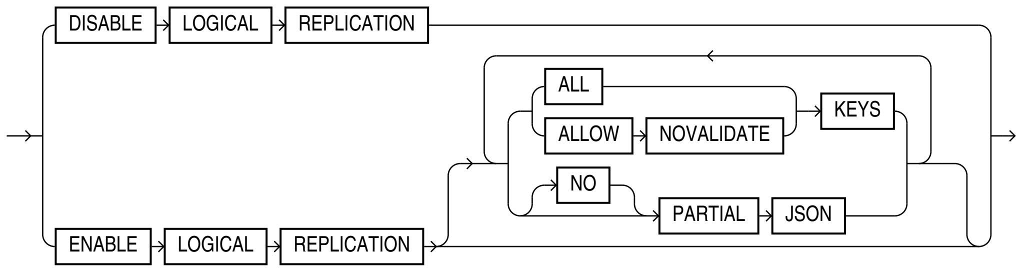
***records_per_block_clause*::=**



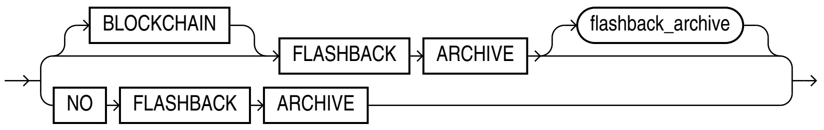
***row_movement_clause*::=**



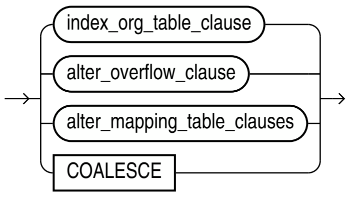
***logical_replication_clause*::=**



***flashback_archive_clause*::=**

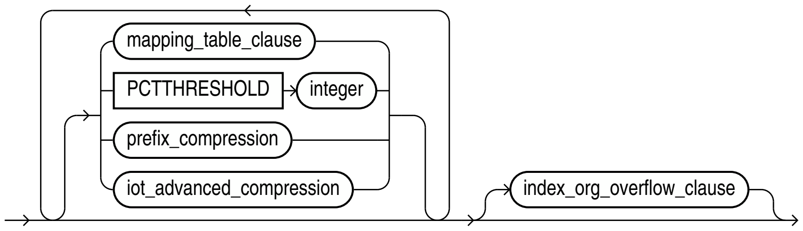


***alter_iot_clauses*::=**



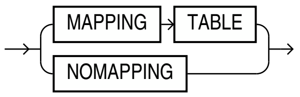
(*alter_overflow_clause*::=, *alter_mapping_table_clauses*::=)

***index_org_table_clause*::=**

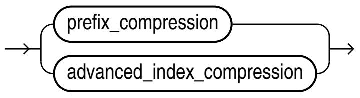


(*mapping_table_clauses*::=, *prefix_compression*::=, *index_org_overflow_clause*::=)

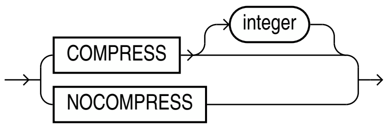
***mapping_table_clauses*::=**



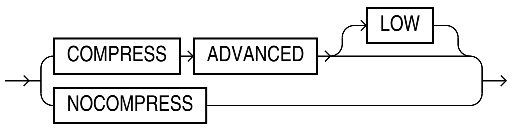
***index_compression*::=**



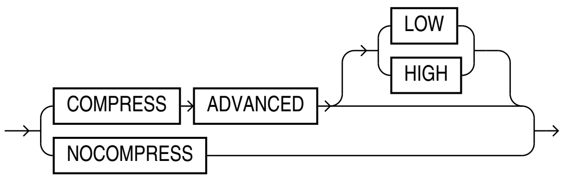
***prefix_compression*::=**



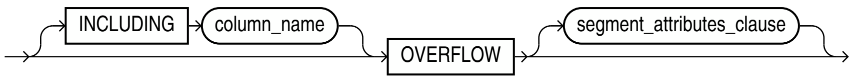
iot_advanced_compression::=



advanced_index_compression::=

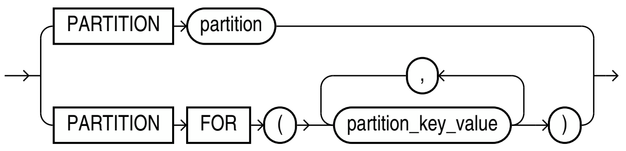


index_org_overflow_clause::=

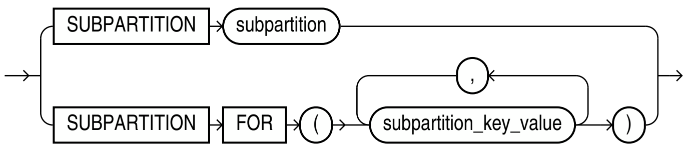


(*segment_attributes_clause::=*)

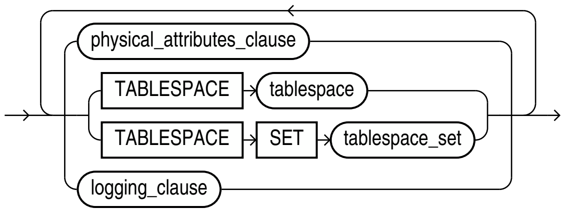
partition_extended_name::=



subpartition_extended_name::=

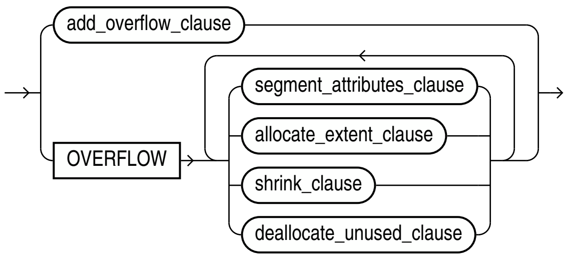


segment_attributes_clause::=



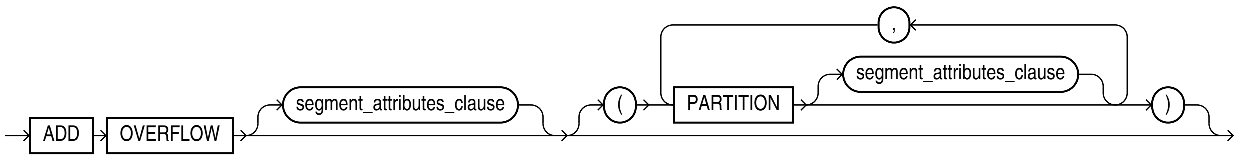
(*physical_attributes_clause*::=, `TABLESPACE SET`: not supported with `ALTER TABLE`, *logging_clause*::=)

***alter_overflow_clause*::=**



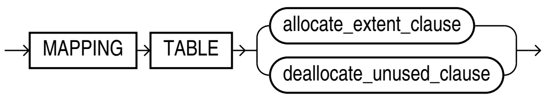
(*segment_attributes_clause*::=, *allocate_extent_clause*::=, *shrink_clause*::=, *deallocate_unused_clause*::=)

***add_overflow_clause*::=**



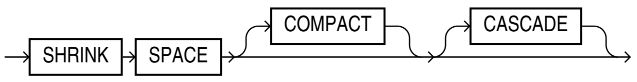
(*segment_attributes_clause*::=)

***alter_mapping_table_clauses*::=**

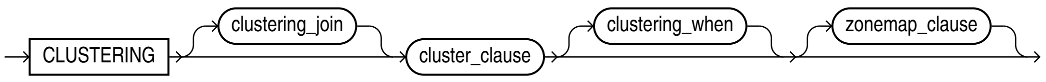


(*allocate_extent_clause*::=, *deallocate_unused_clause*::=)

***shrink_clause*::=**

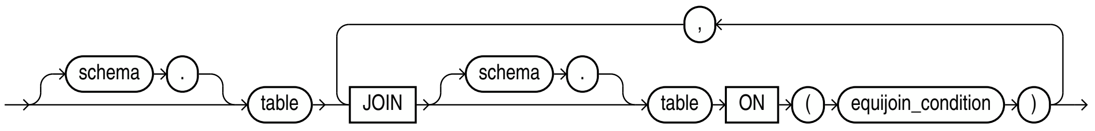


***attribute_clustering_clause*::=**

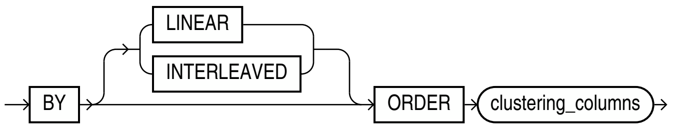


(*clustering_join*::=, *cluster_clause*::=, *clustering_when*::=, *zonemap_clause*::=)

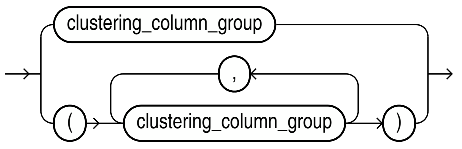
***clustering_join*::=**



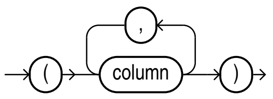
***cluster_clause*::=**



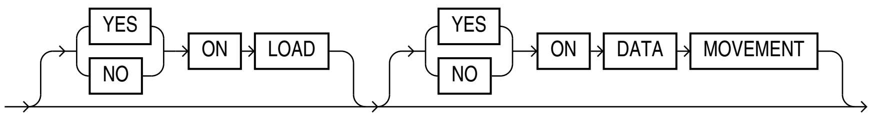
***clustering_columns*::=**



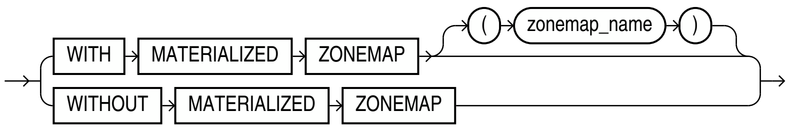
***clustering_column_group*::=**



***clustering_when*::=**



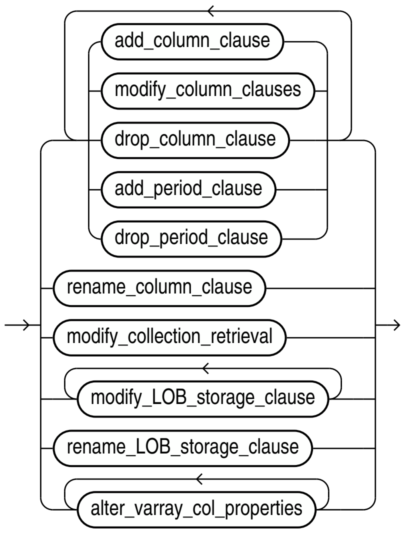
***zonemap_clause*::=**



***annotations_clause*::=**

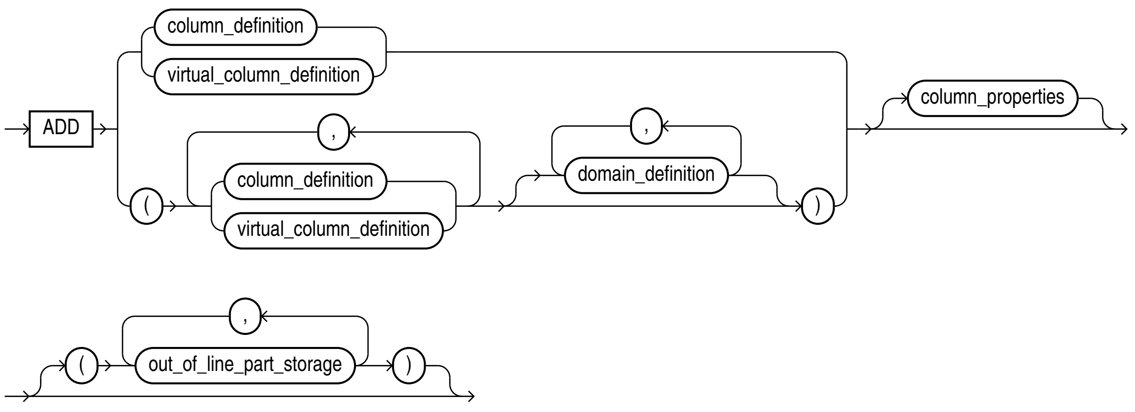
For the full syntax and semantics of the `annotations_clause` see *annotations_clause*.

***column_clauses*::=**



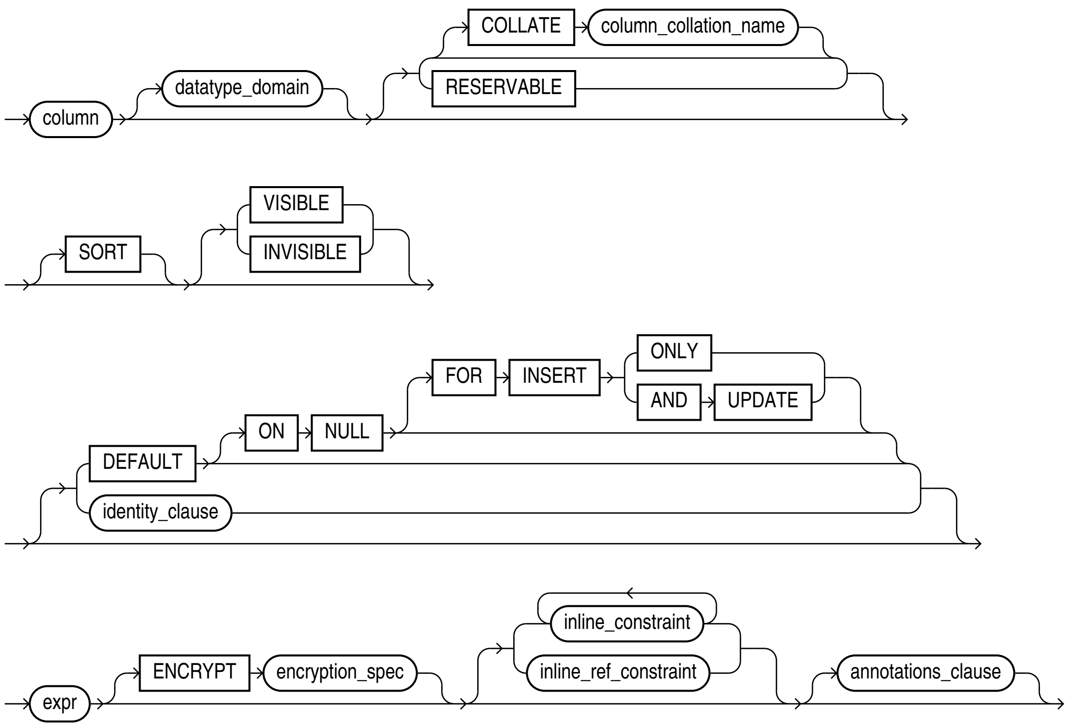
(*add_column_clause*::=, *modify_column_clauses*::=, *drop_column_clause*::=, *add_period_clause*::=, *drop_period_clause*::=, *rename_column_clause*::=, *modify_collection_retrieval*::=, *modify_LOB_storage_clause*::=, *alter_varray_col_properties*::=)

***add_column_clause*::=**



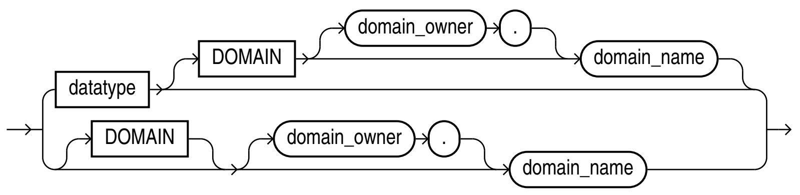
(*column_definition*::=, *virtual_column_definition*::=, *column_properties*::=, *out_of_line_part_storage*::=)

***column_definition*::=**



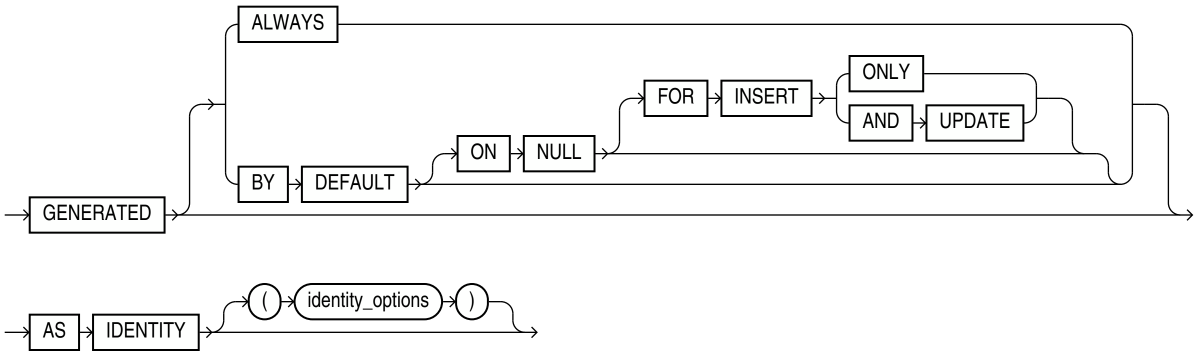
(*identity_clause*::=, *encryption_spec*::=, `inline_constraint` and `inline_ref_constraint`: *constraint*::=)

***datatype_domain*::=**

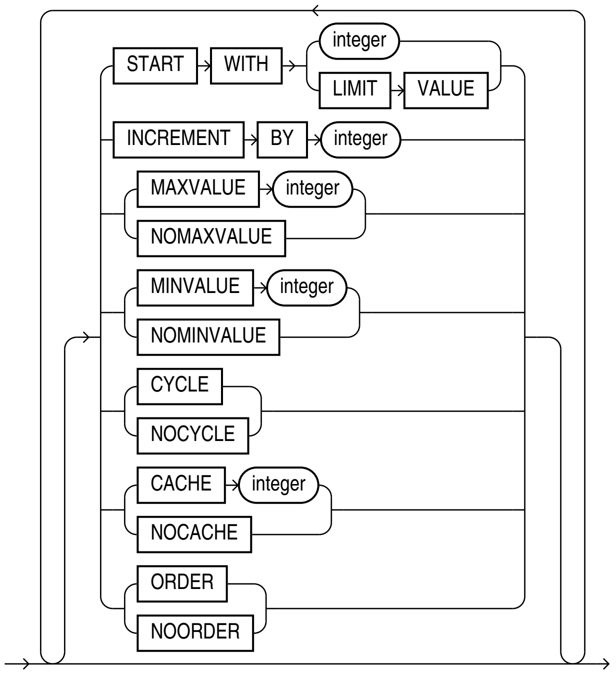


(*datatype*::=)

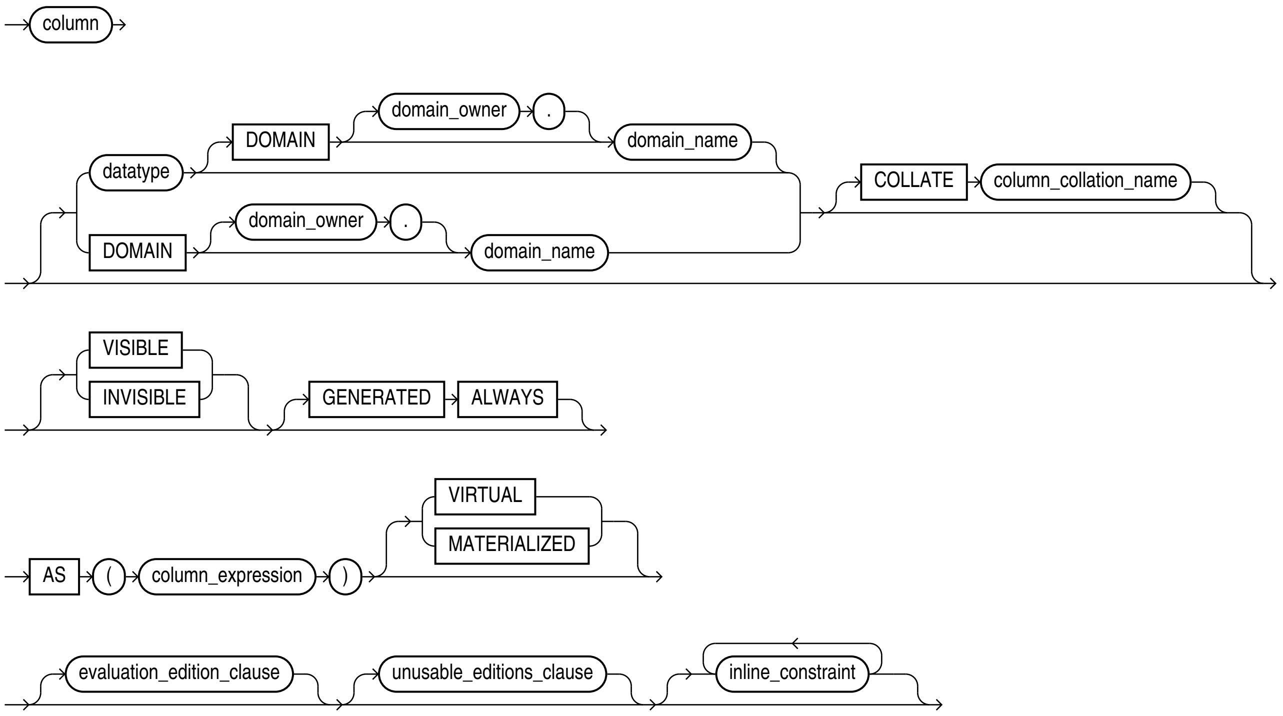
***identity_clause*::=**



***identity_options*::=**

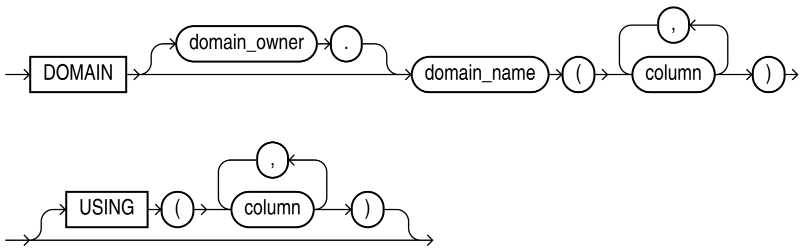


***virtual_column_definition*::=**

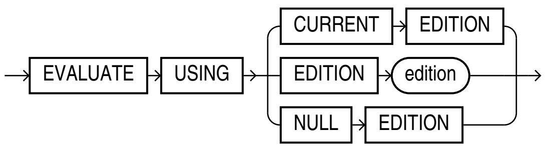


(*datatype*::=,*evaluation_edition_clause*::=, *unusable_editions_clause*::=, *constraint*::=)

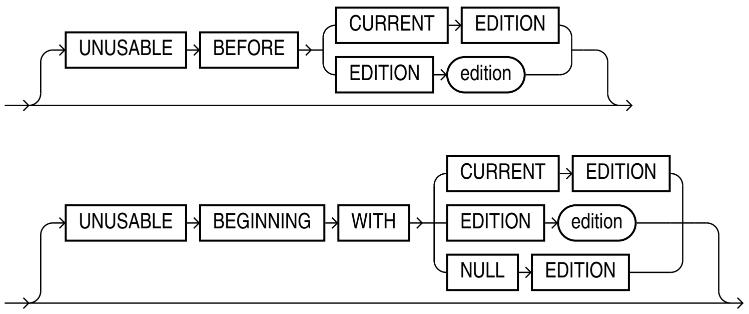
***domain_definition*::=**



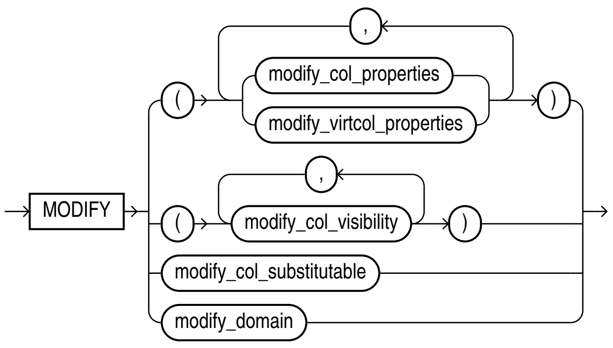
***evaluation_edition_clause*::=**



***unusable_editions_clause*::=**

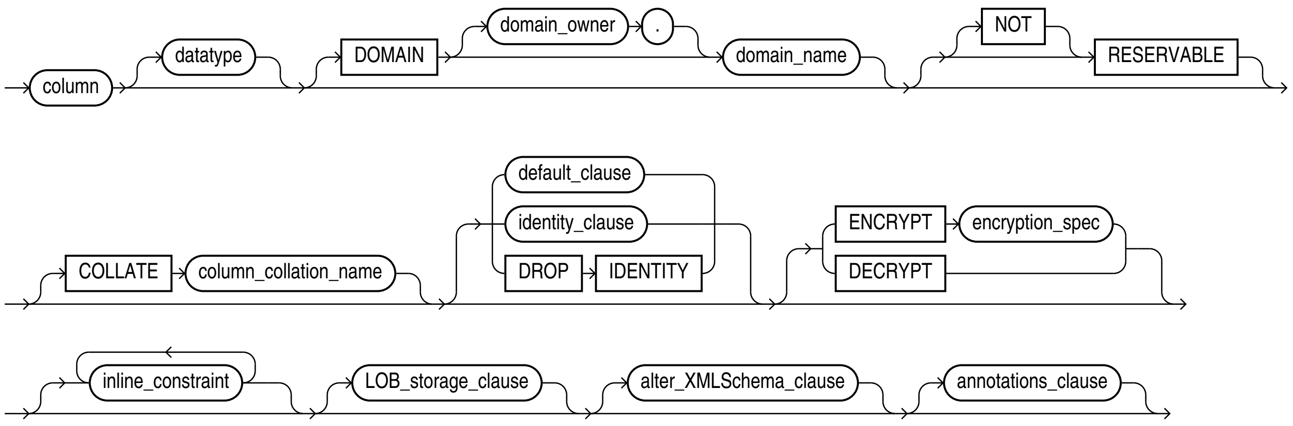


***modify_column_clauses*::=**



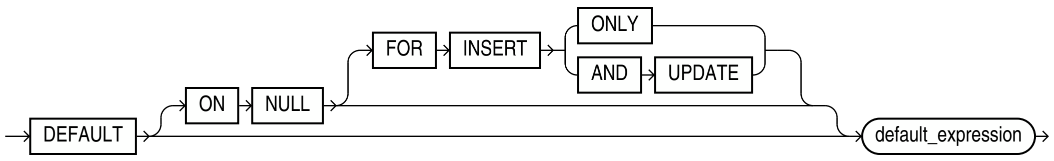
(*modify_col_properties*::=, *modify_virtcol_properties*::=, *modify_col_visibility*::=, *modify_col_substitutable*::=,*modify_domain*::=)

***modify_col_properties*::=**

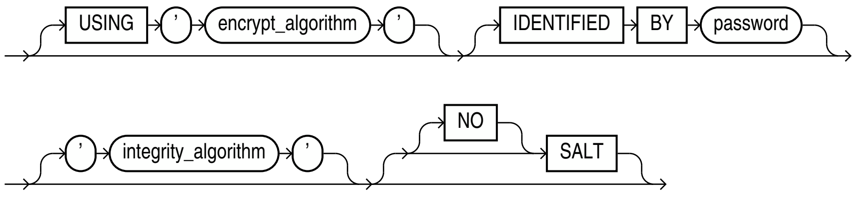


(*datatype*::=,*default_clause*::=, *identity_clause*::=, *encryption_spec*::=, *constraint*::=, *LOB_storage_clause*::=, *alter_XMLSchema_clause*::=, *annotations_clause*)

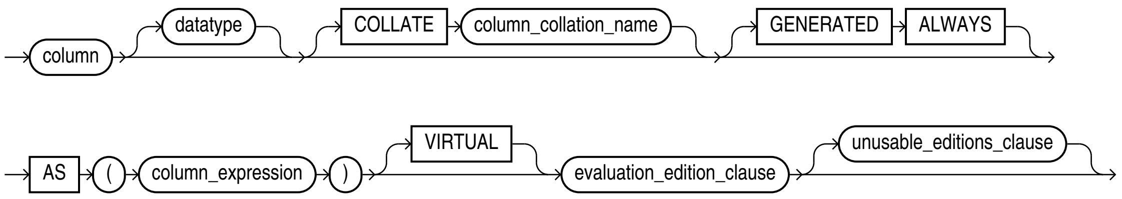
***default_clause*::=**



***encryption_spec*::=**

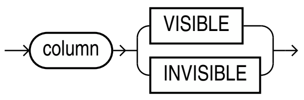


***modify_virtcol_properties*::=**

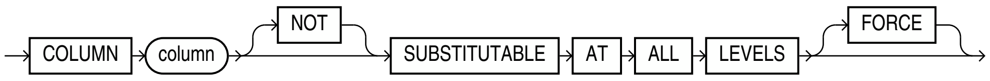


(*datatype*::=,*evaluation_edition_clause*::=, *unusable_editions_clause*::=)

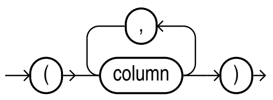
***modify_col_visibility*::=**

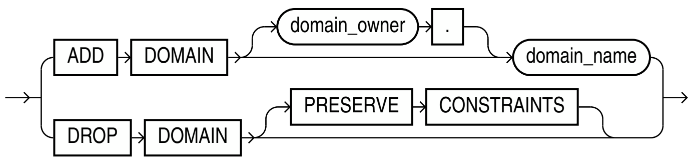


***modify_col_substitutable*::=**

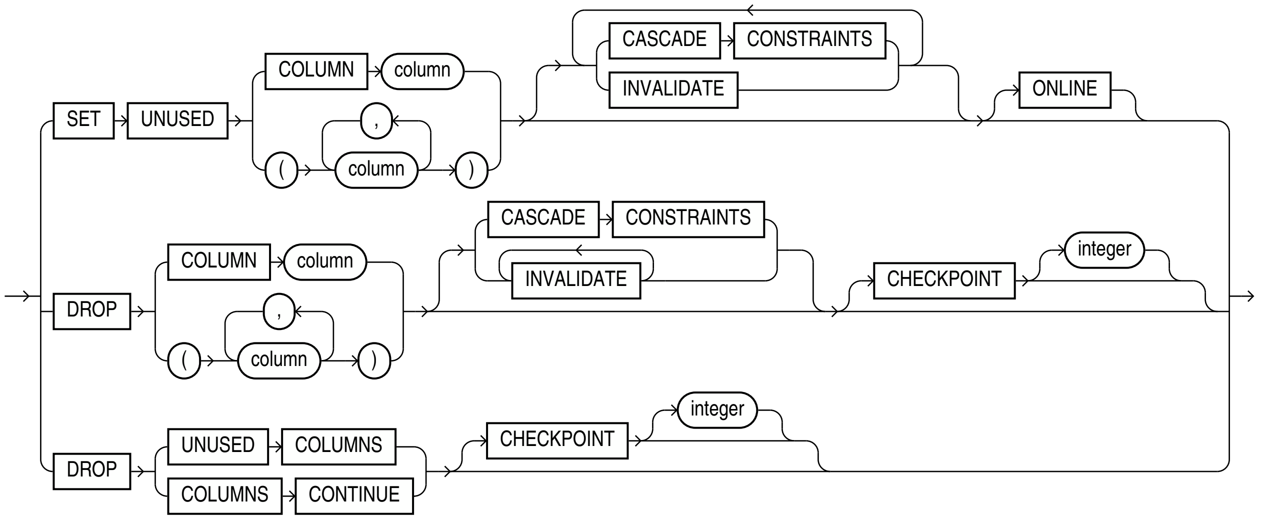


***modify_domain*::=**

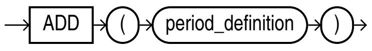




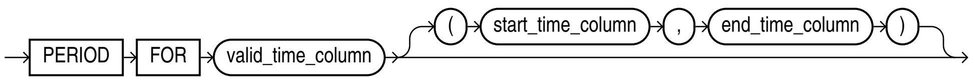
***drop_column_clause*::=**



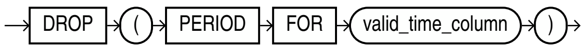
***add_period_clause*::=**



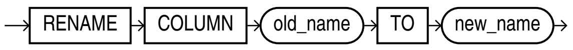
***period_definition*::=**



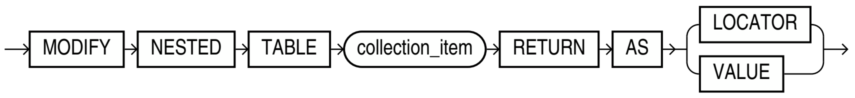
***drop_period_clause*::=**



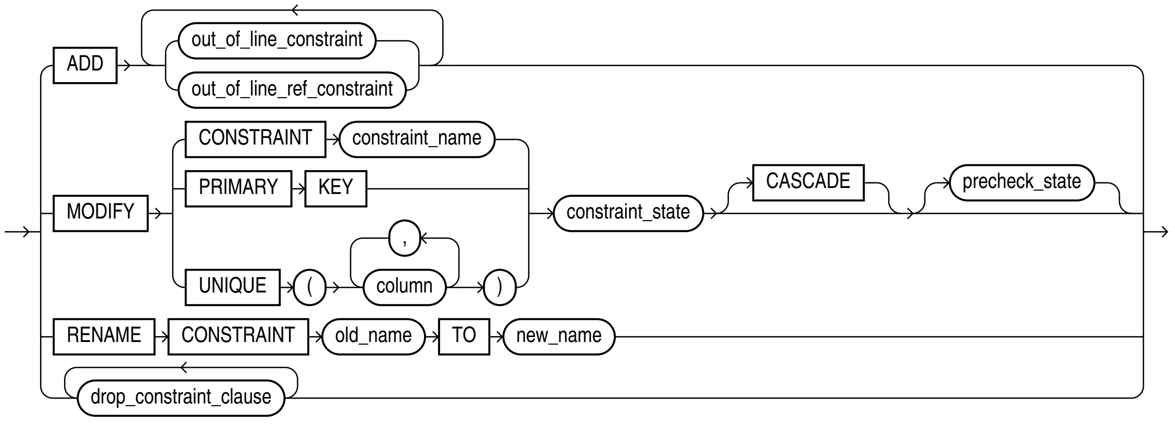
***rename_column_clause*::=**



***modify_collection_retrieval*::=**

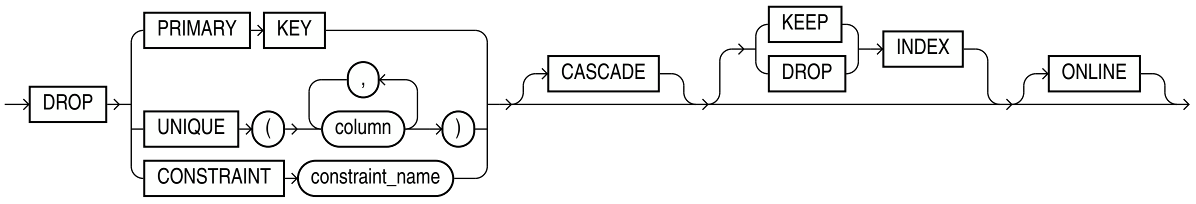


***constraint_clauses*::=**

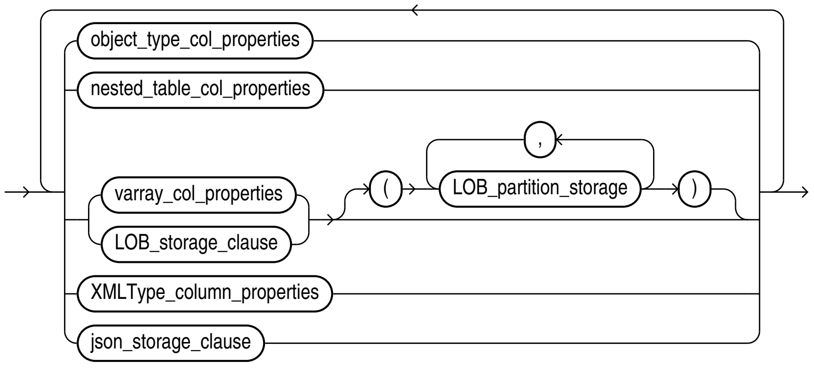


(*out_of_line_constraint*::=, *out_of_line_ref_constraint*::=, *constraint_state*::=)

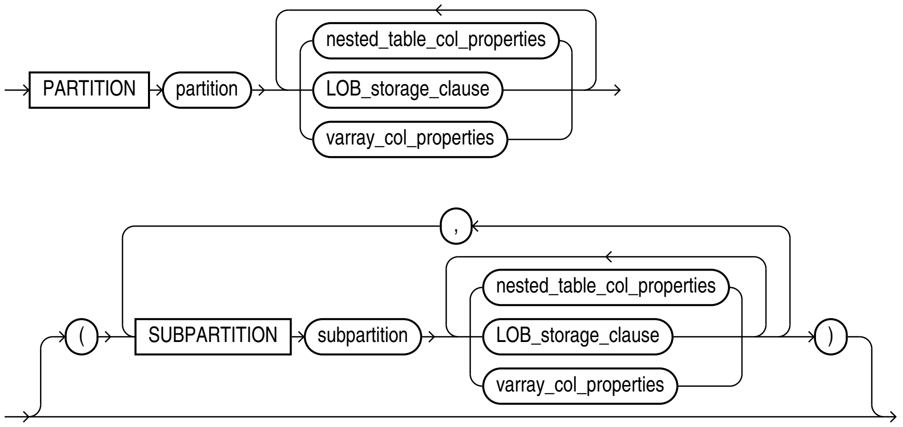
***drop_constraint_clause*::=**



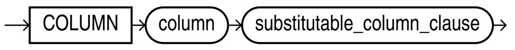
***column_properties*::=**



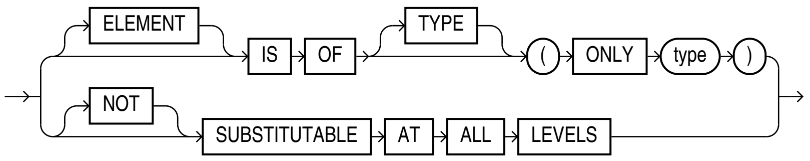
***out_of_line_part_storage*::=**



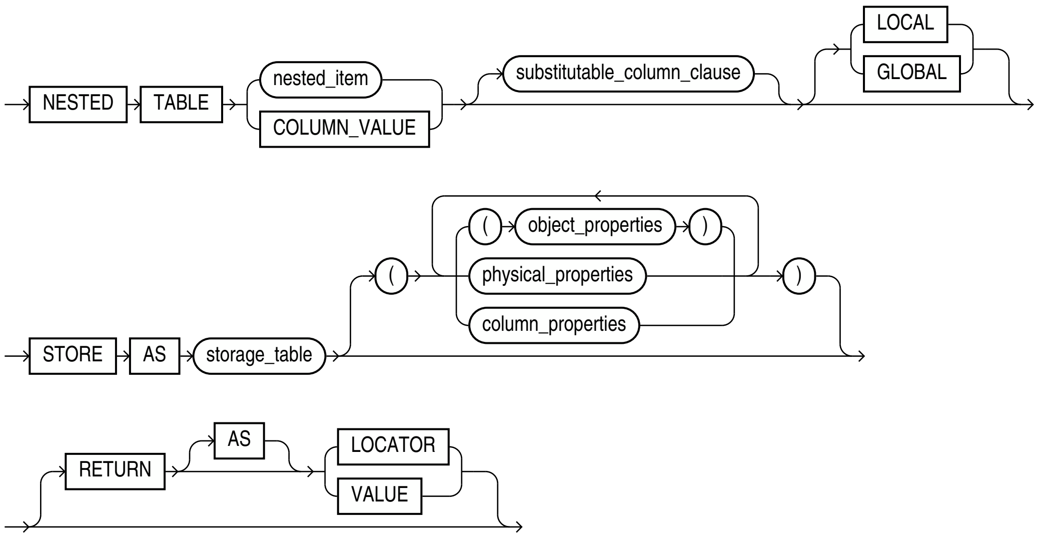
***object_type_col_properties*::=**



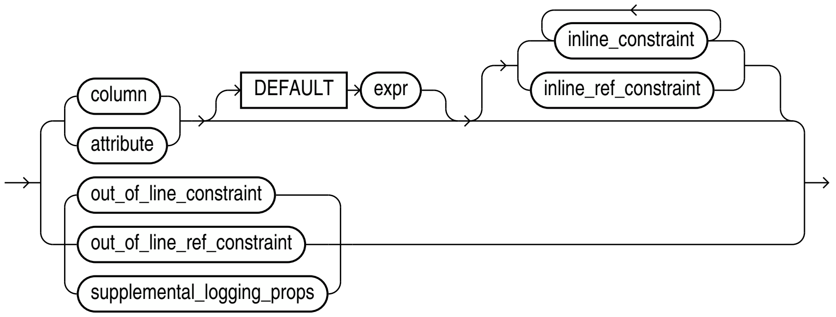
***substitutable_column_clause*::=**



***nested_table_col_properties*::=**

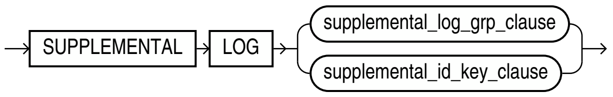


***object_properties*::=**



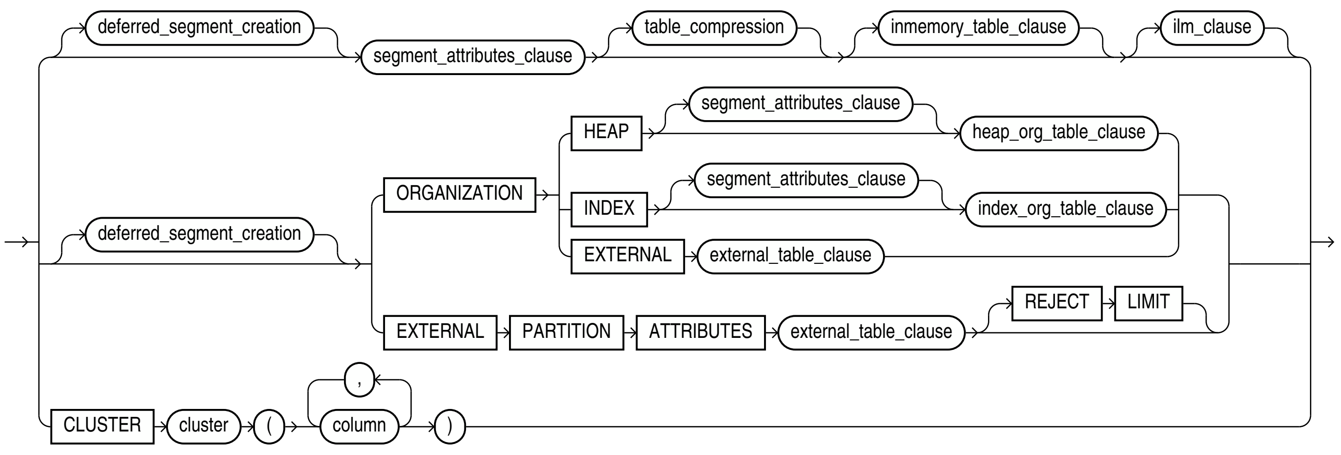
For constraint clauses see *constraint::=*

***supplemental_logging_props*::=**



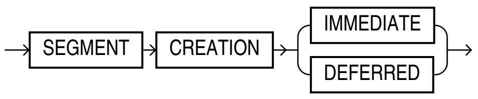
(*supplemental_log_grp_clause*::=, *supplemental_id_key_clause*::=)

***physical_properties*::=**

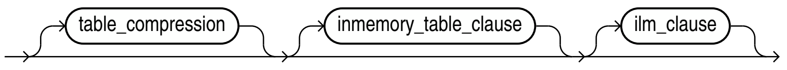


(*deferred_segment_creation*::= , *segment_attributes_clause*::=, *table_compression*::=, *inmemory_table_clause*::=—part of `CREATE TABLE` syntax, *ilm_clause*::=, *heap_org_table_clause*::=, *index_org_table_clause*::=, *external_table_clause*::=—part of `CREATE TABLE` syntax)

***deferred_segment_creation*::=**

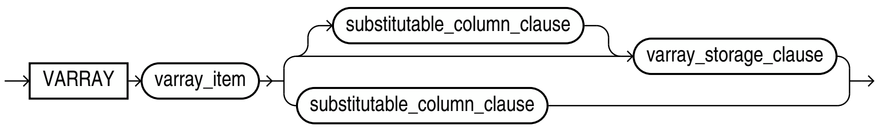


***heap_org_table_clause*::=**



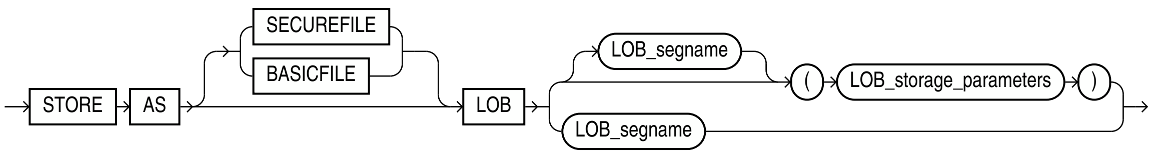
(*table_compression*::=, *inmemory_table_clause*::=—part of `CREATE TABLE` syntax, *ilm_clause*::=)

***varray_col_properties*::=**



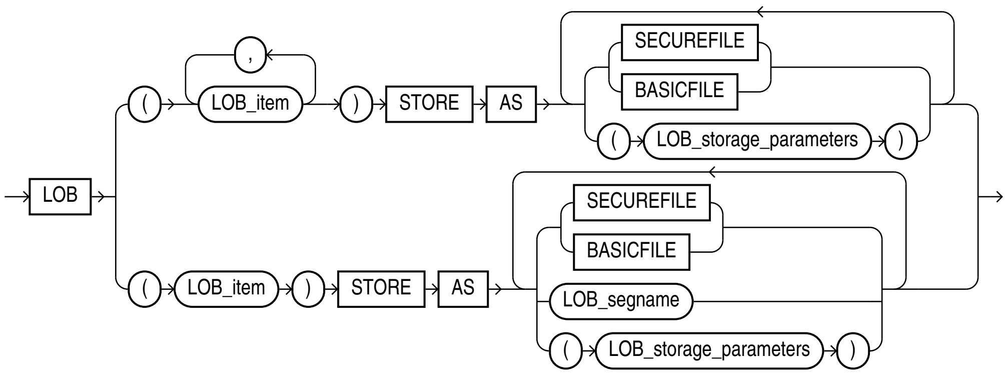
(*substitutable_column_clause*::=, *varray_storage_clause*::=)

***varray_storage_clause*::=**



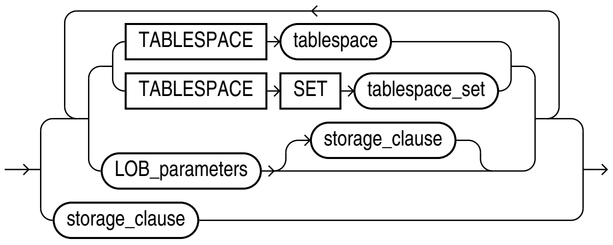
(*LOB_parameters*::=)

***LOB_storage_clause*::=**



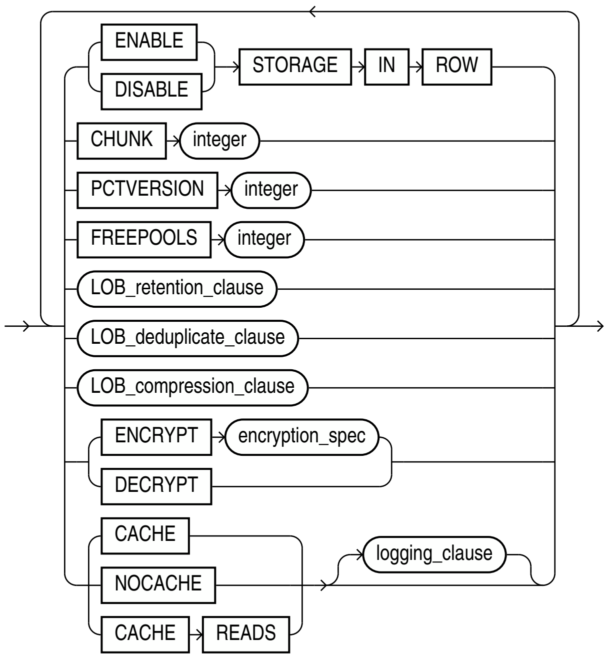
(*LOB_storage_parameters*::=)

***LOB_storage_parameters*::=**



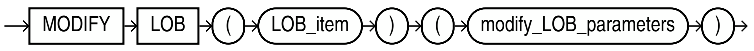
(`TABLESPACE SET`: not supported with `ALTER TABLE`, *LOB_parameters*::=, *storage_clause*::=)

***LOB_parameters*::=**

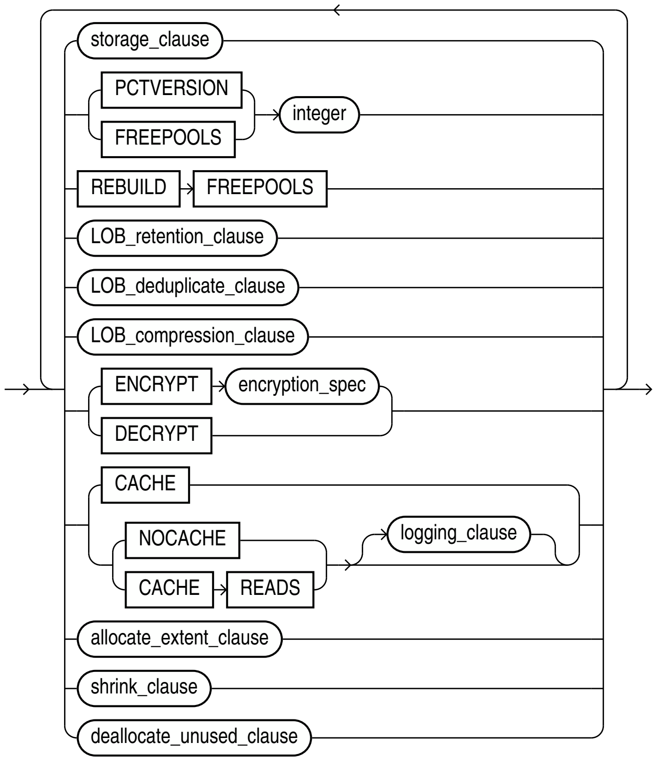


(*LOB_retention_clause*::=, *LOB_deduplicate_clause*::=, *LOB_compression_clause*::=, *encryption_spec*::=, *logging_clause*::=)

***modify_LOB_storage_clause*::=**

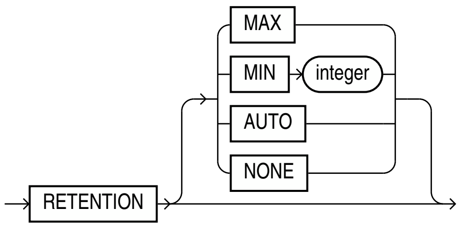


***modify_LOB_parameters*::=**

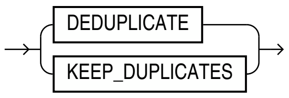


(*storage_clause*::=, *LOB_retention_clause*::=, *LOB_compression_clause*::=, *encryption_spec*::=, *logging_clause*::=, *allocate_extent_clause*::=, *shrink_clause*::=, *deallocate_unused_clause*::=)

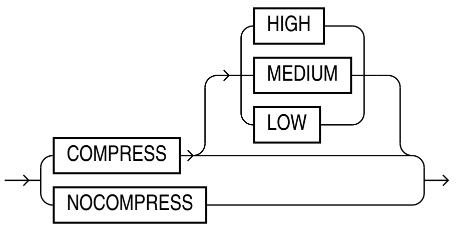
***LOB_retention_clause*::=**



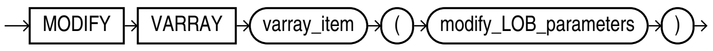
***LOB_deduplicate_clause*::=**



***LOB_compression_clause*::=**

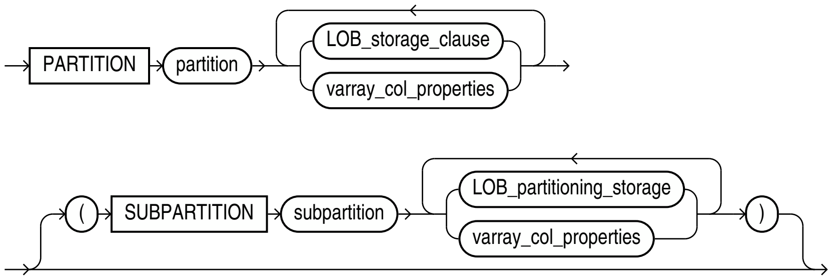


***alter_varray_col_properties*::=**



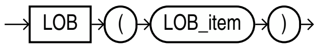
(*modify_LOB_parameters*::=)

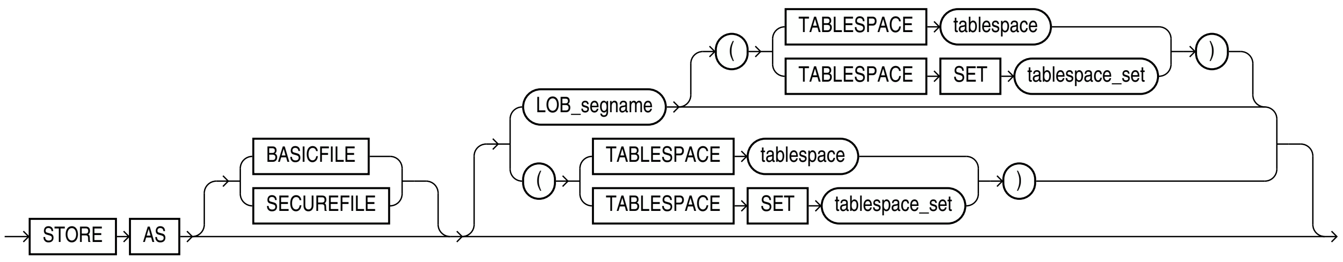
***LOB_partition_storage*::=**



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

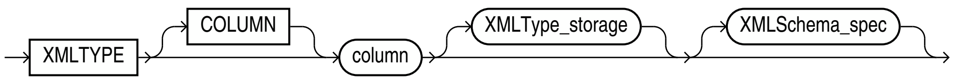
***LOB_partitioning_storage*::=**



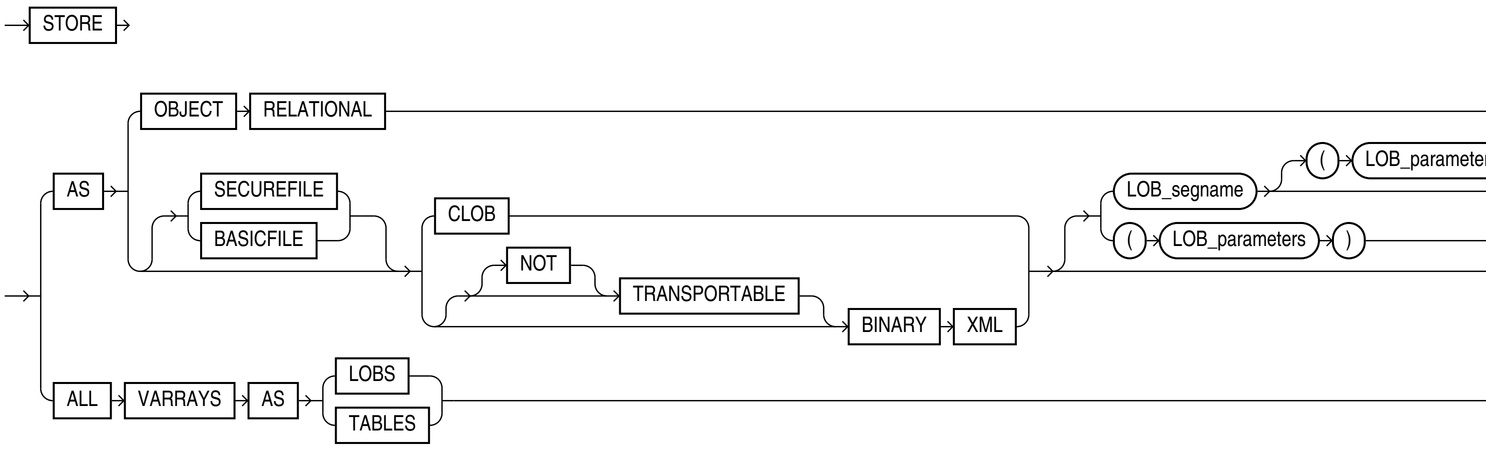


(`TABLESPACE SET`: not supported with `ALTER TABLE`)

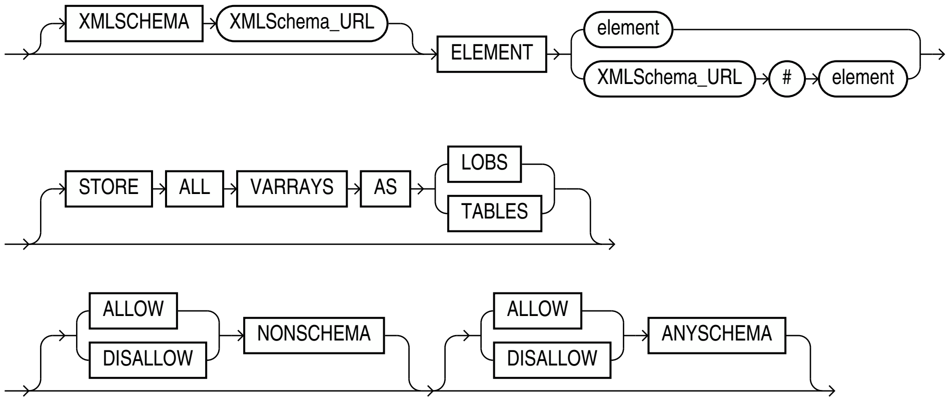
***XMLType_column_properties*::=**



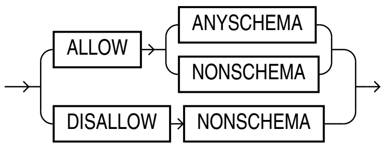
***XMLType_storage*::=**



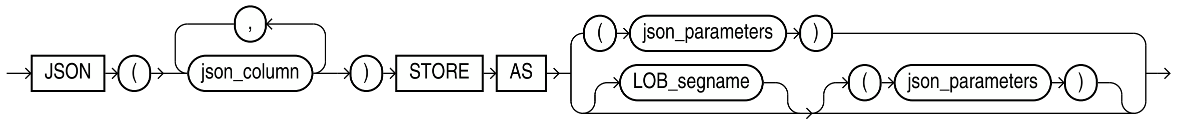
***XMLSchema_spec*::=**



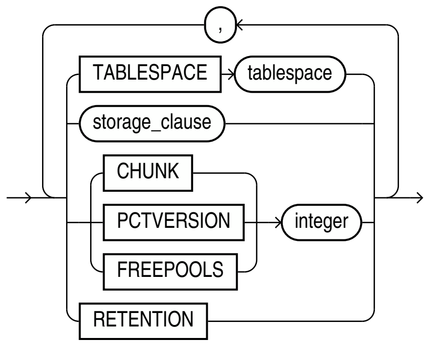
***alter_XMLSchema_clause*::=**



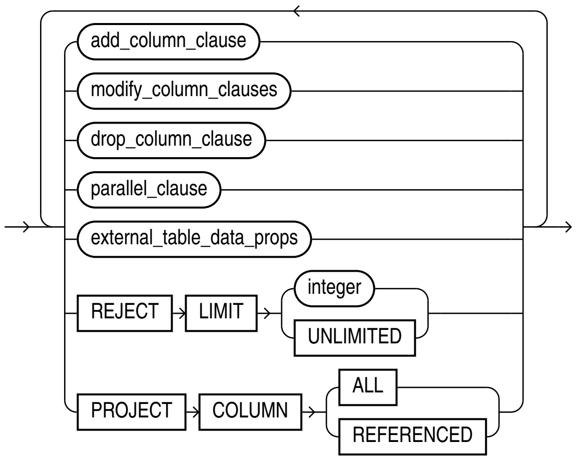
***JSON_storage_clause*::=**



***JSON_parameters* ::=**

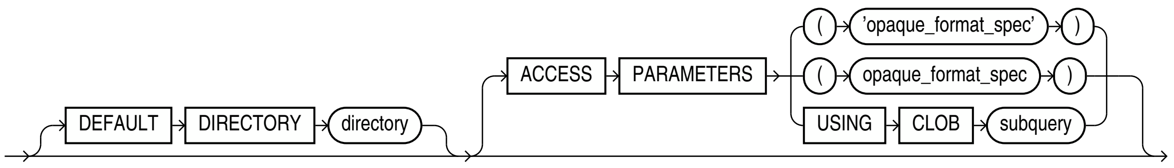


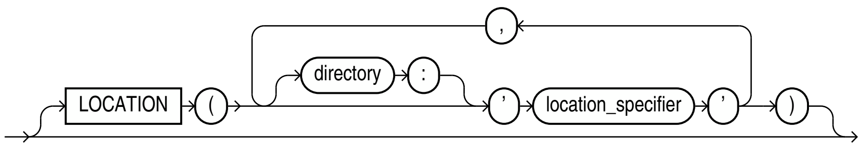
***alter_external_table*::=**



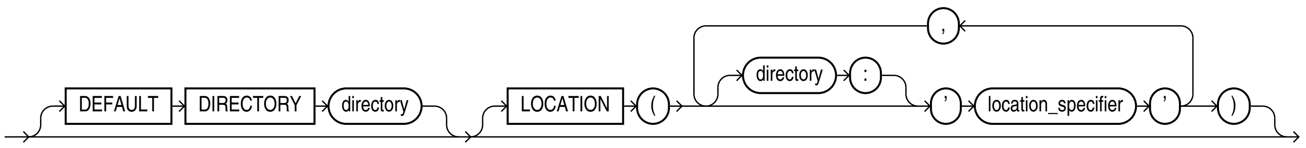
(*add_column_clause*::=, *modify_column_clauses*::=, *drop_column_clause*::=, *parallel_clause*::=, *external_table_data_props*::=)

***external_table_data_props*::=**

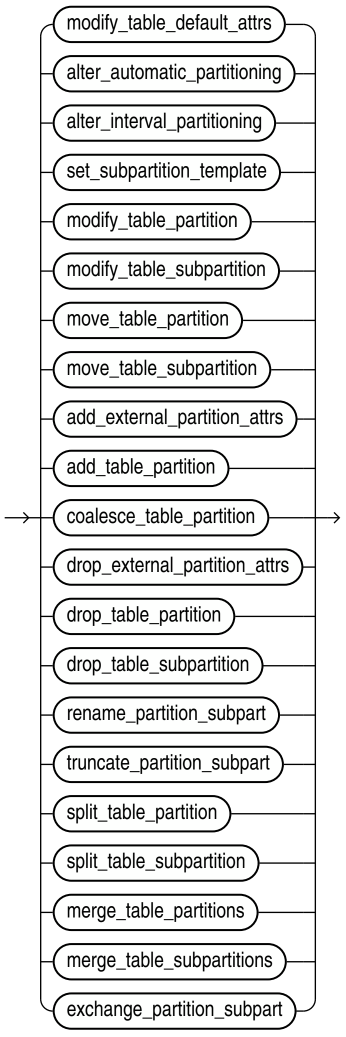




***external_part_subpart_data_props*::=**

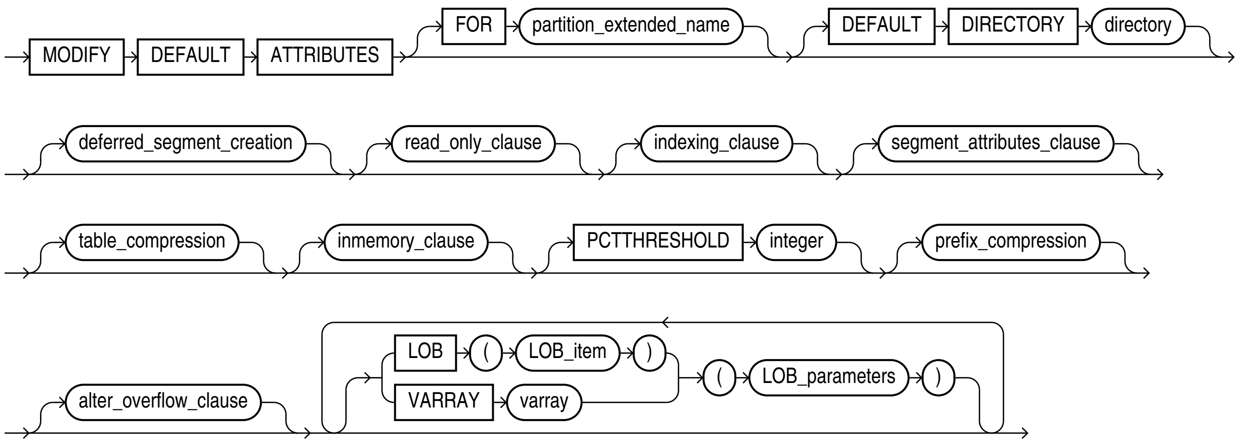


***alter_table_partitioning*::=**


modify_table_default_attrs
alter_automatic_partitioning
alter_interval_partitioning
set_subpartition_template
modify_table_partition
modify_table_subpartition
move_table_partition
move_table_subpartition
add_external_partition_attrs
add_table_partition
coalesce_table_partition
drop_external_partition_attrs
drop_table_partition
drop_table_subpartition
rename_partition_subpart
truncate_partition_subpart
split_table_partition
split_table_subpartition
merge_table_partitions
merge_table_subpartitions
exchange_partition_subpart

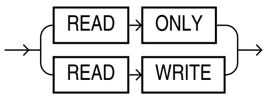
(*modify_table_default_attrs*::=, *alter_automatic_partitioning*::=, *alter_interval_partitioning*::=, *set_subpartition_template*::=, *modify_table_partition*::=, *modify_table_subpartition*::=, *move_table_partition*::=, *move_table_subpartition*::=, *add_table_partition*::=, *coalesce_table_partition*::=, *drop_table_partition*::=, *drop_table_subpartition*::=, *rename_partition_subpart*::=, *truncate_partition_subpart*::=, *split_table_partition*::=, *split_table_subpartition*::=, *merge_table_partitions*::=, *merge_table_subpartitions*::=, *exchange_partition_subpart*::=

***modify_table_default_attrs*::=**

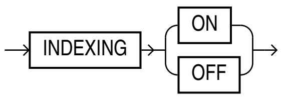


(*partition_extended_name*::=, *deferred_segment_creation*::= , *read_only_clause*::=, *indexing_clause*::=, *segment_attributes_clause*::=, *table_compression*::=, *inmemory_clause*::=, *prefix_compression*::=, *alter_overflow_clause*::=, *LOB_parameters*::=)

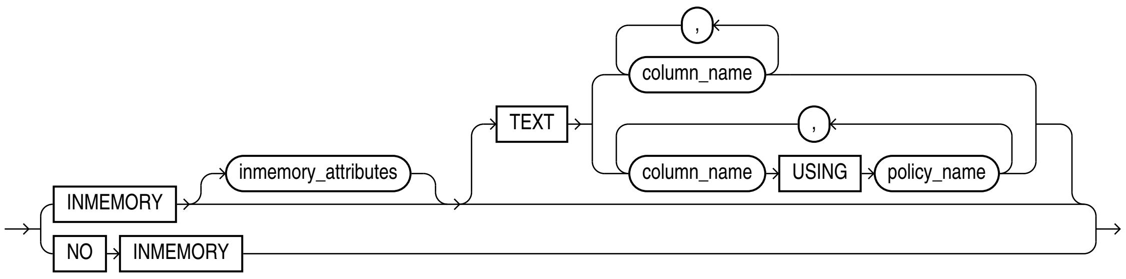
***read_only_clause*::=**



***indexing_clause*::=**

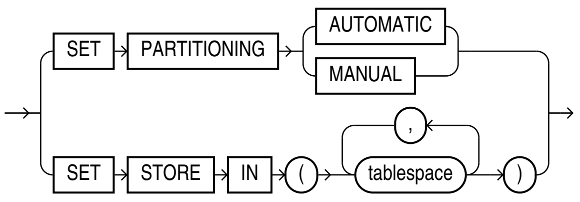


***inmemory_clause*::=**

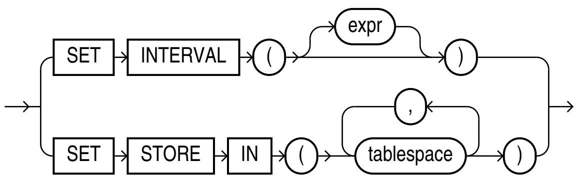


(*inmemory_attributes*::=)

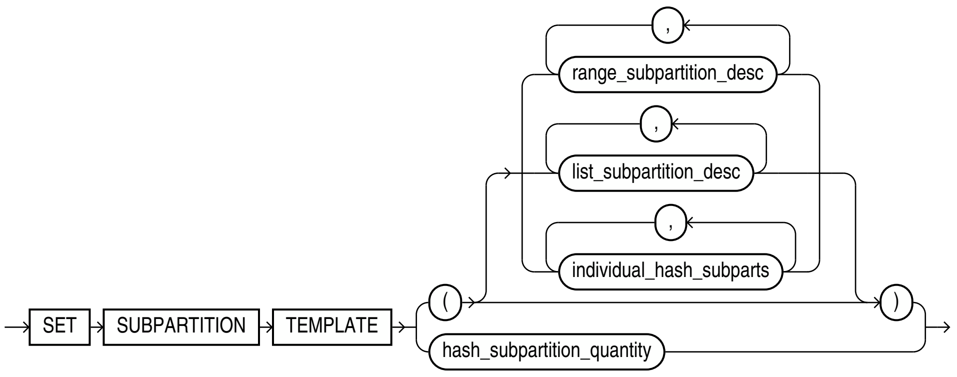
***alter_automatic_partitioning*::=**



***alter_interval_partitioning*::=**

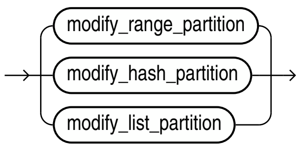


***set_subpartition_template*::=**



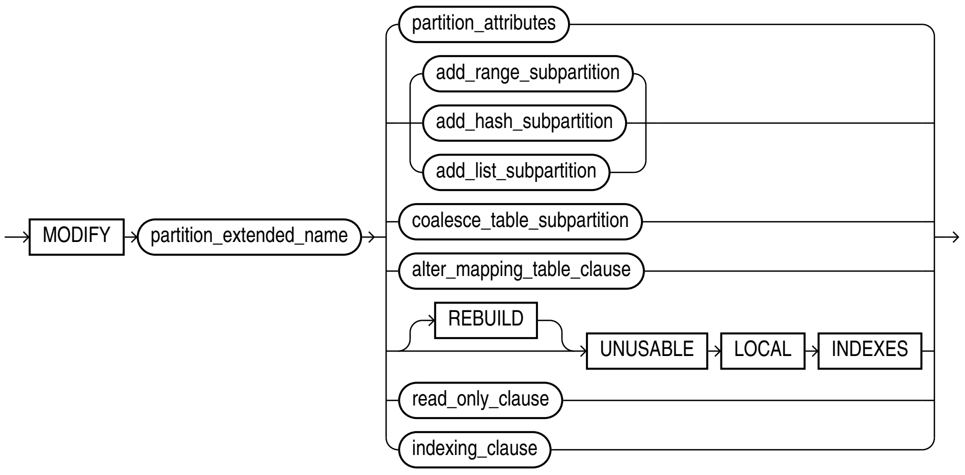
(*range_subpartition_desc*::=, *list_subpartition_desc*::=, *individual_hash_subparts*::=)

***modify_table_partition*::=**



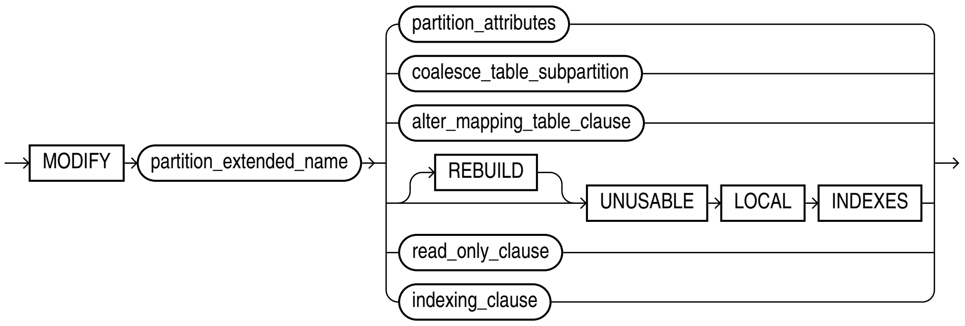
(*modify_range_partition*::=, *modify_hash_partition*::=, *modify_list_partition*::=)

***modify_range_partition*::=**



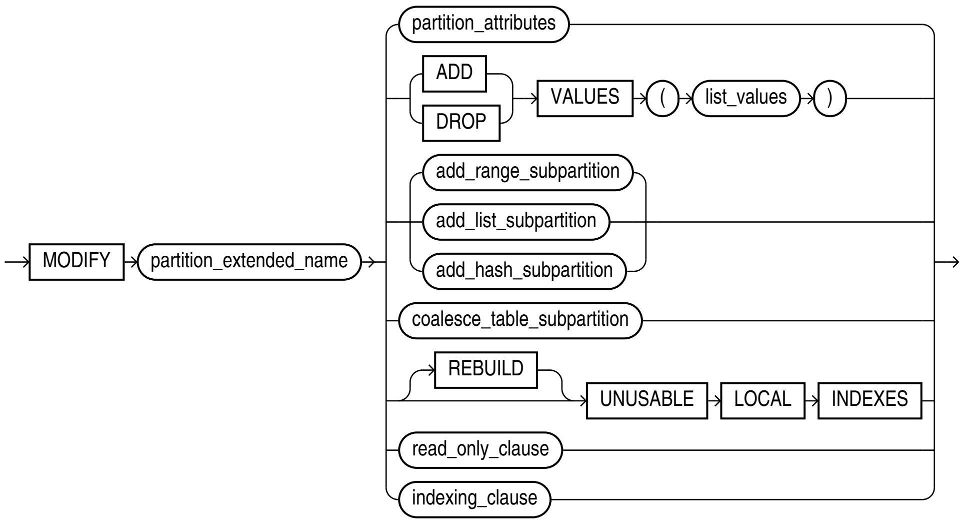
(*partition_extended_name*::=, *partition_attributes*::=, *add_range_subpartition*::=, *add_hash_subpartition*::=, *add_list_subpartition*::=, *coalesce_table_subpartition*::=, *alter_mapping_table_clauses*::=, *read_only_clause*::=, *indexing_clause*::=)

***modify_hash_partition*::=**



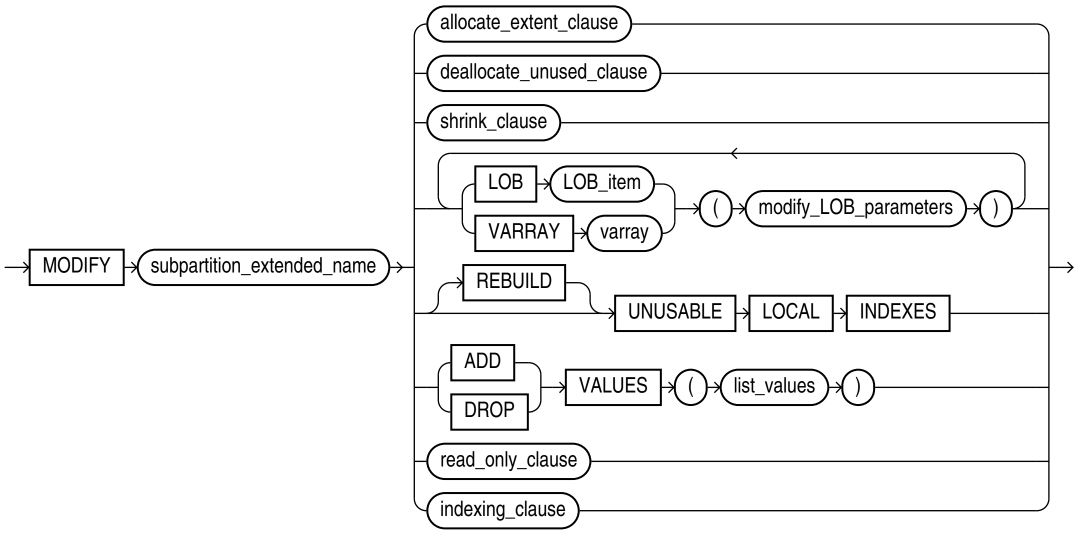
(*partition_extended_name*::=, *coalesce_table_subpartition*::=, *partition_attributes*::=, *alter_mapping_table_clauses*::=, *read_only_clause*::=, *indexing_clause*::=)

***modify_list_partition*::=**



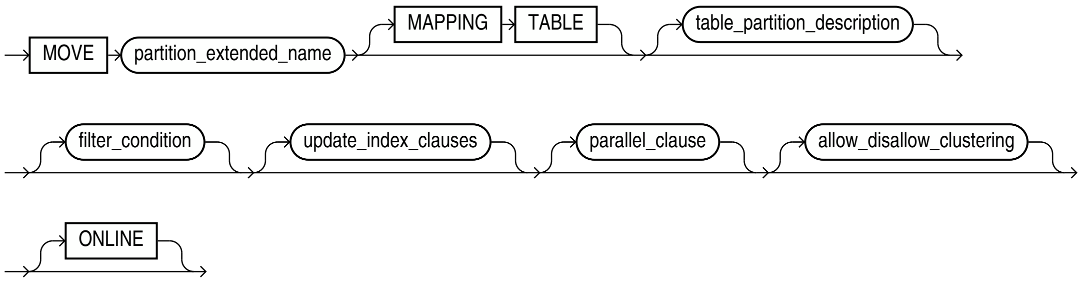
(*partition_extended_name*::=, *partition_attributes*::=, *list_values*::=, *add_range_subpartition*::=, *add_list_subpartition*::=, *add_hash_subpartition*::=, *coalesce_table_subpartition*::=, *read_only_clause*::=, *indexing_clause*::=)

***modify_table_subpartition*::=**



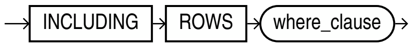
(*subpartition_extended_name*::=, *allocate_extent_clause*::=, *deallocate_unused_clause*::=, *shrink_clause*::=, *modify_LOB_parameters*::=, *list_values*::=, *read_only_clause*::=, *indexing_clause*::=)

***move_table_partition*::=**

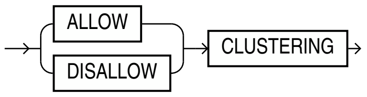


(*partition_extended_name*::=, *table_partition_description*::=, *filter_condition*::=, *update_index_clauses*::=, *parallel_clause*::=, *allow_disallow_clustering*::=)

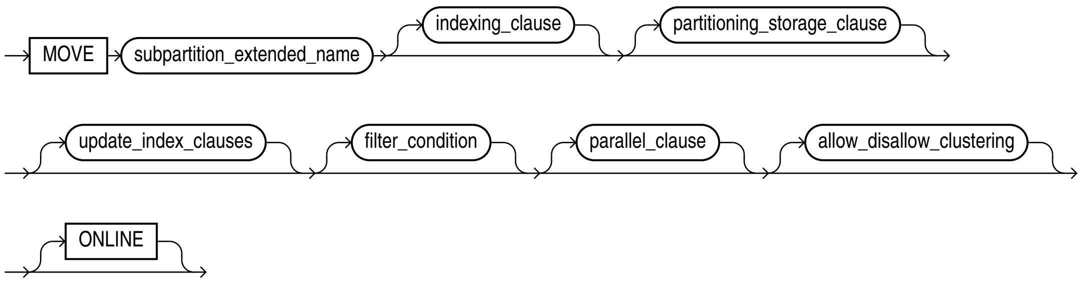
***filter_condition*::=**



***allow_disallow_clustering*::=**

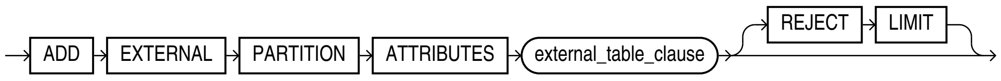


***move_table_subpartition*::=**

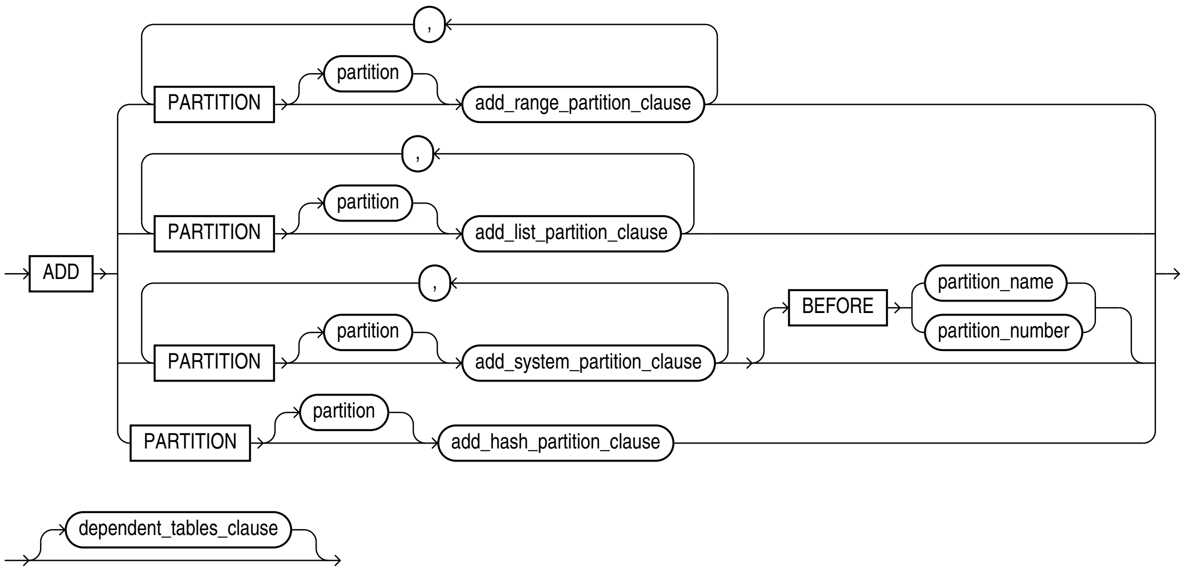

MOVE
subpartition_extended_name
indexing_clause
partitioning_storage_clause
update_index_clauses
filter_condition
parallel_clause
allow_disallow_clustering
ONLINE

(*subpartition_extended_name*::=, *indexing_clause*::=, *partitioning_storage_clause*::=, *update_index_clauses*::=, *filter_condition*::=, *parallel_clause*::=, *allow_disallow_clustering*::=)

add_external_partition_attrs

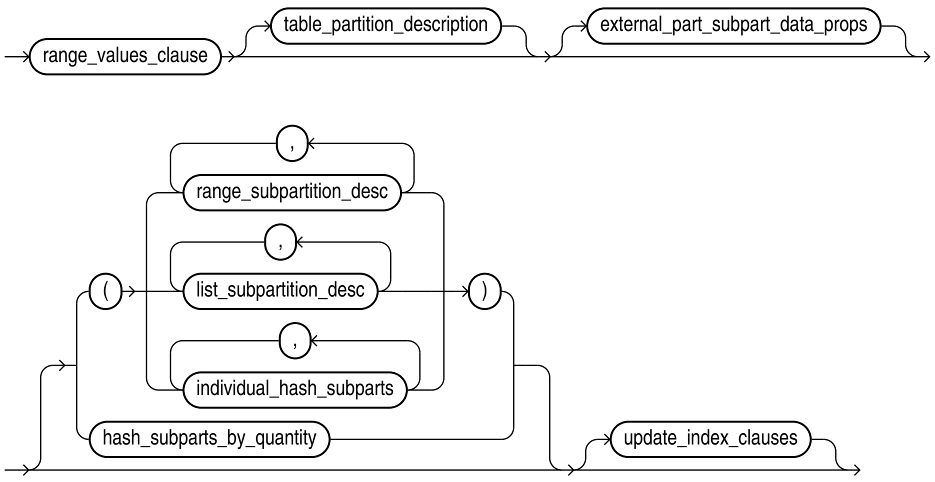


***add_table_partition*::=**



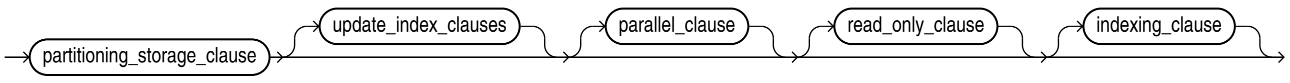
(*add_range_partition_clause*::=, *add_list_partition_clause*::=, *add_system_partition_clause*::=, *add_hash_partition_clause*::=, *dependent_tables_clause*:=)

***add_range_partition_clause*::=**



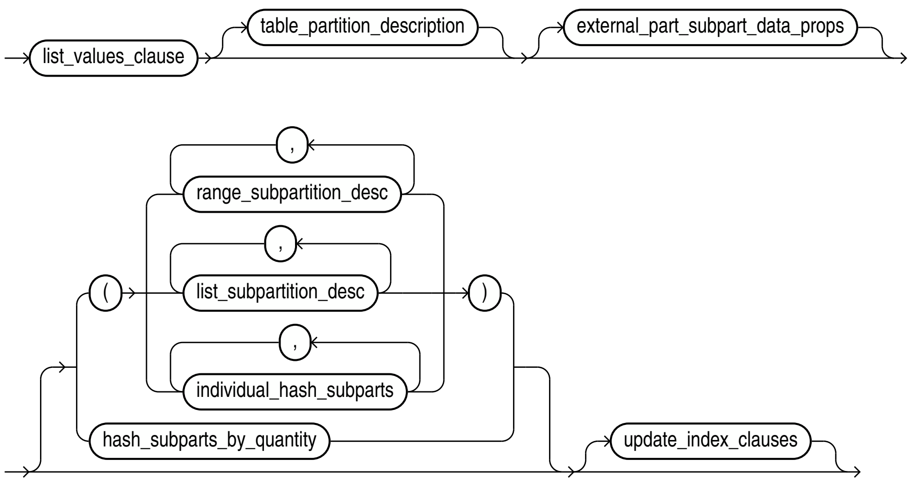
(*range_values_clause*::=, *table_partition_description*::=, *external_part_subpart_data_props*::=, *range_subpartition_desc*::=, *list_subpartition_desc*::=, *individual_hash_subparts*::=, *hash_subparts_by_quantity*::=, *update_index_clauses*::=)

***add_hash_partition_clause*::=**



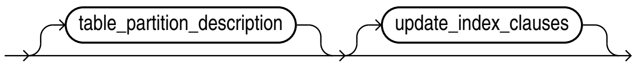
(*partitioning_storage_clause*::=, *update_index_clauses*::=, *parallel_clause*::=, *read_only_clause*::=, *indexing_clause*::=)

***add_list_partition_clause*::=**



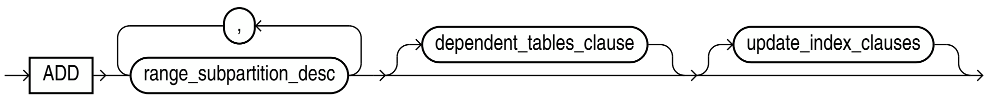
(*list_values_clause*::=, *table_partition_description*::=, *external_part_subpart_data_props*::=, *range_subpartition_desc*::=, *list_subpartition_desc*::=, *individual_hash_subparts*::=, *hash_subparts_by_quantity*::=, *update_index_clauses*::=)

***add_system_partition_clause*::=**



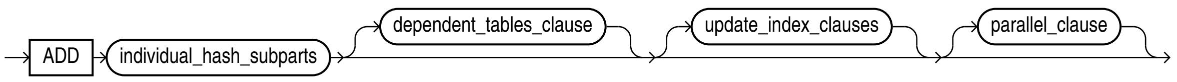
(*table_partition_description*::=, *update_index_clauses*::=)

***add_range_subpartition*::=**



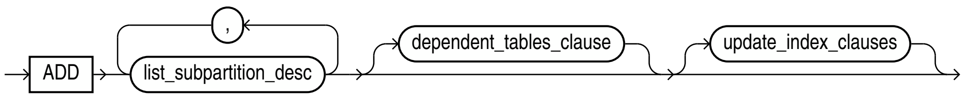
(*range_subpartition_desc*::=, *dependent_tables_clause*:=, *update_index_clauses*::=)

***add_hash_subpartition*::=**



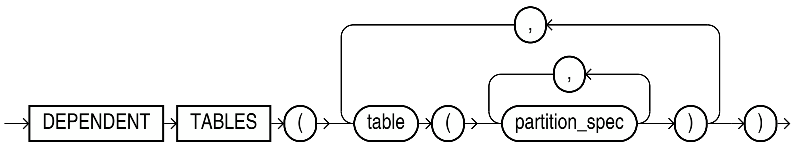
(*individual_hash_subparts*::=, *dependent_tables_clause*:=, *update_index_clauses*::=, *parallel_clause*::=)

add_list_subpartition::=



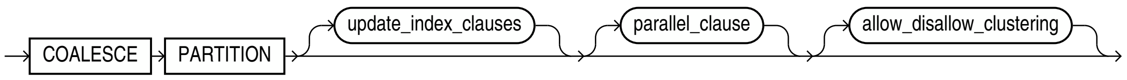
(*list_subpartition_desc::=*, *dependent_tables_clause:=*, *update_index_clauses::=*)

dependent_tables_clause:=



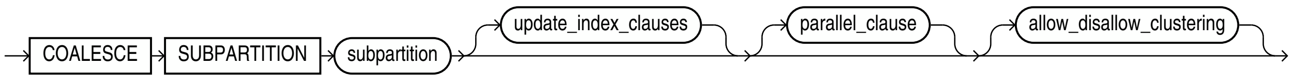
(*partition_spec::=*)

coalesce_table_partition::=



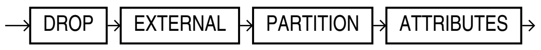
(*update_index_clauses::=*, *parallel_clause::=*, *allow_disallow_clustering::=*)

coalesce_table_subpartition::=

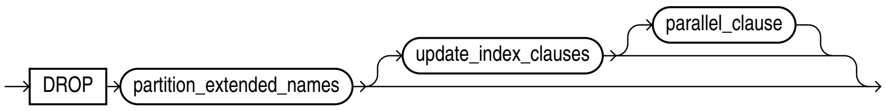


(*update_index_clauses::=*, *parallel_clause::=*, *allow_disallow_clustering::=*)

drop_external_partition_attrs::=

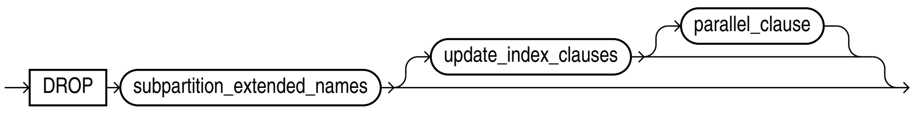


drop_table_partition::=



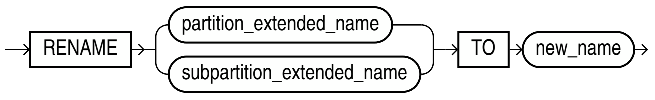
(*partition_extended_names::=*, *update_index_clauses::=*, *parallel_clause::=*)

***drop_table_subpartition*::=**



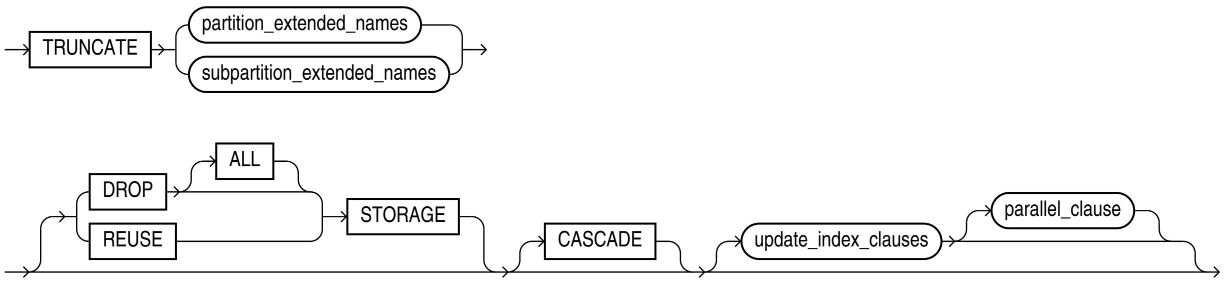
(*subpartition_extended_names*::=, *update_index_clauses*::=, *parallel_clause*::=)

***rename_partition_subpart*::=**



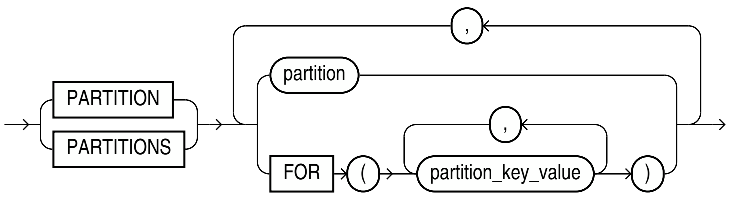
(*partition_extended_name*::=, *subpartition_extended_name*::=)

***truncate_partition_subpart*::=**

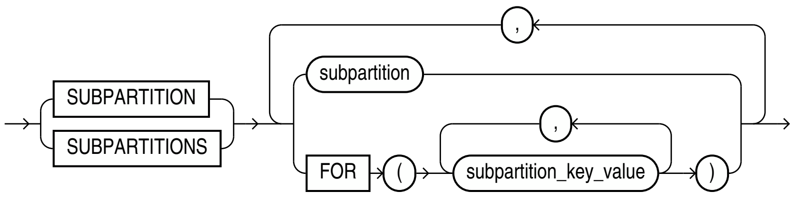


(*partition_extended_names*::=, *subpartition_extended_names*::=, *update_index_clauses*::=, *parallel_clause*::=)

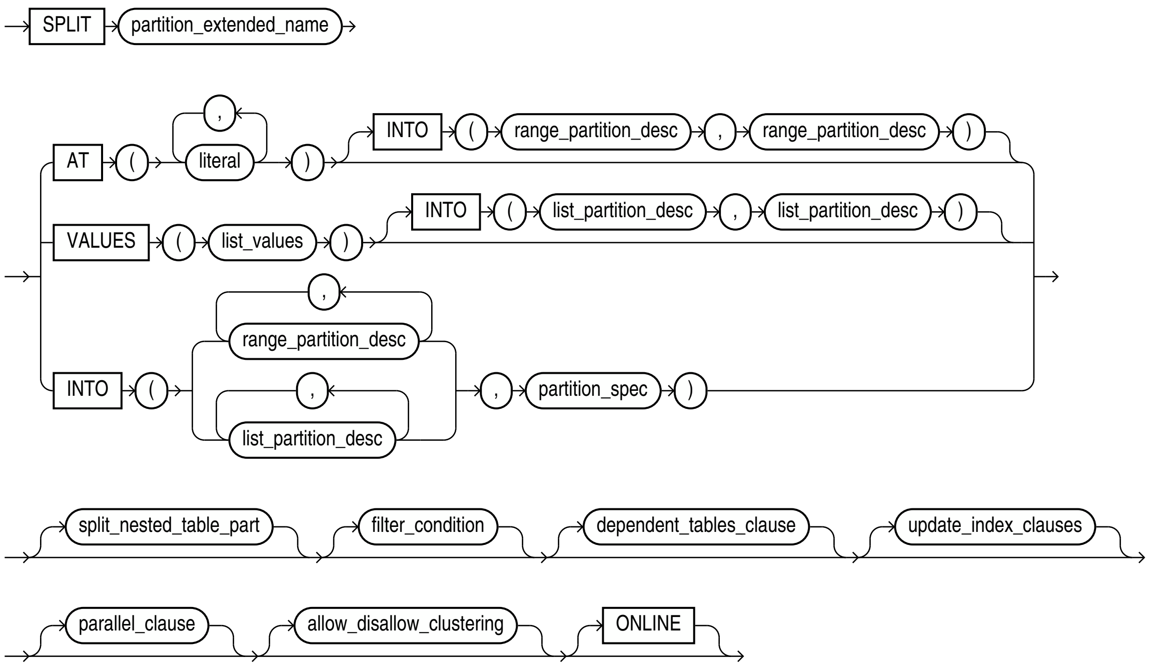
***partition_extended_names*::=**



***subpartition_extended_names*::=**

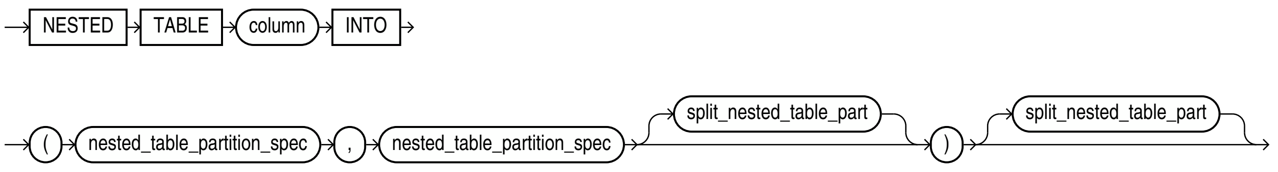


***split_table_partition*::=**

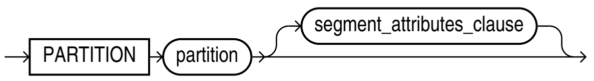


(*partition_extended_name*::=, *range_partition_desc*::=, *list_values*::=, *list_partition_desc*::=, *partition_spec*::=, *split_nested_table_part*::=, *filter_condition*::=, *dependent_tables_clause*:=, *update_index_clauses*::=, *parallel_clause*::=, *allow_disallow_clustering*::=)

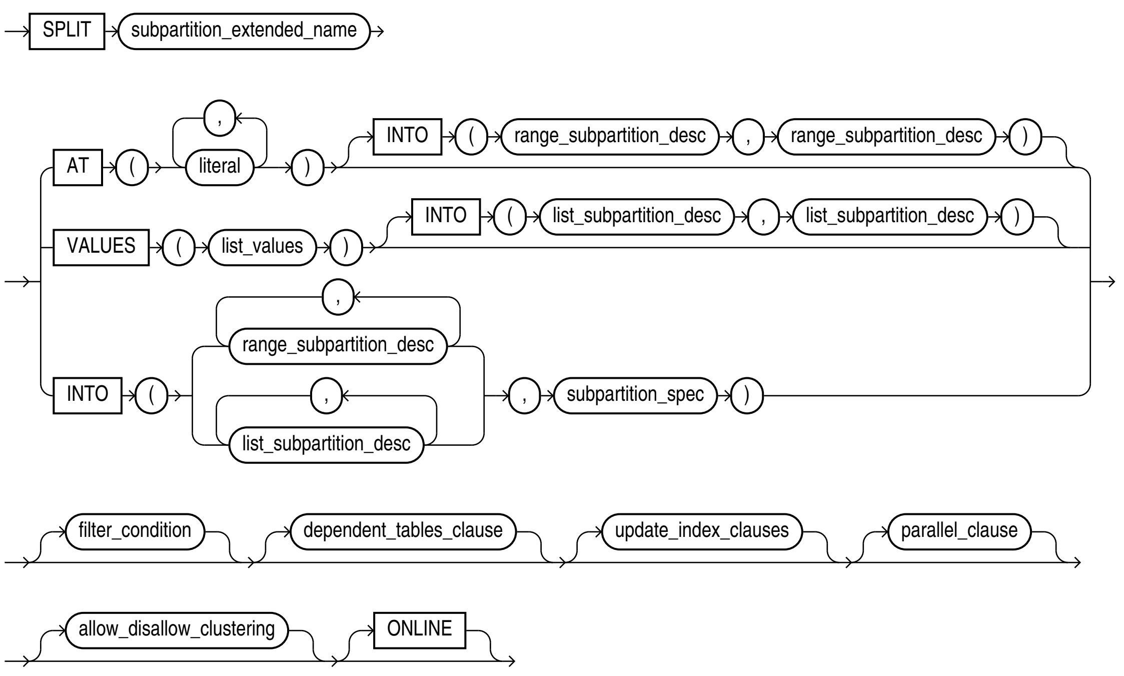
***split_nested_table_part*::=**



***nested_table_partition_spec*::=**

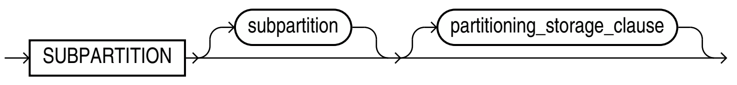


***split_table_subpartition*::=**

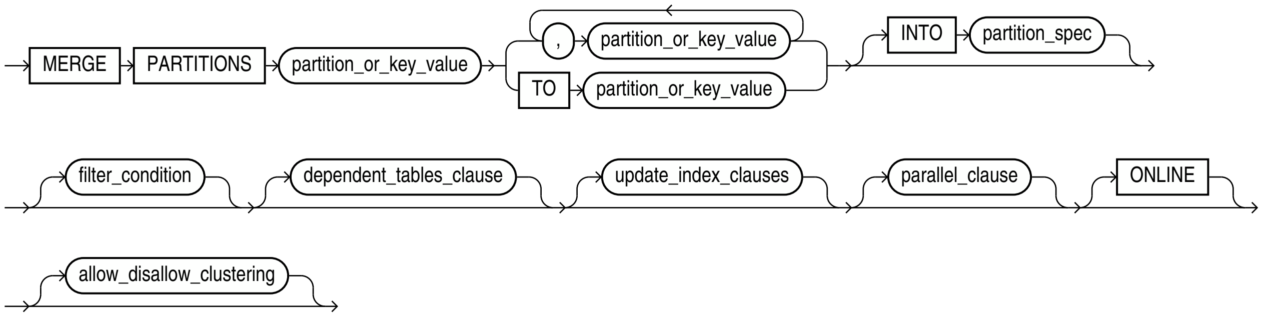


(*subpartition_extended_name*::=, *range_subpartition_desc*::=, *list_values*::=, *list_subpartition_desc*::=, *subpartition_spec*::=, *filter_condition*::=, *dependent_tables_clause*:=, *update_index_clauses*::=, *parallel_clause*::=, *allow_disallow_clustering*::=

***subpartition_spec*::=**

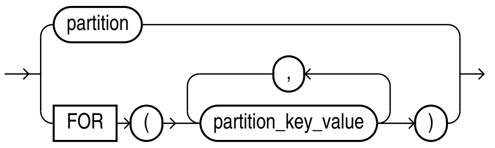


***merge_table_partitions*::=**

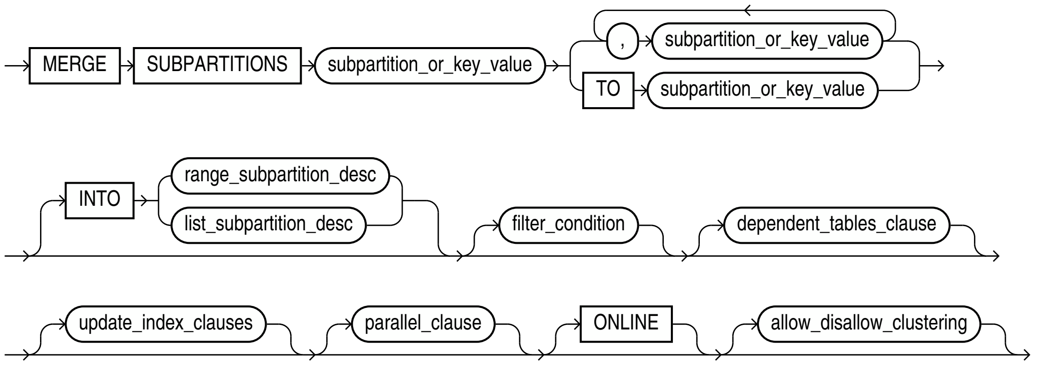


(*partition_or_key_value*::=, *partition_spec*::=, *filter_condition*::=, *dependent_tables_clause*:=, *update_index_clauses*::=, *parallel_clause*::=, *allow_disallow_clustering*::=)

***partition_or_key_value*::=**

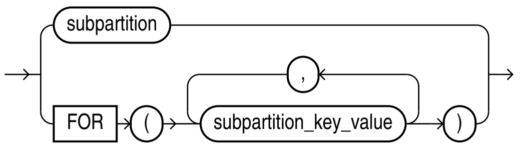


***merge_table_subpartitions*::=**

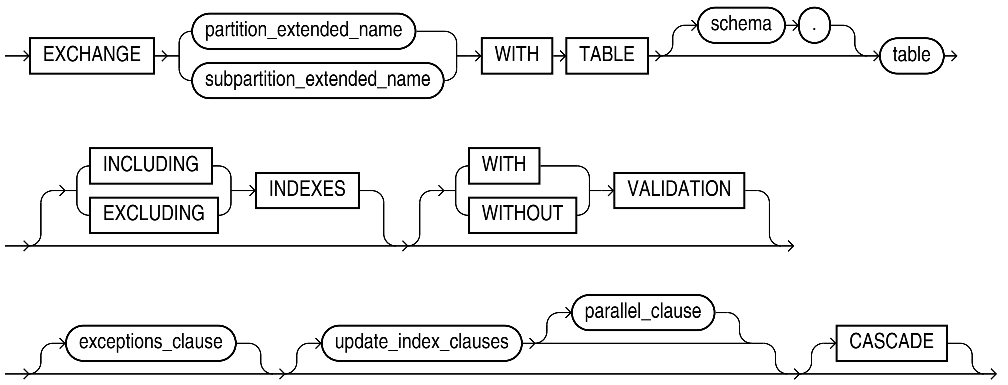


(*subpartition_or_key_value*::=, *range_subpartition_desc*::=, *list_subpartition_desc*::=, *filter_condition*::=, *dependent_tables_clause*:=, *update_index_clauses*::=, *parallel_clause*::=, *allow_disallow_clustering*::=)

***subpartition_or_key_value*::=**

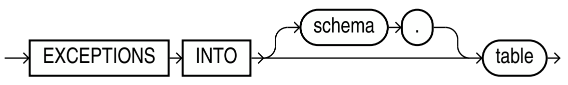


***exchange_partition_subpart*::=**

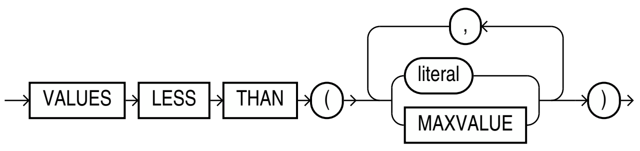


(*partition_extended_name::=*, *subpartition_extended_name::=*, *exceptions_clause::=*, *update_index_clauses::=*, *parallel_clause::=*)

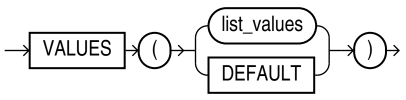
exceptions_clause::=



range_values_clause::=

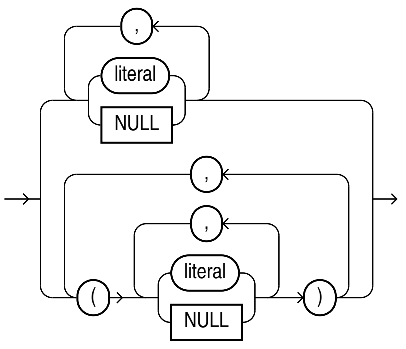


list_values_clause::=

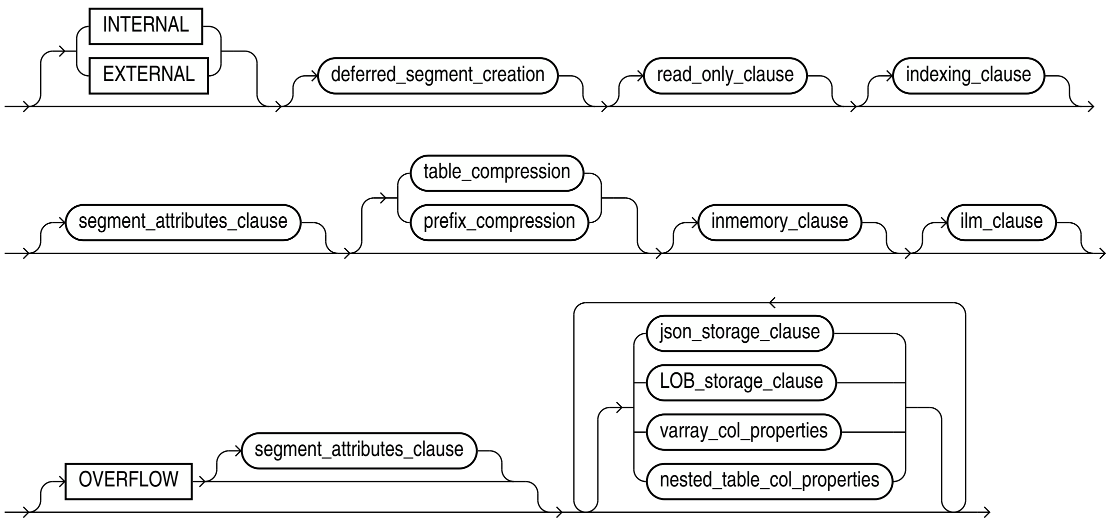


(*list_values::=*)

list_values::=

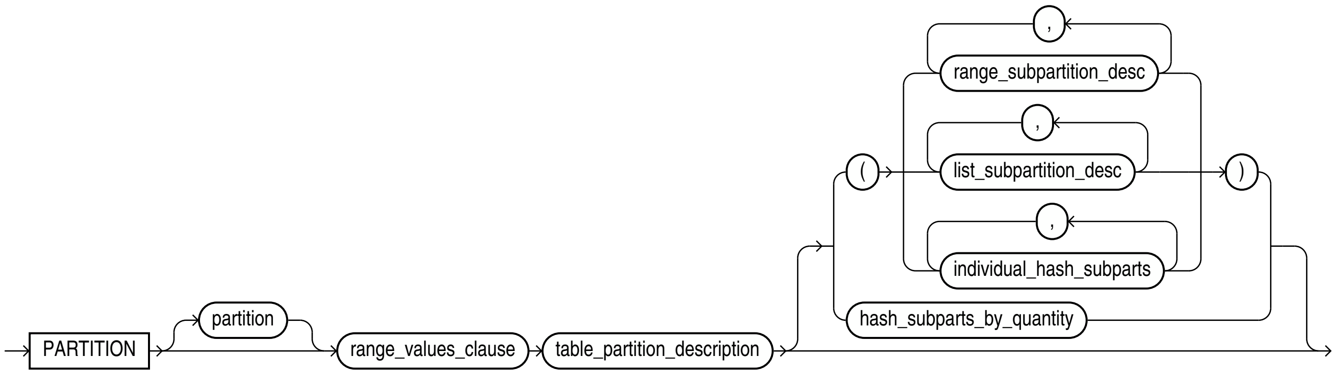


***table_partition_description*::=**



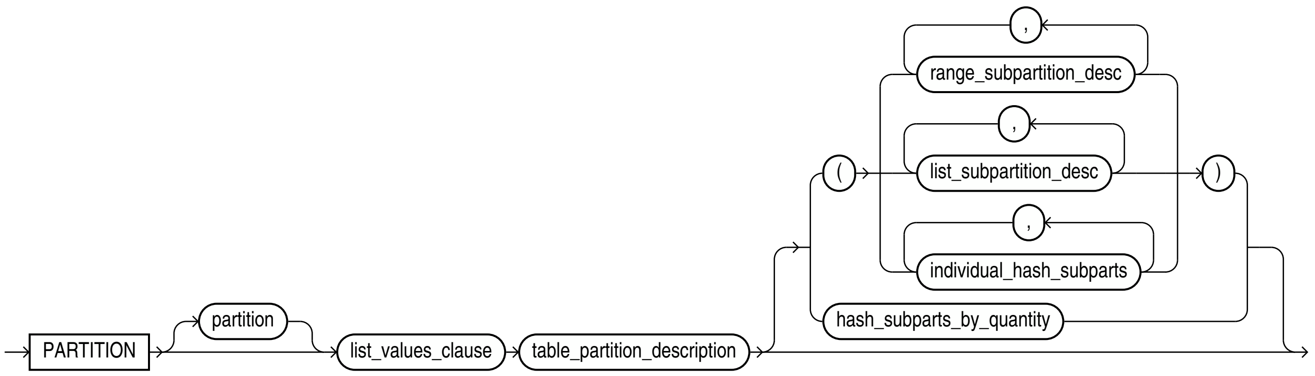
(*deferred_segment_creation*::= , *read_only_clause*::=, *indexing_clause*::=, *segment_attributes_clause*::=, *table_compression*::=, *prefix_compression*::=, *inmemory_clause*::=, *LOB_storage_clause*::=, *varray_col_properties*::=)

***range_partition_desc*::=**



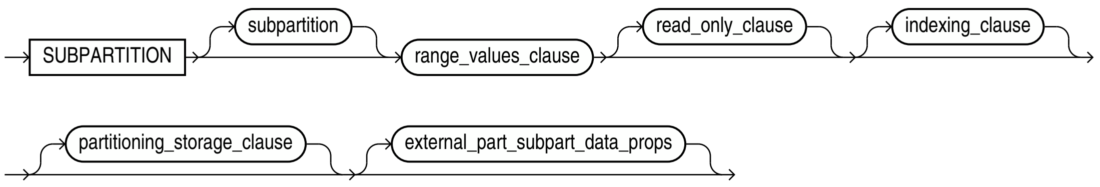
(*range_values_clause*::=, *table_partition_description*::=, *range_subpartition_desc*::=, *list_subpartition_desc*::=)

***list_partition_desc*::=**



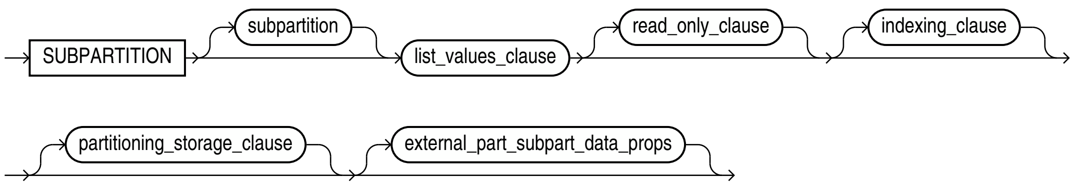
(*list_values_clause*::=, *table_partition_description*::=, *range_subpartition_desc*::=, *list_subpartition_desc*::=)

***range_subpartition_desc*::=**



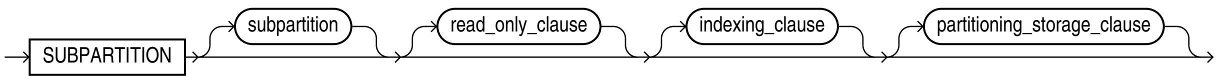
(*range_values_clause*::=, *read_only_clause*::=, *indexing_clause*::=, *partitioning_storage_clause*::=, *external_part_subpart_data_props*::=)

***list_subpartition_desc*::=**



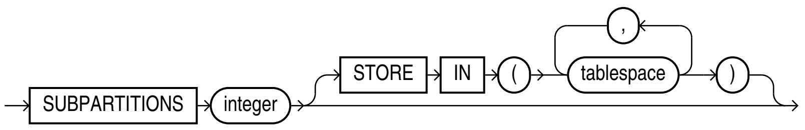
(*list_values_clause*::=, *read_only_clause*::=, *indexing_clause*::=, *partitioning_storage_clause*::=, *external_part_subpart_data_props*::=)

***individual_hash_subparts*::=**

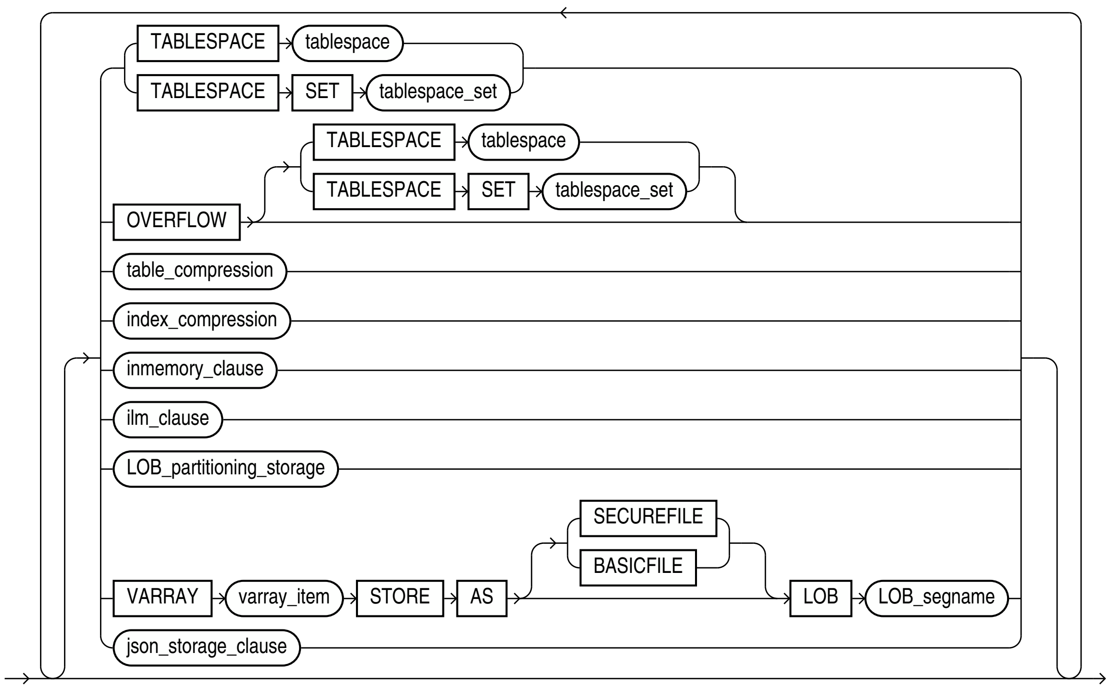


(*read_only_clause*::=, *indexing_clause*::=, *partitioning_storage_clause*::=)

hash_subparts_by_quantity::=

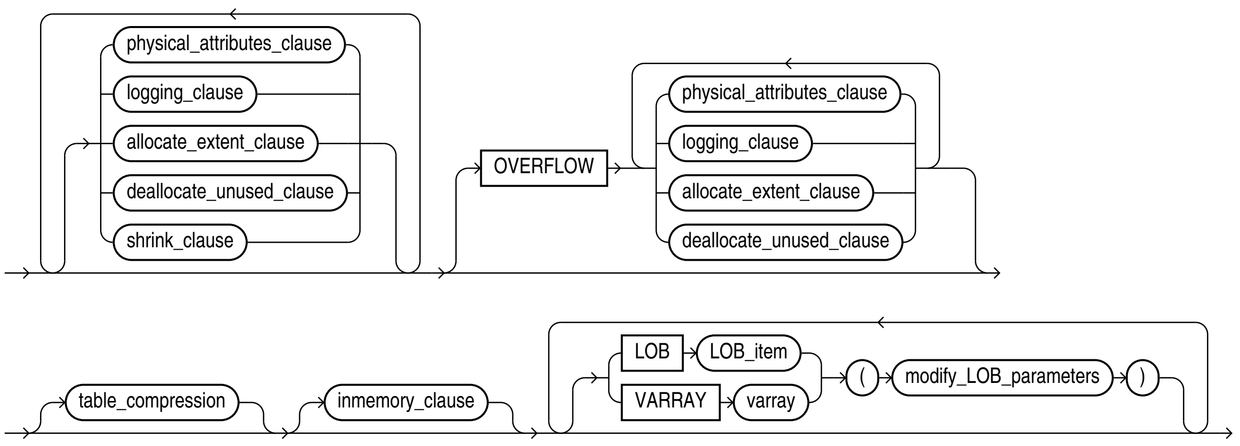


partitioning_storage_clause::=



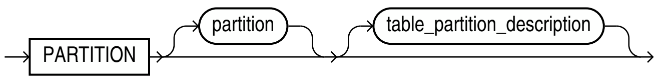
(`TABLESPACE SET`: not supported with `ALTER TABLE`, *table_compression*::=, *index_compression*::=, *inmemory_clause*::=, *LOB_partitioning_storage*::=)

partition_attributes::=



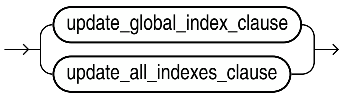
(*physical_attributes_clause*::=, *logging_clause*::=, *allocate_extent_clause*::=, *deallocate_unused_clause*::=, *shrink_clause*::=, *table_compression*::=, *inmemory_clause*::=, *modify_LOB_parameters*::=)

***partition_spec*::=**



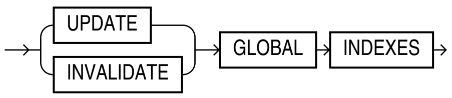
(*table_partition_description*::=)

***update_index_clauses*::=**

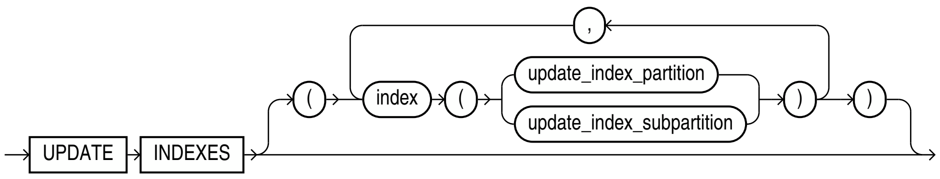


(*update_global_index_clause*::=, *update_all_indexes_clause*::=)

***update_global_index_clause*::=**

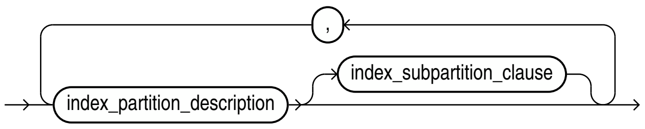


***update_all_indexes_clause*::=**



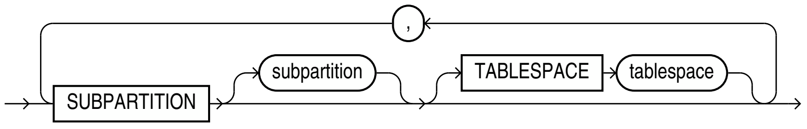
(*update_index_partition*::=, *update_index_subpartition*::=)

***update_index_partition*::=**

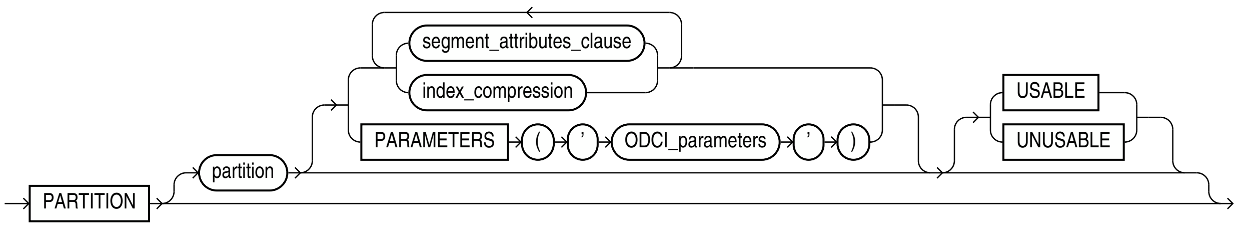


(*index_partition_description*::=, *index_subpartition_clause*::=)

***update_index_subpartition*::=**

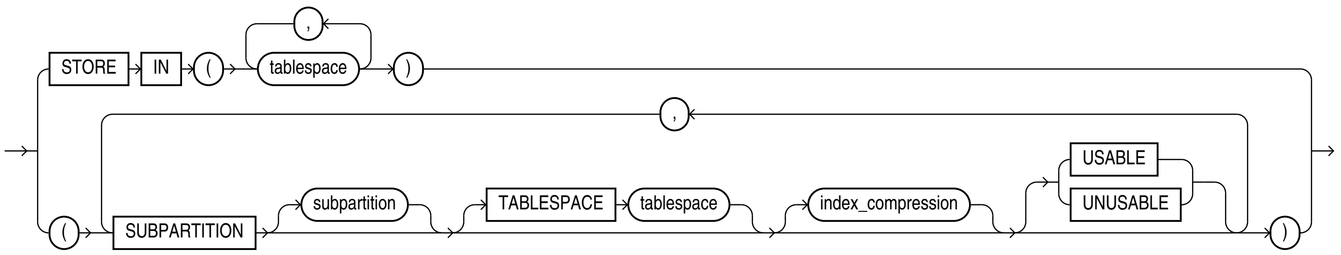


***index_partition_description*::=**



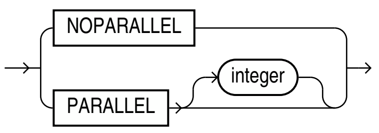
(*segment_attributes_clause*::=, *index_compression*::=)

***index_subpartition_clause*::=**

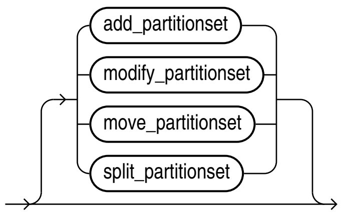


(*index_compression*::=)

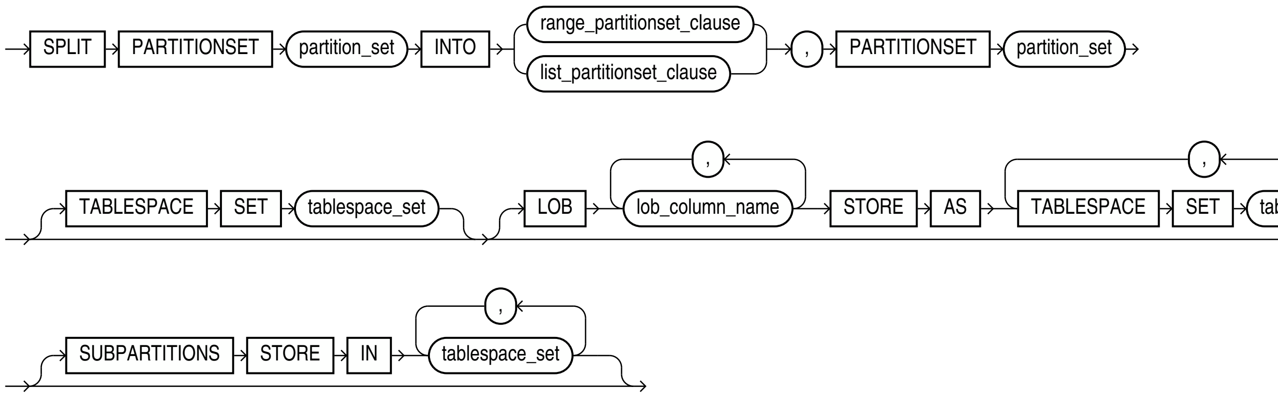
***parallel_clause*::=**



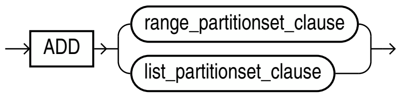
***alter_table_partitionset*::=**



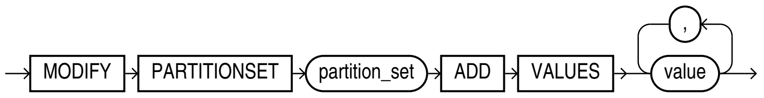
split_partitionset::=



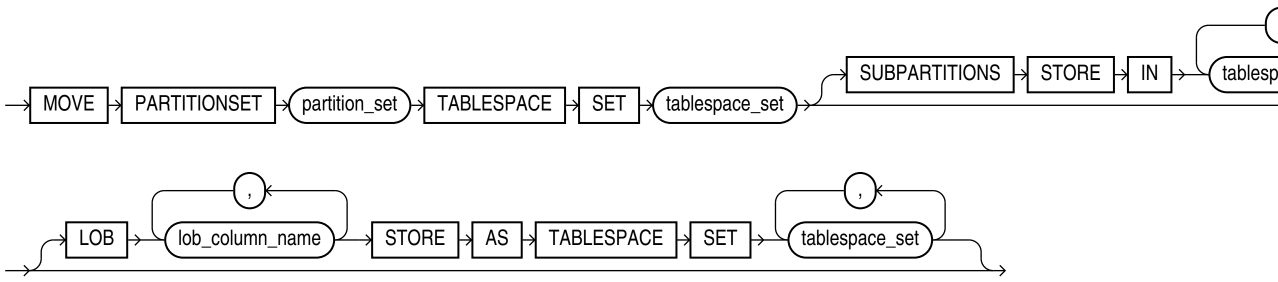
add_partitionset::=



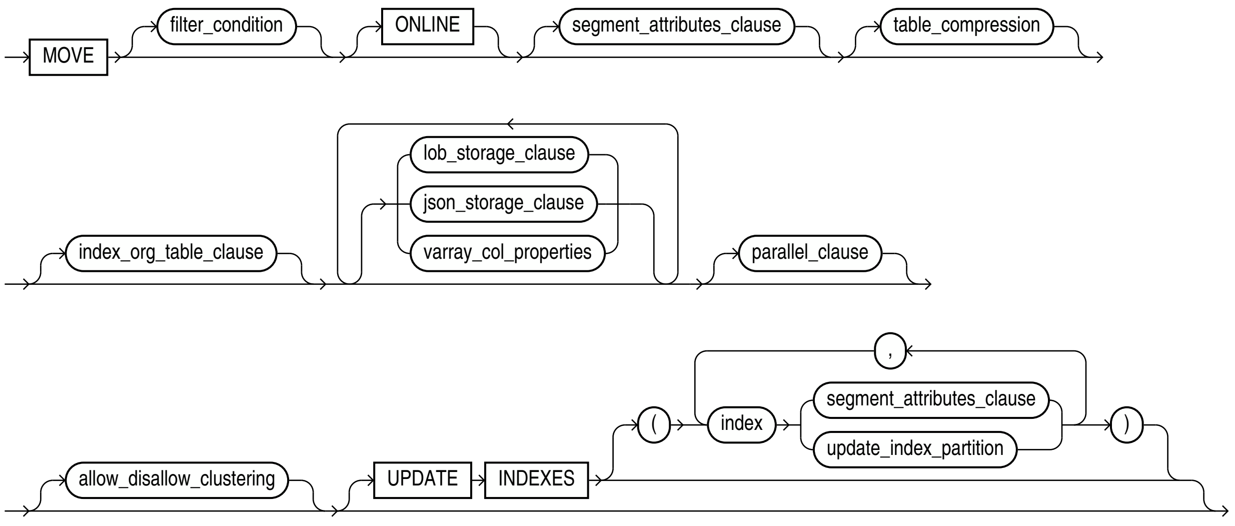
modify_partitionset::=



move_partitionset::=

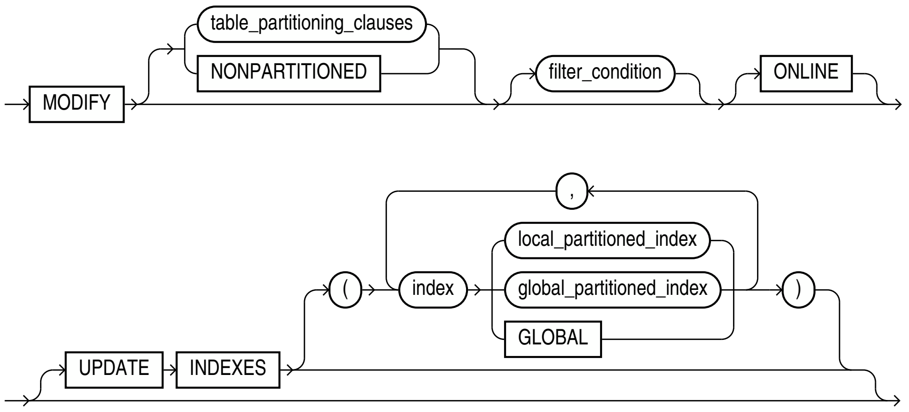


move_table_clause::=

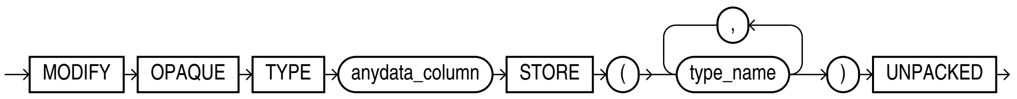


(filter_condition::=, segment_attributes_clause::=, table_compression::=, index_org_table_clause::=, LOB_storage_clause::=, varray_col_properties::=, parallel_clause::=, allow_disallow_clustering::=, update_index_partition::=)

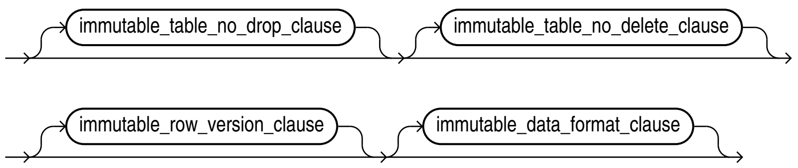
***modify_to_partitioned*::=**



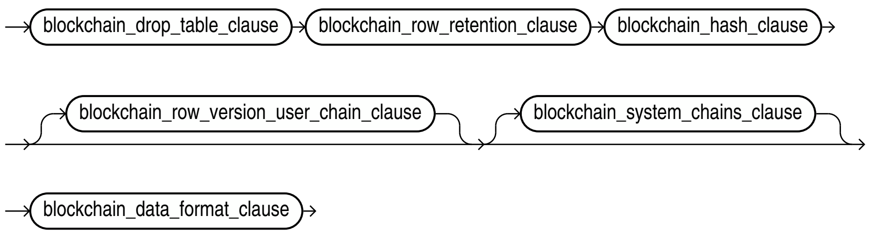
***modify_opaque_type*::=**



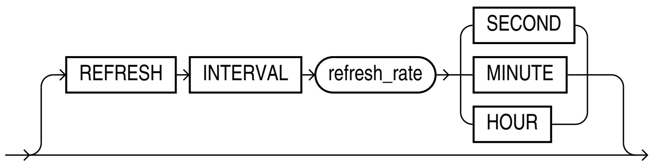
***immutable_table_clauses*::=**


immutable_table_no_drop_clause
immutable_table_no_delete_clause
immutable_row_version_clause
immutable_data_format_clause

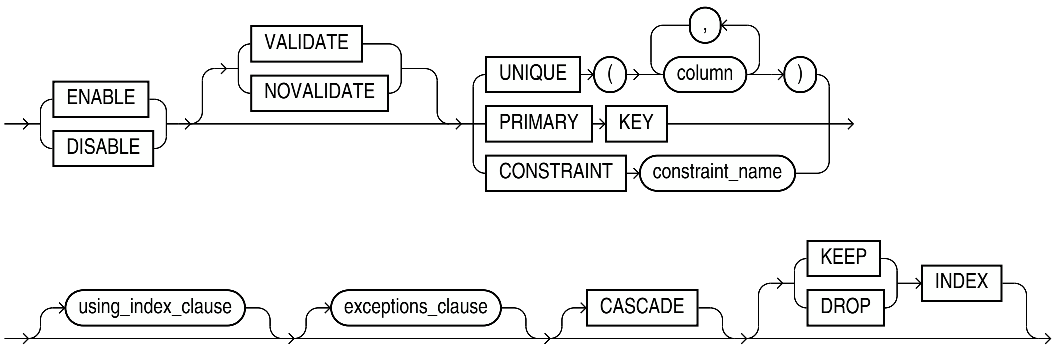
***blockchain_table_clauses*::=**


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

***duplicated_table_refresh*::=**

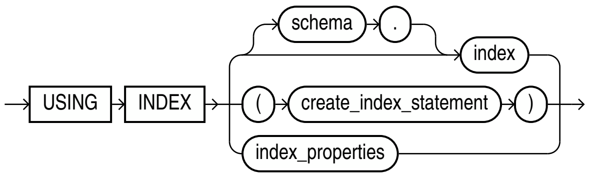


***enable_disable_clause*::=**



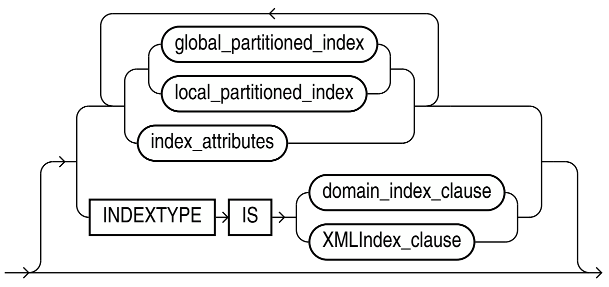
(*using_index_clause*::=, *exceptions_clause*::=,)

***using_index_clause*::=**



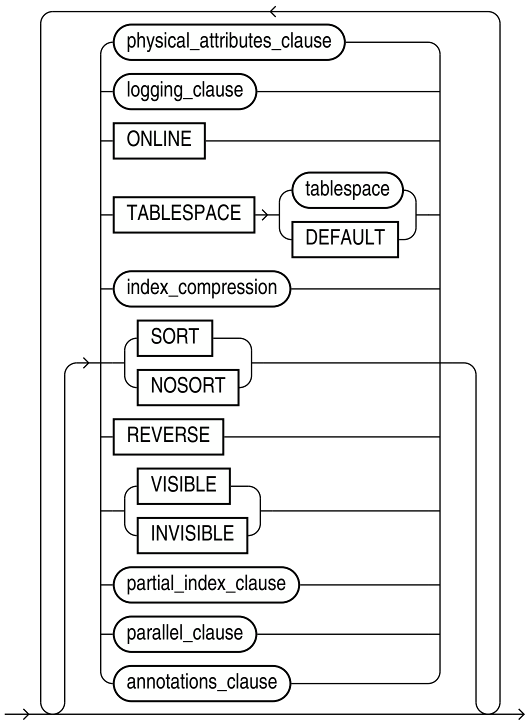
(*create_index*::=, *index_properties*::=)

***index_properties*::=**



(*global_partitioned_index*::=, *local_partitioned_index*::=—part of CREATE INDEX, *index_attributes*::=, *domain_index_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*)

Semantics

Many clauses of the ALTER TABLE statement have the same functionality they have in a CREATE TABLE statement. For more information on such clauses, see CREATE TABLE.

Note:

Operations performed by the `ALTER TABLE` statement can cause Oracle Database to invalidate procedures and stored functions that access the table. For information on how and when the database invalidates such objects, see *Oracle Database Development Guide*.

IF EXISTS

Specify `IF EXISTS` to alter an existing table.

Specifying `IF NOT EXISTS` with `ALTER VIEW` results in `ORA-11544: Incorrect IF EXISTS clause for ALTER/DROP statement`.

schema

Specify the schema containing the table. If you omit *`schema`*, then Oracle Database assumes the table is in your own schema.

table

Specify the name of the table to be altered.

Note:

If you alter a table that is a master table for one or more materialized views, then Oracle Database marks the materialized views `INVALID`. Invalid materialized views cannot be used by query rewrite and cannot be refreshed. For information on revalidating a materialized view, see ALTER MATERIALIZED VIEW .

See Also:

Oracle Database Data Warehousing Guide for more information on materialized views in general

Restrictions on Altering Temporary Tables

You can modify, drop columns from, or rename a temporary table. However, for a temporary table you cannot:

- Add columns of nested table type. You can add columns of other types.
- Specify referential integrity (foreign key) constraints for an added or modified column.
- Specify the following clauses of the *`LOB_storage_clause`* for an added or modified LOB column: `TABLESPACE`, *`storage_clause`*, *`logging_clause`*, *`allocate_extent_clause`*, or *`deallocate_unused_clause`*.
- Specify the *`physical_attributes_clause`*, *`nested_table_col_properties`*, *`parallel_clause`*, *`allocate_extent_clause`*, *`deallocate_unused_clause`*, or any of the index-organized table clauses.

- Exchange partitions between a partition and a temporary table.
- Specify the *logging_clause*.
- Specify MOVE.
- Add an INVISIBLE column or modify an existing column to be INVISIBLE.

Restrictions on Altering External Tables

You can add, drop, or modify the columns of an external table. However, for an external table you cannot:

- Add a LONG, LOB, or object type column or change the data type of an external table column to any of these data types.
- Modify the storage parameters of an external table.
- Specify the *logging_clause*.
- Specify MOVE.
- Add an INVISIBLE column or modify an existing column to be INVISIBLE.

memoptimize_read_clause

Use this clause to improve 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.
- You must explicitly disable the table for NO MEMOPTIMIZE FOR READ when you no longer need it.

memoptimize_write_clause

Use this clause to enable fast ingest. Fast ingest optimizes the processing of high frequency single row data inserts from Internet of Things (IoT) applications by using a large buffering pool to store the inserts before writing them to disk.

Restrictions

Blockchain and immutable tables do not support the *memoptimize_write_clause*.

alter_table_properties

Use the *alter_table_clauses* to modify a database table.

physical_attributes_clause

The *physical_attributes_clause* lets you change the value of the PCTFREE, PCTUSED, and INITRANS parameters and storage characteristics. Refer to *physical_attributes_clause* and storage_clause for a full description of these parameters and characteristics.

Restrictions on Altering Table Physical Attributes

Altering physical attributes is subject to the following restrictions:

- You cannot specify the `PCTUSED` parameter for the index segment of an index-organized table.
- If you attempt to alter the storage attributes of tables in locally managed tablespaces, then Oracle Database raises an error. However, if some segments of a partitioned table reside in a locally managed tablespace and other segments reside in a dictionary-managed tablespace, then the database alters the storage attributes of the segments in the dictionary-managed tablespace but does not alter the attributes of the segments in the locally managed tablespace, and does not raise an error.
- For segments with automatic segment-space management, the database ignores attempts to change the `PCTUSED` setting. If you alter the `PCTFREE` setting, then you must subsequently run the `DBMS_REPAIR.SEGMENT_FIX_STATUS` procedure to implement the new setting on blocks already allocated to the segment.

Cautions on Altering Tables Physical Attributes

The values you specify in this clause affect the table as follows:

- For a nonpartitioned table, the values you specify override any values specified for the table at create time.
- For a range-, list-, or hash-partitioned table, the values you specify are the default values for the table and the actual values for every existing partition, overriding any values already set for the partitions. To change default table attributes without overriding existing partition values, use the *`modify_table_default_attrs`* clause.
- For a composite-partitioned table, the values you specify are the default values for the table and all partitions of the table and the actual values for all subpartitions of the table, overriding any values already set for the subpartitions. To change default partition attributes without overriding existing subpartition values, use the *`modify_table_default_attrs`* clause with the `FOR PARTITION` clause.

logging_clause

Use the *`logging_clause`* to change the logging attribute of the table. The *`logging_clause`* specifies whether subsequent `ALTER TABLE` ... `MOVE` and `ALTER TABLE` ... `SPLIT` operations will be logged or not logged.

When used with the *`modify_table_default_attrs`* clause, this clause affects the logging attribute of a partitioned table.

See Also:

- *logging_clause* for a full description of this clause
- *Oracle Database VLDB and Partitioning Guide* for more information about the *`logging_clause`* and parallel DML

table_compression

The *`table_compression`* clause is valid only for heap-organized tables. Use this clause to instruct Oracle Database whether to compress data segments to reduce disk and memory use. Refer to the `CREATE TABLE` *table_compression* for the full semantics of this clause and for information on creating objects with table compression.

Note:

The first time a table is altered in such a way that compressed data will be added, all bitmap indexes and bitmap index partitions on that table must be marked `UNUSABLE`.

inmemory_table_clause

Use this clause to enable or disable a table or table column for the In-Memory Column Store (IM column store), or to change the In-Memory attributes for a table or table column.

- Specify `INMEMORY` to enable a table for the IM column store, or to change the *`inmemory_attributes`* for a table that is already enabled for the IM column store.
- Specify `NO INMEMORY` to disable a table for the IM column store.
- Specify the *`inmemory_column_clause`* to enable or disable a table column for the IM column store, or to change the *`inmemory_memcompress`* setting for a table column. If you specify this clause when the table or partition is disabled for the IM column store, then the column settings will take effect when the table or partition is subsequently enabled for the IM column store. Regardless of whether the table or partition is enabled or disabled for the IM column store, when you specify `NO INMEMORY` for a column, any previously specified *`inmemory_memcompress`* setting for the column is lost. Refer to the *inmemory_column_clause* of `CREATE TABLE` for the full semantics of this clause.

This *`inmemory_table_clause`* has the same semantics as the *inmemory_table_clause* of `CREATE TABLE`, with the following additions:

- When you specify the *`inmemory_memcompress`* clause to change the data compression method for a table that is already enabled for the IM column store, any columns that were previously assigned a specific data compression method will retain that data compression method. Refer to the *inmemory_memcompress* clause of `CREATE TABLE` for more information on this clause.
- When you specify the *`inmemory_distribute`* clause, if you omit one subclause, then its setting remains unchanged. That is, if you specify only the `AUTO` or `BY` clause, then the `FOR SERVICE` setting for the table remains unchanged, and if you specify only the `FOR SERVICE` clause, then the `AUTO` or `BY` setting for the table remains unchanged. If you omit both subclauses and specify only the `DISTRIBUTE` keyword, then the table is assigned the `DISTRIBUTE AUTO` setting and its `FOR SERVICE` setting remains unchanged. Refer to the *inmemory_distribute* clause of `CREATE TABLE` for more information on this clause.
- When you specify `NO INMEMORY` to disable a partitioned or nonpartitioned table for the IM column store, any column-level In-Memory settings are lost. If you subsequently enable the table for the IM column store, then all columns will use the In-Memory settings for the table, unless you specify otherwise when enabling the table.
- When you specify `NO INMEMORY` to disable a partition for the IM column store, the column-level In-Memory settings are retained, even if all partitions in the table are disabled. If you subsequently enable the table or a partition for the IM column store, then the column-level In-Memory settings will go into effect, unless you specify otherwise when enabling the table or partition.
- If a table is currently populated in the IM column store and you change any *`inmemory_attribute`* of the table other than `PRIORITY`, then the database evicts the table from the IM column store. The repopulation behavior depends on the `PRIORITY` setting.

inmemory_clause

Use this clause to enable or disable a table partition for the IM column store, or to change the In-Memory parameters for a table partition. This clause has the same semantics in `CREATE TABLE` and `ALTER TABLE`. Refer to the *inmemory_clause* in the documentation on `CREATE TABLE` for the full semantics of this clause.

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

For more details on the In-Memory column architecture see Oracle Database In-Memory Guide

Restriction

If a segment on disk is 64 KB or less, then it is not populated in the IM column store. Therefore, some small database objects that were enabled for the IM column store might not be populated.

ilm_clause

Use this clause to add, delete, enable, or disable Automatic Data Optimization policies for the table.

ADD POLICY

Specify this clause to add a policy for the table.

Use *`ilm_policy_clause`* to specify the policy. Refer to the *ilm_policy_clause* for the full semantics of this clause

Oracle Database assigns a name to the policy of the form `P`*`n`* where *`n`* is an integer value

{ DELETE | ENABLE | DISABLE } POLICY

Specify these clauses to delete a policy for the table, enable a policy for the table, or disable a policy for the table, respectively.

For *`ilm_policy_name`*, specify the name of the policy. You can view policy names by querying the `POLICY_NAME` column of the `DBA_ILMPOLICIES` view.

{ DELETE_ALL, ENABLE_ALL, DISABLE_ALL }

Specify these clauses to delete all policies for the table, enable all policies for the table, or disable all policies for the table, respectively.

See Also:

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

ilm_policy_clause

This clause lets you specify an Automatic Data Optimization policy. You can use the *`ilm_compression_policy`* clause to specify a compression policy, the *`ilm_tiering_policy`* clause to specify a storage tiering policy, or the *`ilm_inmemory_policy`* clause to specify an In-Memory Column Store policy.

ilm_compression_policy

Use this clause to specify a compression policy. This type of policy instructs the database to compress data when a specified condition is met. Use the `SEGMENT`, `GROUP`, or `ROW` clause to specify a segment-level, group-level, or row-level compression policy.

table_compression

Use the *`table_compression`* clause to specify the compression type. This clause applies to segment-level and group-level compression policies.

You must specify a compression type that is higher than the current compression type. The order of compression types from lowest to highest is:

```
NOCOMPRESS
ROW STORE COMPRESS BASIC
ROW STORE COMPRESS ADVANCED
COLUMN STORE COMPRESS FOR QUERY LOW
COLUMN STORE COMPRESS FOR QUERY HIGH
COLUMN STORE COMPRESS FOR ARCHIVE LOW
COLUMN STORE COMPRESS FOR ARCHIVE HIGH
```

Refer to *table_compression* for the full semantics of this clause.

SEGMENT

Specify `SEGMENT` to create a segment-level compression policy. This type of policy instructs the database to compress table segments when the condition specified in the `AFTER` clause is met or when the PL/SQL function specified in the `ON` clause returns `TRUE`.

Note that you cannot modify a segment using `ALTER TABLE`.

GROUP

Specify `GROUP` to create a group-level compression policy. This type of policy instructs the database to compress the table and its dependent objects, such as indexes and SecureFiles LOBs, when the condition specified in the `AFTER` clause is met or when the PL/SQL function specified in the `ON` clause returns `TRUE`.

ROW

Specify `ROW` to create a row-level compression policy. This type of policy instructs the database to compress database blocks in which all the rows have not been modified for a specified period of time. When creating a row-level policy, you must specify `ROW STORE COMPRESS ADVANCED` or `COLUMN STORE COMPRESS FOR QUERY` compression, and you must specify `AFTER` *`ilm_time_period`* `OF NO MODIFICATION`. Refer to *table_compression* for the full semantics of the `ROW STORE COMPRESS ADVANCED` and `COLUMN STORE COMPRESS FOR QUERY` clauses.

AFTER

Use this clause to describe the condition that must be met in order for the policy to take effect. The condition consists of a length of time, specified with the *`ilm_time_period`* clause, and one of the following condition types:

- `OF NO ACCESS`: The policy will take effect after table has not been accessed for the specified length of time.
- `OF NO MODIFICATION`: The policy will take effect after table has not been modified for the specified length of time.

- `OF CREATION`: The policy will take effect when the specified length of time has passed since table was created.

ilm_time_period

Specify a length of time in days, months, or years after which the condition must be met. For *`integer`*, specify a positive integer. The `DAY` and `DAYS` keywords can be used interchangeably and are provided for semantic clarity. This is also the case for the `MONTH` and `MONTHS` keywords, and the `YEAR` and `YEARS` keywords.

ON

Use this clause to specify a PL/SQL function that returns a boolean value. For *`function_name`*, specify the name of the function. The policy will take effect when the function returns `TRUE`.

> **Note:**
>
> The **`ON`** `function_name` clause is not supported for tablespaces.

ilm_tiering_policy

Use this clause to specify a storage tiering policy. This type of policy instructs the database to migrate data to a specified tablespace, either when a specified condition is met or when data usage reaches a specified limit. Use the `SEGMENT` or `GROUP` clause to specify a segment-level or group-level policy. You can migrate data to a read/write tablespace or a read-only tablespace.

TIER TO *tablespace*

Use this clause to migrate data to a read/write *`tablespace`*.

- If you specify the `ON` *`function`* clause, then data will be migrated when function returns `TRUE`. Refer to the ON clause for the full semantics of this clause.
- If you omit the `ON` *`function`* clause, then data will be migrated when data usage of the tablespace quota reaches the percentage defined by `TBS_PERCENT_USED`. The database will make a best effort to migrate enough data so that the amount of free space within the tablespace quota reaches the percentage defined by `TBS_PERCENT_FREE`. Refer to *Oracle Database PL/SQL Packages and Types Reference* for more information on `TBS_PERCENT_USED` and `TBS_PERCENT_FREE`, which are constants in the `DBMS_ILM_ADMIN` package.

TIER TO *tablespace* READ ONLY

Use this clause to migrate data to a read-only tablespace. When migrating data to the tablespace, the database temporarily places the tablespace in read/write mode, migrates the data, and then places the tablespace back in read-only mode.

- If you specify the `AFTER` clause, then data will be migrated when the specified condition is met. Refer to the AFTER clause for the full semantics of this clause
- If you specify the `ON` *`function`* clause, then data will be migrated when function returns `TRUE`. Refer to the ON clause for the full semantics of this clause.

SEGMENT | GROUP

Specify `SEGMENT` to create a segment-level storage tiering policy. This type of policy instructs the database to migrate table segments to *`tablespace`*. Specify `GROUP` to create a group-level storage tiering policy. This type of policy instructs the database to migrate the table and its dependent objects, such as indexes and SecureFiles LOBs, to `tablespace`. The default is `SEGMENT`.

Note:

The **`ON`** `function_name` clause is not supported for tablespaces.

ilm_inmemory_policy

Use this clause to specify an In-Memory Column Store (IM column store) policy. This type of policy instructs the database to enable or disable the table for the IM column store, or to change the compression method for the table in the IM column store, when a specified condition is met.

SET INMEMORY

Use this clause to enable the table for the IM column store when the specified condition is met. You can optionally use the *`inmemory_attributes`* clause to specify how table data will be stored in the IM column store. Refer to *inmemory_attributes* for the full semantics of this clause.

MODIFY INMEMORY

Use this clause to change the compression method for table data stored in the IM column store when the specified condition is met. The table must be enabled for the IM column store.

You must specify a compression method that his higher than the current compression method. The order of compression methods from lowest to highest is:

```
NO INMEMORY
MEMCOMPRESS FOR DML
MEMCOMPRESS FOR QUERY LOW
MEMCOMPRESS FOR QUERY HIGH
MEMCOMPRESS FOR CAPACITY LOW
MEMCOMPRESS FOR CAPACITY HIGH
```

Refer to *inmemory_memcompress* for the full semantics of this clause.

NO INMEMORY

Use this clause to disable the table for the IM column store when the specified condition is met.

SEGMENT

The `SEGMENT` keyword is optional and is provided for semantic clarity. IM column store policies are always segment-level policies.

AFTER | ON

The `AFTER` and `ON` clauses enable you to specify the condition that must be met in order for the IM column store policy to take effect:

- If you specify the `AFTER` clause, then the policy will take effect when the specified condition is met. Refer to the AFTER clause for the full semantics of this clause
- If you specify the `ON` function clause, then the policy will take effect when function returns `TRUE`. Refer to the ON clause for the full semantics of this clause.

Note:

The `ON function_name` clause is not supported for tablespaces.

See Also:

Oracle Database In-Memory Guide for more information on using Automatic Data Optimization policies with the IM column store

supplemental_table_logging

Use the *`supplemental_table_logging`* clause to add or drop a redo log group or one or more supplementally logged columns in a redo log group.

- In the `ADD` clause, use *`supplemental_log_grp_clause`* to create named supplemental log group. Use the *`supplemental_id_key_clause`* to create a system-generated log group.
- On the `DROP` clause, use `GROUP` *`log_group`* syntax to drop a named supplemental log group and use the *`supplemental_id_key_clause`* to drop a system-generated log group.

The *`supplemental_log_grp_clause`* and the *`supplemental_id_key_clause`* have the same semantics in `CREATE TABLE` and `ALTER TABLE` statements. For full information on these clauses, refer to supplemental_log_grp_clause and supplemental_id_key_clause in the documentation on `CREATE TABLE`.

See Also:

Oracle Data Guard Concepts and Administration for information on supplemental redo log groups

allocate_extent_clause

Use the *`allocate_extent_clause`* to explicitly allocate a new extent for the table, the partition or subpartition, the overflow data segment, the LOB data segment, or the LOB index.

Restriction on Allocating Table Extents

You cannot allocate an extent for a temporary table or for a range- or composite-partitioned table.

See Also:

allocate_extent_clause for a full description of this clause and "Allocating Extents: Example"

deallocate_unused_clause

deallocate_unused_clause Use the `deallocate_unused_clause` to explicitly deallocate unused space at the end of the table, partition or subpartition, overflow data segment, LOB data segment, or LOB index and make the space available for other segments in the tablespace.

See Also:

deallocate_unused_clause for a full description of this clause and "Deallocating Unused Space: Example"

rename_lob_storage_clause

Specify this clause to rename a LOB data segment internally.

Specify the following arguments:

- `table_name`
- `column_name` can be of type `BLOB` or `BLOB`
- `old_segment_name` can query `all_lobs`, `all_lob_partition` or `all_lob_subpartition`
- `PARTITION` or `SUBPARTITION` for partitioned tables, NULL for non-partitioned tables
- `new_segment_name`. The rename ensures that the new segment name is unique within the database. A name conflict results in an error.

CACHE | NOCACHE

The `CACHE` and `NOCACHE` clauses have the same semantics in `CREATE TABLE` and `ALTER TABLE` statements. For complete information on these clauses, refer to "CACHE | NOCACHE | CACHE READS" in the documentation on `CREATE TABLE`. If you omit both of these clauses in an `ALTER TABLE` statement, then the existing value is unchanged.

result_cache_clause

The `result_cache_clause` clause has the same semantics in `CREATE TABLE` and `ALTER TABLE` statements. For complete information on this clause, refer to "*result_cache_clause*" in the documentation on `CREATE TABLE`. If you omit this clause in an `ALTER TABLE` statement, then the existing setting is unchanged.

Examples

```
ALTER TABLE employee RESULT_CACHE (MODE DEFAULT)
    ALTER TABLE employee RESULT_CACHE (STANDBY ENABLE)
    ALTER TABLE employee RESULT_CACHE (MODE DEFAULT, STANDBY ENABLE)
    ALTER TABLE employee RESULT_CACHE (STANDBY ENABLE, MODE FORCE)
```

upgrade_table_clause

The `upgrade_table_clause` is relevant for object tables and for relational tables with object columns. It lets you instruct Oracle Database to convert the metadata of the target table to conform with the latest version of each referenced type. If table is already valid, then the table metadata remains unchanged.

Restriction on Upgrading Object Tables and Columns

Within this clause, you cannot specify *object_type_col_properties* as a clause of *column_properties*.

INCLUDING DATA

Specify INCLUDING DATA if you want Oracle Database to convert the data in the table to the latest type version format. You can define the storage for any new column while upgrading the table by using the *column_properties* and the *LOB_partition_storage* . This is the default.

You can convert data in the table at the time you upgrade the type by specifying CASCADE INCLUDING TABLE DATA in the *dependent_handling_clause* of the ALTER TYPE statement. See *Oracle Database PL/SQL Language Reference* for information on this clause. For information on whether a table contains data based on an older type version, refer to the DATA_UPGRADED column of the USER_TAB_COLUMNS data dictionary view.

NOT INCLUDING DATA

Specify NOT INCLUDING DATA if you want Oracle Database to leave column data unchanged.

Restriction on NOT INCLUDING DATA

You cannot specify NOT INCLUDING DATA if the table contains columns in Oracle8 release 8.0.x image format. To determine whether the table contains such columns, refer to the V80_FMT_IMAGE column of the USER_TAB_COLUMNS data dictionary view.

See Also:

- *Oracle Database Reference* for information on the data dictionary views
- ALTER TYPE for information on converting dependent table data when modifying a type upon which the table depends

records_per_block_clause

The *records_per_block_clause* lets you specify whether Oracle Database restricts the number of records that can be stored in a block. This clause ensures that any bitmap indexes subsequently created on the table will be as compressed as possible.

Restrictions on Records in a Block

The *record_per_block_clause* is subject to the following restrictions:

- You cannot specify either MINIMIZE or NOMINIMIZE if a bitmap index has already been defined on table. You must first drop the bitmap index.
- You cannot specify this clause for an index-organized table or a nested table.

MINIMIZE

Specify MINIMIZE to instruct Oracle Database to calculate the largest number of records in any block in the table and to limit future inserts so that no block can contain more than that number of records.

Oracle recommends that a representative set of data already exist in the table before you specify MINIMIZE. If you are using table compression (see *table_compression*), then a representative set of compressed data should already exist in the table.

Restriction on MINIMIZE

You cannot specify `MINIMIZE` for an empty table.

NOMINIMIZE

Specify `NOMINIMIZE` to disable the `MINIMIZE` feature. This is the default.

row_movement_clause

You cannot disable row movement in a reference-partitioned table unless row movement is also disabled in the parent table. Otherwise, this clause has the same semantics in `CREATE TABLE` and `ALTER TABLE` statements. For complete information on these clauses, refer to *row_movement_clause* in the documentation on `CREATE TABLE`.

logical_replication_clause

You can 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. For full semantics see `CREATE TABLE` *logical_replication_clause*::=.

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 the archive name, 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`*.

  You cannot specify `FLASHBACK ARCHIVE` to *change* the flashback archive for this table. Instead you must first issue an `ALTER TABLE` statement with the `NO FLASHBACK ARCHIVE` clause and then issue an `ALTER TABLE` statement with the `FLASHBACK ARCHIVE` clause.
- Specify `NO FLASHBACK ARCHIVE` to disable tracking for the table.

> **See Also:**
>
> The `CREATE TABLE` *flashback_archive_clause* for information on creating a table with tracking enabled and CREATE FLASHBACK ARCHIVE for information on creating default flashback archives

RENAME TO

Use the `RENAME` clause to rename *`table`* to *`new_table_name`*.

Using this clause invalidates any dependent materialized views. For more information on materialized views, see CREATE MATERIALIZED VIEW and *Oracle Database Data Warehousing Guide.*

If a domain index is defined on the table, then the database invokes the ODCIIndexAlter() method with the `RENAME` option. This operation establishes correspondence between the indextype metadata and the base table.

Restriction on Renaming Tables

You cannot rename a sharded table or a duplicated table.

shrink_clause

The shrink clause lets you manually shrink space in a table, index-organized table or its overflow segment, index, partition, subpartition, LOB segment, materialized view, or materialized view log. This clause is valid only for segments in tablespaces with automatic segment management. By default, Oracle Database compacts the segment, adjusts the high water mark, and releases the recuperated space immediately.

Compacting the segment requires row movement. Therefore, you must enable row movement for the object you want to shrink before specifying this clause. Further, if your application has any rowid-based triggers, you should disable them before issuing this clause.

With release 21c, you can use the `shrink_clause` on SecureFile LOB segments. There are two ways to invoke the `shrink_clause`:

1. This command targets a specific LOB column and all its partitions.

   ```
   ALTER TABLE <table_name> MODIFY LOB <lob_column> SHRINK SPACE
   ```

2. This command cascades the shrink operation for all the LOB columns and its partitions for the given table .

   ```
   ALTER TABLE <table_name> SHRINK SPACE CASCADE
   ```

Restrictions:

The `shrink_clause` is not supported on IOT partition tables.

Note:

Do not attempt to enable row movement for an index-organized table before specifying the `shrink_clause`. The `ROWID` of an index-organized table is its primary key, which never changes. Therefore, row movement is neither relevant nor valid for such tables.

COMPACT

If you specify `COMPACT`, then Oracle Database only defragments the segment space and compacts the table rows for subsequent release. The database does not readjust the high water mark and does not release the space immediately. You must issue another `ALTER TABLE` ... `SHRINK SPACE` statement later to complete the operation. This clause is useful if you want to accomplish the shrink operation in two shorter steps rather than one longer step.

For an index or index-organized table, specifying `ALTER` [`INDEX` | `TABLE`] ... `SHRINK SPACE COMPACT` is equivalent to specifying `ALTER` [`INDEX` | `TABLE` ... `COALESCE`. The `shrink_clause` can be cascaded (refer to the `CASCADE` clause, which follows) and compacts the segment more densely than does a coalesce operation, which can improve performance. However, if you do not want to release the unused space, then you can use the appropriate `COALESCE` clause.

CASCADE

If you specify `CASCADE`, then Oracle Database performs the same operations on all dependent objects of `table`, including secondary indexes on index-organized tables.

Restrictions on the *shrink_clause*

The *shrink_clause* is subject to the following restrictions:

- You cannot combine this clause with any other clauses in the same ALTER TABLE statement.

  You cannot specify this clause for a cluster, a clustered table, or any object with a LONG column.
- Segment shrink is not supported for tables with function-based indexes, domain indexes, or bitmap join indexes.
- This clause does not shrink mapping tables of index-organized tables, even if you specify CASCADE.
- You can specify this clause for a table with Advanced Row Compression enabled (ROW STORE COMPRESS ADVANCED). You cannot specify this clause for a table with any other type of table compression enabled.
- You cannot shrink a table that is the master table of an ON COMMIT materialized view. Rowid materialized views must be rebuilt after the shrink operation.

READ ONLY | READ WRITE

Specify READ ONLY to put the table in read-only mode. When the table is in READ ONLY mode, you cannot issue any DML statements that affect the table or any SELECT ... FOR UPDATE statements. You can issue DDL statements as long as they do not modify any table data. Operations on indexes associated with the table are allowed when the table is in READ ONLY mode. See *Oracle Database Administrator's Guide* for the complete list of operations that are allowed and disallowed on read-only tables.

Specify READ WRITE to return a read-only table to read/write mode.

REKEY *encryption_spec*

Use the REKEY clause to generate a new encryption key or to switch between different algorithms. This operation returns only after all encrypted columns in the table, including LOB columns, have been reencrypted.

DEFAULT COLLATION

This clause lets you change the default collation for the table. For *collation_name*, specify a valid named collation or pseudo-collation.

The new default collation for the table is assigned to columns of a character data type that are subsequently added to the table with an ALTER TABLE ADD statement or modified from a non-character data type with an ALTER TABLE MODIFY statement. The collations for existing columns in the table are not changed. Refer to the DEFAULT COLLATION clause of CREATE TABLE for the full semantics of this clause.

[NO] ROW ARCHIVAL

Specify this clause to enable or disable *table* for row archival.

- Specify ROW ARCHIVAL to enable *table* for row archival. A hidden column ORA_ARCHIVE_STATE is created in the table. If the table is already populated with data, then the value of ORA_ARCHIVE_STATE is set to 0 for each existing row in the table. You can subsequently use the UPDATE statement to set the value of ORA_ARCHIVE_STATE to 1 for rows you want to archive.

- Specify `NO ROW ARCHIVAL` to disable *`table`* for row archival. The hidden column `ORA_ARCHIVE_STATE` is dropped from the table.

Restrictions on [NO] ROW ARCHIVAL

The following restrictions apply to this clause:

- You cannot specify the `ROW ARCHIVAL` clause for a table that already contains a column named `ORA_ARCHIVE_STATE`.
- You cannot specify the `NO ROW ARCHIVAL` clause for tables owned by `SYS`.

See Also:

- The `CREATE TABLE` ROW ARCHIVAL clause for the full semantics of this clause
- *Oracle Database VLDB and Partitioning Guide* for more information on In-Database Archiving

attribute_clustering_clause

Use the `ADD` *`attribute_clustering_clause`* to enable the table for attribute clustering. The *`attribute_clustering_clause`* has the same semantics for `ALTER TABLE` and `CREATE TABLE`. Refer to the *attribute_clustering_clause* in the documentation on `CREATE TABLE`.

MODIFY CLUSTERING

Use this clause to allow or disallow attribute clustering for the table during direct-path insert operations or data movement operations. The table must be enabled for attribute clustering. The *`clustering_when`* clause and the *`zonemap_clause`* have the same semantics for `ALTER TABLE` and `CREATE TABLE`. Refer to the clustering_when clause and the zonemap_clause in the documentation on `CREATE TABLE`.

DROP CLUSTERING

Use this clause to disable the table for attribute clustering.

If a zone map on the table was created using the `WITH MATERIALIZED ZONEMAP` clause of `CREATE TABLE` or `ALTER TABLE`, then the zone map will be dropped. If a zone map on the table was created using the `CREATE MATERIALIZED ZONEMAP` statement, then the zone map will not be dropped.

FOR STAGING

You can change the staging property of an exisiting table with `ALTER TABLE t FOR STAGING`. The staging table `t` now has all the characteristics of a staging table created with `CREATE TABLE t FOR STAGING`.

Refer to `CREATE TABLE` clause for the full semantics of staging tables: FOR STAGING

ALTER TABLE t NOT FOR STAGING

You can change a staging table with `ALTER TABLE t NOT FOR STAGING`. This means:

- You can enable compression explicitly on the table, its partitions, and future data loads.
- You can gather statistics on the table explictly or by using an statistics gathering application.

- When you drop the table, it can be put in the recyclebin.

alter_iot_clauses

index_org_table_clause

This clause lets you alter some of the characteristics of an existing index-organized table. Index-organized tables keep data sorted on the primary key and are therefore best suited for primary-key-based access and manipulation. See *index_org_table_clause* in the context of `CREATE TABLE`.

See Also:

"Modifying Index-Organized Tables: Examples"

prefix_compression

Use the *`prefix_compression`* clause to enable prefix compression for the table. Specify `COMPRESS` to instruct Oracle Database to combine the primary key index blocks of the index-organized table where possible to free blocks for reuse. You can specify this clause with the *`parallel_clause`*. Specify `NOCOMPRESS` to disable prefix compression for the table.

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.

PCTTHRESHOLD *integer*

Refer to "PCTTHRESHOLD *integer*" in the documentation on `CREATE TABLE`.

INCLUDING *column_name*

Refer to "INCLUDING column_name" in the documentation on `CREATE TABLE`.

overflow_attributes

The *`overflow_attributes`* let you specify the overflow data segment physical storage and logging attributes to be modified for the index-organized table. Parameter values specified in this clause apply only to the overflow data segment.

See Also:

CREATE TABLE

add_overflow_clause

The *`add_overflow_clause`* lets you add an overflow data segment to the specified index-organized table. You can also use this clause to explicitly allocate an extent to or deallocate unused space from an existing overflow segment.

Use the `STORE IN` *`tablespace`* clause to specify tablespace storage for the entire overflow segment. Use the `PARTITION` clause to specify tablespace storage for the segment by partition.

For a partitioned index-organized table:

- If you do not specify `PARTITION`, then Oracle Database automatically allocates an overflow segment for each partition. The physical attributes of these segments are inherited from the table level.
- If you want to specify separate physical attributes for one or more partitions, then you must specify such attributes for *every* partition in the table. You need not specify the name of the partitions, but you must specify their attributes in the order in which they were created.

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 `TABLESPACE` for a particular partition, then the database uses the tablespace specified for the table. If you do not specify `TABLESPACE` at the table level, then the database uses the tablespace of the partition primary key index segment.

Restrictions on Overflow Attributes

Within the *`segment_attributes_clause`*:

- You cannot specify the `OPTIMAL` parameter of the *`physical_attributes_clause`*.
- You cannot specify tablespace storage for the overflow segment using this clause. For a nonpartitioned table, you can use `ALTER TABLE` ... `MOVE` ... `OVERFLOW` to move the segment to a different tablespace. For a partitioned table, use `ALTER TABLE` ... `MODIFY DEFAULT ATTRIBUTES` ... `OVERFLOW` to change the default tablespace of the overflow segment.

Additional restrictions apply if *`table`* is in a locally managed tablespace, because in such tablespaces several segment attributes are managed automatically by the database.

See Also:

allocate_extent_clause and *deallocate_unused_clause* for full descriptions of these clauses of the *`add_overflow_clause`*

alter_overflow_clause

The *`alter_overflow_clause`* lets you change the definition of the overflow segment of an existing index-organized table.

The restrictions that apply to the *`add_overflow_clause`* also apply to the *`alter_overflow_clause`*.

Note:

When you add a column to 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 `ALTER TABLE` statement. This checking function guarantees that subsequent DML operations on the index-organized table will not fail because an overflow segment is lacking.

alter_mapping_table_clauses

The *`alter_mapping_table_clauses`* is valid only if *`table`* is index organized and has a mapping table.

allocate_extent_clause

Use the *`allocate_extent_clause`* to allocate a new extent at the end of the mapping table for the index-organized table. Refer to *allocate_extent_clause* for a full description of this clause.

deallocate_unused_clause

Specify the *`deallocate_unused_clause`* to deallocate unused space at the end of the mapping table of the index-organized table. Refer to *deallocate_unused_clause* for a full description of this clause.

Oracle Database automatically maintains all other attributes of the mapping table or its partitions.

COALESCE Clause

Specify `COALESCE` to instruct Oracle Database to merge the contents of index blocks of the index the database uses to maintain the index-organized table where possible to free blocks for reuse. Refer to the *shrink_clause* for information on the relationship between these two clauses.

alter_XMLSchema_clause

This clause is valid as part of *`alter_table_properties`* only if you are modifying an `XMLType` table with `BINARY XML` storage. Refer to XMLSchema_spec in the documentation on `CREATE TABLE` for more information on the `ALLOW` and `DISALLOW` clauses.

column_clauses

Use these clauses to add, drop, or otherwise modify a column.

add_column_clause

The *`add_column_clause`* lets you add a column to a table.

See Also:

CREATE TABLE for a description of the keywords and parameters of this clause and "Adding a Table Column: Example"

Use `ALTER TABLE ADD` to associate columns to a domain:

```
ALTER TABLE  [owner.]name ADD (<column_list_def_clause> [, DOMAIN
[domain_owner.]domain_name (<column_name_list>)]+)
```

column_definition

Unless otherwise noted in this section, the elements of *`column_definition`* have the same behavior when adding a column to an existing table as they do when creating a new table. Refer to the*column_definition* clause of `CREATE TABLE` for information.

Restriction on *column_definition*

The `SORT` parameter is valid only when creating a new table. You cannot specify `SORT` in the column_definition of an `ALTER TABLE` ... `ADD` statement.

When you add a column, the initial value of each row for the new column is null, unless you specify the `DEFAULT` clause.

You can add an overflow data segment to each partition of a partitioned index-organized table.

You can add LOB columns to nonpartitioned and partitioned tables. You can specify LOB storage at the table and at the partition or subpartition level.

If you previously created a view with a query that used the `SELECT *` syntax to select all columns from table, and you now add a column to *`table`*, then the database does not automatically add the new column to the view. To add the new column to the view, re-create the view using the `CREATE VIEW` statement with the `OR REPLACE` clause. Refer to CREATE VIEW for more information.

Restrictions on Adding Columns

The addition of columns is subject to the following restrictions:

- You cannot add a LOB column or an `INVISIBLE` column to a cluster table.
- If you add a LOB column to a hash-partitioned table, then the only attribute you can specify for the new partition is `TABLESPACE`.
- You cannot add a column with a `NOT NULL` constraint if *`table`* has any rows unless you also specify the `DEFAULT` clause.
- If you specify this clause for an index-organized table, then you cannot specify any other clauses in the same statement.
- You cannot add a column to a duplicated table.

DEFAULT

Use the `DEFAULT` clause to specify a default for a new column or a new default for an existing column. Oracle Database assigns this value 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 are adding a new column to a table, then the order in which NEXTVAL is assigned to each existing row is nondeterministic. 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 ALTER TABLE statement. For example, if user MARY adds a column to 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.

If you specify the DEFAULT clause for a column, then the default value is stored as metadata but the column itself is not populated with data. However, subsequent queries that specify the new column are rewritten so that the default value is returned in the result set. This optimized behavior is subject to the following restrictions:

- It cannot be index-organized, temporary, or part of a cluster. It also cannot be a queue table, an object table, or the container table of a materialized view.
- If the table has a Virtual Private Database (VPD) policy on it, then the optimized behavior will not take place unless the user who issues the ALTER TABLE ... ADD statement has the EXEMPT ACCESS POLICY system privilege.
- The column being added cannot be encrypted, and cannot be an object column, nested table column, or a LOB column.
- The DEFAULT expression cannot include the sequence pseudocolumns CURRVAL or NEXTVAL.

If the optimized behavior cannot take place due to the preceding restrictions, then Oracle Database updates each row in the newly created column with the default value. In this case, the database does not fire any UPDATE triggers that are defined on the table.

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.

ON NULL

If you specify the ON NULL clause, then Oracle Database assigns the DEFAULT column value when a subsequent INSERT or optionally an UPDATE 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.

Refer to CREATE TABLE ON NULL for the full semantics of DEFAULT ON NULL.

See Also:

"Specifying a Default Column Value: Examples"

identity_clause

The *identity_clause* has the same semantics when you add an identity column that it has when you create an identity column. Refer to CREATE TABLE *identity_clause* for more information.

When you add a new identity column to a table, all existing rows are updated using the sequence generator. The order in which a value is assigned to each existing row is nondeterministic.

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.

inline_constraint

Use *inline_constraint* to add a constraint to the new column.

inline_ref_constraint

This clause lets you describe a new column of type REF. Refer to *constraint* for syntax and description of this type of constraint, including restrictions.

virtual_column_definition

The *virtual_column_definition* has the same semantics when you add a column that it has when you create a column.

See Also:

The CREATE TABLE *virtual_column_definition* and "Adding a Virtual Table Column: Example" for more information

Restriction on Adding a Virtual Column

You cannot add a virtual column when the SQL expression for the virtual column involves a column on which an Oracle Data Redaction policy is defined.

column_properties

The clauses of *column_properties* determine the storage characteristics of an object type, nested table, varray, or LOB column.

object_type_col_properties

This clause is valid only when you are adding a new object type column or attribute. To modify the properties of an existing object type column, use the *modify_column_clauses*. The semantics of this clause are the same as for CREATE TABLE unless otherwise noted.

Use the *object_type_col_properties* clause to specify storage characteristics for a new object column or attribute or an element of a collection column or attribute.

For complete information on this clause, refer to object_type_col_properties in the documentation on CREATE TABLE.

nested_table_col_properties

The *nested_table_col_properties* clause lets you specify separate storage characteristics for a nested table, which in turn lets you to define the nested table as an index-organized table. You must include this clause when creating a table with columns or column attributes whose type is a nested table. (Clauses within this clause that function the same way they function for parent object tables are not repeated here. See the CREATE TABLE clause *nested_table_col_properties* for more information about these clauses.)

- For *nested_item*, specify the name of a column (or a top-level attribute of the nested table object type) whose type is a nested table.

  If the nested table is a multilevel collection, and the inner nested table does not have a name, then specify COLUMN_VALUE in place of the *nested_item* name.
- For *storage_table*, specify the name of the table where the rows of *nested_item* reside. The storage table is created in the same schema and the same tablespace as the parent table.

Restrictions on Nested Table Column Properties

Nested table column properties are subject to the following restrictions:

- You cannot specify the *parallel_clause*.
- You cannot specify CLUSTER as part of the *physical_properties* clause.

See Also:

"Nested Tables: Examples"

varray_col_properties

The *varray_col_properties* clause lets you specify separate storage characteristics for the LOB in which a varray will be stored. If you specify this clause, then Oracle Database will always store the varray in a LOB, even if it is small enough to be stored inline. 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.

Restriction on Varray Column Properties

You cannot specify TABLESPACE as part of *LOB_parameters* for a varray column. The LOB tablespace for a varray defaults to the tablespace of the containing table.

out_of_line_part_storage

This clause lets you specify storage attributes the newly added column for each partition or subpartition in a partitioned table. For any partition or subpartition you do not name in this clause, the storage attributes for the new column are the same as those specified in the `nested_table_col_properties` at the table level.

LOB_storage_clause

Use the *`LOB_storage_clause`* to specify the LOB storage characteristics for a newly added LOB column, LOB partition, or LOB subpartition, or when you are converting a `LONG` column into a LOB column. You cannot use this clause to modify an existing LOB. Instead, you must use the *modify_LOB_storage_clause*.

Unless otherwise noted in this section, all LOB parameters, in both the *`LOB_storage_clause`* and the *`modify_LOB_storage_clause`*, have the same semantics in an `ALTER TABLE` statement that they have in a `CREATE TABLE` statement. Refer to the `CREATE TABLE` *LOB_storage_clause* for complete information on this clause.

Restriction on LOB Parameters

The only parameter of *`LOB_parameters`* you can specify for a hash partition or hash subpartition is `TABLESPACE`.

CACHE READS Clause

When you add a new LOB column, you can specify the logging attribute with `CACHE READS`, as you can when defining a LOB column at create time. Refer to the `CREATE TABLE` clause CACHE READS for full information on this clause.

ENABLE | DISABLE STORAGE IN ROW

You cannot change `STORAGE IN ROW` once it is set. Therefore, you cannot specify this clause as part of the *`modify_col_properties`* clause. However, you can change this setting when adding a new column (*add_column_clause*) or when moving the table (*move_table_clause*). Refer to the `CREATE TABLE` clause ENABLE STORAGE IN ROW for complete information on this clause.

CHUNK *integer*

You cannot use the *`modify_col_properties`* clause to change the value of `CHUNK` after it has been set. If you require a different `CHUNK` value for a column after it has been created, use `ALTER TABLE ... MOVE`. Refer to the `CREATE TABLE` clause CHUNK integer for more information.

RETENTION

For BasicFiles LOBs, if the database is in automatic undo mode, then you can specify `RETENTION` instead of `PCTVERSION` to instruct Oracle Database to retain old versions of this LOB. This clause overrides any prior setting of `PCTVERSION`. Refer to the `CREATE TABLE` clause LOB_retention_clause for a full description of this parameter.

FREEPOOLS *integer*

For BasicFiles LOBs, if the database is in automatic undo mode, then you can use this clause to specify the number of freelist groups for this LOB. This clause overrides any prior setting of `FREELIST GROUPS`. Refer to the `CREATE TABLE` clause FREEPOOLS integer for a full description of this parameter. The database ignores this parameter for SecureFiles LOBs.

LOB_partition_storage

You can specify only one list of *`LOB_partition_storage`* clauses in a single `ALTER TABLE` statement, and all *`LOB_storage_clauses`* and *`varray_col_properties`* clause must precede the list of *`LOB_partition_storage`* clauses. Refer to the `CREATE TABLE` clause LOB_partition_storage for full information on this clause, including restrictions.

XMLType_column_properties

Refer to the `CREATE TABLE` clause *XMLType_column_properties* for a full description of this clause.

See Also:

- *LOB_storage_clause* for information on the *`LOB_segname`* and *`LOB_parameters`* clauses
- "XMLType Column Examples" for an example 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 an XMLSchema

XMLType_storage

Refer to the `CREATE TABLE` clause *XMLType_storage*.

JSON_storage_clause

With 21c you can define a column of `JSON` data type using the *`JSON_storage_clause`*.

Example

```
ALTER TABLE t ADD (jcol JSON)
```

modify_column_clauses

Use the *`modify_column_clauses`* to modify the properties of an existing column, the visibility of an existing column, or the substitutability of an existing object type column.

See Also:

"Modifying Table Columns: Examples"

modify_col_properties

Use this clause to modify the properties of the column. Any of the optional parts of the column definition (data type, default value, or constraint) that you omit from this clause remain unchanged.

datatype

You can change the data type of any column if all rows of the column contain nulls. However, if you change the data type of a column in a materialized view container table, then Oracle Database invalidates the corresponding materialized view.

You can omit the data type only if the statement also designates the column as part of the foreign key of a referential integrity constraint. The database automatically assigns the column the same data type as the corresponding column of the referenced key of the referential integrity constraint.

You can always increase the size of a character or raw column or the precision of a numeric column, whether or not all the rows contain nulls. You can reduce the size of a data type of a column as long as the change does not require data to be modified. The database scans existing data and returns an error if data exists that exceeds the new length limit.

When you increase the size of a `VARCHAR2`, `NVARCHAR2`, or `RAW` column to exceed 4000 bytes, Oracle Database performs an in-place length extension and does not migrate the inline storage to external LOB storage. This enables uninterrupted migration of large tables, especially after migration, to leverage extended data types. However, the inline storage of the column will not be preserved during table reorganization operations, such as `CREATE TABLE` ... `AS SELECT`, export, import, or online redefinition. To migrate to the new out-of-line storage of extended data type columns, you must recreate the table using one of the aforementioned methods. The inline storage of the column will be preserved during table or partition movement operations, such as `ALTER TABLE MOVE [[SUB]PARTITION]`, and partition maintenance operations, such as `ALTER TABLE SPLIT [SUB]PARTITION`, `ALTER TABLE MERGE [SUB]PARTITIONS`, and `ALTER TABLE COALESCE [SUB]PARTITIONS`.

Note:

Oracle recommends against excessively increasing the size of a `VARCHAR2`, `NVARCHAR2`, or `RAW` column beyond 4000 bytes for the following reasons:

- Row chaining may occur.
- Data that is stored inline must be read in its entirety, whether a column is selected or not. Therefore, extended data type columns that are stored inline can have a negative impact on performance.

You can reduce the size of a data type of a column as long as the change does not require data to be modified. The database scans existing data and returns an error if data exists that exceeds the new length limit.

You can change a `DATE` column to a `TIMESTAMP` or `TIMESTAMP WITH LOCAL TIME ZONE` column, and you can change a `TIMESTAMP` or `TIMESTAMP WITH LOCAL TIME ZONE` column to a `DATE` column. The following rules apply:

- When you change a `TIMESTAMP` or `TIMESTAMP WITH LOCAL TIME ZONE` column to a `DATE` column, Oracle Database updates each column value that has nonzero fractional seconds by rounding the value to the nearest second. If, while updating such a value, Oracle Database encounters a minute field greater than or equal to 60 (which can occur in a boundary case when the daylight saving rule switches), then it updates the minute field by subtracting 60 from it.
- After you change a `TIMESTAMP WITH LOCAL TIME ZONE` column to a `DATE` column, the values in the column still represent the local time that they represented in the database time zone. However, the database time zone is no longer associated with the values. When queried in SQL*Plus, the values are no longer automatically adjusted to the session time zone. It is

now the responsibility of applications processing the column values to interpret them in a particular time zone.

If the table is empty, then you can increase or decrease the leading field or the fractional second value of a datetime or interval column. If the table is not empty, then you can only increase the leading field or fractional second of a datetime or interval column.

You can use the `TO_LOB` function to change a `LONG` column to a `CLOB` or `NCLOB` column, and a `LONG RAW` column to a `BLOB` column. However, you cannot use the `TO_LOB` function from within a PL/SQL package. Instead use the `TO_CLOB` (character) or `TO_BLOB` (raw) functions.

- The modified LOB column inherits all constraints and triggers that were defined on the original `LONG` column. If you want to change any constraints, then you must do so in a subsequent `ALTER TABLE` statement.
- If any domain indexes are defined on the `LONG` column, then you must drop them before modifying the column to a LOB.
- After the modification, you will have to rebuild all other indexes on all columns of the table.

You can use the `TO_CLOB` (character) function to convert `NCLOB` columns `CLOB` columns.

See Also:

- *Oracle Database SecureFiles and Large Objects Developer's Guide* for information on `LONG` to LOB migration
- ALTER INDEX for information on dropping and rebuilding indexes

For `CHAR` and `VARCHAR2` columns, you can change the length semantics by specifying `CHAR` (to indicate character semantics for a column that was originally specified in bytes) or `BYTE` (to indicate byte semantics for a column that was originally specified in characters). To learn the length semantics of existing columns, query the `CHAR_USED` column of the `ALL_`, `USER_`, or `DBA_TAB_COLUMNS` data dictionary view.

See Also:

- *Oracle Database Globalization Support Guide* for information on byte and character semantics
- *Oracle Database Reference* for information on the data dictionary views

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

Use this clause to associate *`domain_name`* with the column. The domain's data type must be compatible with the column's data type.

COLLATE

Use this clause to set or change the data-bound collation for a column. For *column_collation_name*, specify a valid named collation or pseudo-collation. Refer to the DEFAULT COLLATION clause of CREATE TABLE for more information on data-bound collations.

Restrictions on Changing Column Collation

The modification of the column collation is subject to the following restrictions:

- If the column belongs to an index key, then its collation can only be changed:
  - among collations: BINARY, USING_NLS_COMP, USING_NLS_SORT, and USING_NLS_SORT_CS
  - between collations BINARY_CI and USING_NLS_SORT_CI
  - between collations BINARY_AI and USING_NLS_SORT_AI
- If the column belongs to a range- or list-partitioning key, is referenced by a bitmap join index, belongs to the primary key of an index-organized table, or to the key of a domain index, including an Oracle Text index, then its collation can only be changed among the collations BINARY, USING_NLS_COMP, USING_NLS_SORT, and USING_NLS_SORT_CS.
- If the column belongs to an attribute clustering key, then its collation can only be changed between the collations BINARY and USING_NLS_COMP.

See Also:

Modifying the Collation of a Column for Fine-Grained Case-Insensitivity: Example

identity_clause

Use *identity_clause* to modify the properties of an identity column. You cannot specify this clause on a column that is not an identity column. If you do not specify ALWAYS or BY DEFAULT, then the current generation type is retained. Refer to CREATE TABLE *identity_clause* for more information on ALWAYS and BY DEFAULT.

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 exceptions are:

- START WITH LIMIT VALUE, which is specific to *identity_options*, can only be used with ALTER TABLE MODIFY. If you specify START WITH LIMIT VALUE, then Oracle Database locks the table and finds the maximum identity column value in the table (for increasing sequences) or the minimum identity column value (for decreasing sequences) and assigns the value as the sequence generator's high water mark. The next value returned by the sequence generator will be the high water mark + INCREMENT BY *integer* for increasing sequences, or the high water mark - INCREMENT BY *integer* for decreasing sequences.
- If you change the value of START WITH, then the default values will be used for all other parameters in this clause unless you specify otherwise.

DROP IDENTITY

Use this clause to remove the identity property from a column, including the sequence generator and NOT NULL and NOT DEFERRABLE constraints. Identity column values in existing rows are not affected.

ENCRYPT *encryption_spec* | DECRYPT

Use this clause to decrypt an encrypted column, to encrypt an unencrypted column, or to change the integrity algorithm or the SALT option of an encrypted column.

When encrypting an existing column, if you specify *encryption_spec*, it must match the encryption specification of any other encrypted columns in the same table. Refer to the CREATE TABLE clause *encryption_spec* for additional information and restrictions on the *encryption_spec*.

If a materialized view log is defined on the table, then Oracle Database encrypts or decrypts in the materialized view log any columns you encrypt or decrypt in this clause.

Restrictions on ENCRYPT *encryption_spec* | DECRYPT

This clause is subject to the following restrictions:

- If the new or existing column is a LOB column, then it must be stored as a SecureFiles LOB, and you cannot specify the SALT option.
- You cannot encrypt or decrypt a column on which a fine-grained audit policy for the UPDATE statement is enabled. However, you can disable the fine-grained audit policy, encrypt or decrypt the column, and then enable the fine-grained audit policy.

See Also:

"Data Encryption: Examples"

inline_constraint

This clause lets you add a constraint to a column you are modifying. To change the state of existing constraints on existing columns, use the *constraint_clauses*.

LOB_storage_clause

The *LOB_storage_clause* is permitted within *modify_col_properties* only if you are converting a LONG column to a LOB column. In this case only, you can specify LOB storage for the column using the *LOB_storage_clause*. However, you can specify only the single column as a *LOB_item*. Default LOB storage attributes are used for any attributes you omit in the *LOB_storage_clause*.

alter_XMLSchema_clause

This clause is valid within *modify_col_properties* only for XMLType tables with BINARY XML storage. Refer to XMLSchema_spec in the documentation on CREATE TABLE for more information on the ALLOW and DISALLOW clauses.

Restrictions on Modifying Column Properties

The modification of column properties is subject to the following restrictions:

- You cannot change the data type of a LOB column.
- You cannot modify a column of a table if a domain index is defined on the column. You must first drop the domain index and then modify the column.
- You cannot modify the data type or length of a column that is part of the partitioning or subpartitioning key of a table or index.

- You can change a CHAR column to VARCHAR2 (or VARCHAR) and a VARCHAR2 (or VARCHAR) column to CHAR only if the BLANK_TRIMMING initialization parameter is set to TRUE and the column size stays the same or increases. If the BLANK_TRIMMING initialization parameter is set to TRUE, then you can also reduce the column size to any size greater than or equal to the maximum trimmed data value.
- You cannot change a LONG or LONG RAW column to a LOB if the table is part of a cluster. If you do change a LONG or LONG RAW column to a LOB, then the only other clauses you can specify in this ALTER TABLE statement are the DEFAULT clause and the *LOB_storage_clause*.
- You can specify the *LOB_storage_clause* as part of *modify_col_properties* only when you are changing a LONG or LONG RAW column to a LOB.
- You cannot specify a column of data type ROWID for an index-organized table, but you can specify a column of type UROWID.
- You cannot change the data type of a column to REF.
- You cannot modify the properties of a column in a duplicated table.

See Also:

ALTER MATERIALIZED VIEW for information on revalidating a materialized view

modify_virtcol_properties

This clause lets you modify a virtual column in the following ways:

- Specify the COLLATE clause to set or change the data-bound collation for a virtual column. For *column_collation_name*, specify a valid named collation or pseudo-collation. Refer to the DEFAULT COLLATION clause of CREATE TABLE for more information on data-bound collations.
- If the virtual column refers to an editioned PL/SQL function, then you can modify the evaluation edition or the unusable editions for a virtual column. The *evaluation_edition_clause* and the *unusable_editions_clause* have the same semantics when you modify a virtual column that they have when you create a virtual column. For complete information, refer to evaluation_edition_clause and unusable_editions_clause in the documentation on CREATE TABLE.

Restrictions on Modifying Virtual Columns

The following restrictions apply to modifying virtual columns:

- Specifying the COLLATE clause to set or change the data-bound collation for a virtual column is subject to the restrictions listed in Restrictions on Changing Column Collation.
- If an index is defined on a virtual column and you modify its evaluation edition or unusable editions, then the database will invalidate all indexes on the virtual column. If you attempt to modify any other properties of the virtual column, then an error occurs.

modify_col_visibility

Use this clause to change the visibility of *column*. For complete information, refer to "VISIBLE | INVISIBLE" in the documentation on CREATE TABLE.

Restriction on Modifying Column Visibility

You cannot change a `VISIBLE` column to `INVISIBLE` in a table owned by `SYS`.

modify_col_substitutable

Use this clause to set or change the substitutability of an existing object type column.

The `FORCE` keyword drops any hidden columns containing typeid information or data for subtype attributes. You must specify `FORCE` if the column or any attributes of its type are not `FINAL`.

Restrictions on Modifying Column Substitutability

The modification of column substitutability is subject to the following restrictions:

- You can specify this clause only once in any `ALTER TABLE` statement.
- You cannot modify the substitutability of a column in an object table if the substitutability of the table itself has been set.
- You cannot specify this clause if the column was created or added using the `IS OF TYPE` syntax, which limits the range of subtypes permitted in an object column or attribute to a particular subtype. Refer to substitutable_column_clause in the documentation on `CREATE TABLE` for information on the `IS OF TYPE` syntax.
- You cannot change a varray column to `NOT SUBSTITUTABLE`, even by specifying `FORCE`, if any of its attributes are nested object types that are not `FINAL`.

***modify_domain*::=**

Use this clause to add or remove a domain from the table.

ADD DOMAIN

Use this to add a domain to the listed columns. You must specify as many columns as the domain. The first column is associated with the first domain column, second column is associated with the second domain column, and so on.

Example: Create a Domain phone_number and Add a Column

```
CREATE DOMAIN phone_number as VARCHAR2(12)
  CONSTRAINT CHECK (phone_number not like '%[0-9]%')
  NOT NULL;
```

To add a new column to a table with domain, the `ALTER TABLE` statement can be used as follows:

```
ALTER TABLE customers ADD (cust_cell_phone_number Varchar2(12) DOMAIN phone_number);

ALTER TABLE customers ADD (cust_cell_phone_number Varchar2(12) DOMAIN phone_number
DEFAULT ON NULL '650-000-0000');
```

DROP DOMAIN

Use this clause to dissociate a domain from the listed columns. The number of columns in this statement must match the number of columns in the domain.

If the domain has a collation, this will be preserved.

PRESERVE CONSTRAINTS

By default, any constraints from the domain are removed from the column. Use this clause to copy the domain constraints to the column.

An error is raised in the following cases:

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

drop_column_clause

The *`drop_column_clause`* lets you free space in the database by dropping columns you no longer need or by marking them to be dropped at a future time when the demand on system resources is less.

- If you drop a nested table column, then its storage table is removed.
- If you drop a LOB column, then the LOB data and its corresponding LOB index segment are removed.
- If you drop a `BFILE` column, then only the locators stored in that column are removed, not the files referenced by the locators.
- If you drop or mark unused a column defined as an `INCLUDING` column, then the column stored immediately before this column will become the new `INCLUDING` column.

SET UNUSED Clause

Specify `SET UNUSED` to mark one or more columns as unused. For an internal heap-organized table, specifying this clause does not actually remove the target columns from each row in the table. It does not restore the disk space used by these columns. Therefore, the response time is faster than when you execute the `DROP` clause.

When you specify this clause for a column in an external table, the clause is transparently converted to an `ALTER TABLE` ... `DROP COLUMN` statement. The reason for this is that any operation on an external table is a metadata-only operation, so there is no difference in the performance of the two commands.

You can view all tables with columns marked `UNUSED` in the data dictionary views `USER_UNUSED_COL_TABS`, `DBA_UNUSED_COL_TABS`, and `ALL_UNUSED_COL_TABS`.

See Also:

Oracle Database Reference for information on the data dictionary views

Unused columns are treated as if they were dropped, even though their column data remains in the table rows. After a column has been marked `UNUSED`, you have no access to that column. A `SELECT *` query will not retrieve data from unused columns. In addition, the names and types of columns marked `UNUSED` will not be displayed during a `DESCRIBE`, and you can add to the table a new column with the same name as an unused column.

Note:

Until you actually drop these columns, they continue to count toward the maximum number of columns in a single table, i.e. 1000 if the `MAX_COLUMNS` initialization parameter is set to `STANDARD`, or 4096 columns if `MAX_COLUMNS` is set to `EXTENDED`. However, as with all DDL statements, you cannot roll back the results of this clause. You cannot issue `SET USED` counterpart to retrieve a column that you have `SET UNUSED`. Refer to CREATE TABLE for more information on the 1000-column limit.

Also, if you mark a `LONG` column as `UNUSED`, then you cannot add another `LONG` column to the table until you actually drop the unused `LONG` column.

ONLINE

Specify `ONLINE` to indicate that DML operations on the table will be allowed while marking the column or columns `UNUSED`.

Restrictions on Marking Columns Unused

The following restrictions apply to the `SET UNUSED` clause:

- You cannot specify the `ONLINE` clause when marking a column with a `DEFERRABLE` constraint as `UNUSED`.
- Columns in tables owned by `SYS` cannot be marked as `UNUSED`.

DROP Clause

Specify `DROP` to remove the column descriptor and the data associated with the target column from each row in the table. If you explicitly drop a particular column, then all columns currently marked `UNUSED` in the target table are dropped at the same time.

When the column data is dropped:

- All indexes defined on any of the target columns are also dropped.
- All constraints that reference a target column are removed.
- If any statistics types are associated with the target columns, then Oracle Database disassociates the statistics from the column with the `FORCE` option and drops any statistics collected using the statistics type.

Note:

If the target column is a parent key of a nontarget column, or if a check constraint references both the target and nontarget columns, then Oracle Database returns an error and does not drop the column unless you have specified the `CASCADE CONSTRAINTS` clause. If you have specified that clause, then the database removes all constraints that reference any of the target columns.

See Also:

DISASSOCIATE STATISTICS for more information on disassociating statistics types

DROP UNUSED COLUMNS Clause

Specify `DROP UNUSED COLUMNS` to remove from the table all columns currently marked as unused. Use this statement when you want to reclaim the extra disk space from unused columns in the table. If the table contains no unused columns, then the statement returns with no errors.

column

Specify one or more columns to be set as unused or dropped. Use the `COLUMN` keyword only if you are specifying only one column. If you specify a column list, then it cannot contain duplicates.

CASCADE CONSTRAINTS

Specify `CASCADE CONSTRAINTS` if you want to drop all foreign key constraints that refer to the primary and unique keys defined on the dropped columns as well as all multicolumn constraints defined on the dropped columns. If any constraint is referenced by columns from other tables or remaining columns in the target table, then you must specify `CASCADE CONSTRAINTS`. Otherwise, the statement aborts and an error is returned.

INVALIDATE

The `INVALIDATE` keyword is optional. Oracle Database automatically invalidates all dependent objects, such as views, triggers, and stored program units. Object invalidation is a recursive process. Therefore, all directly dependent and indirectly dependent objects are invalidated. However, only local dependencies are invalidated, because the database manages remote dependencies differently from local dependencies.

An object invalidated by this statement is automatically revalidated when next referenced. You must then correct any errors that exist in that object before referencing it.

See Also:

Oracle Database Concepts for more information on dependencies

CHECKPOINT

Specify `CHECKPOINT` if you want Oracle Database to apply a checkpoint for the `DROP COLUMN` operation after processing *`integer`* rows; *`integer`* is optional and must be greater than zero. If *`integer`* is greater than the number of rows in the table, then the database applies a checkpoint after all the rows have been processed. If you do not specify *`integer`*, then the database sets the default of 512. Checkpointing cuts down the amount of undo logs accumulated during the `DROP COLUMN` operation to avoid running out of undo space. However, if this statement is interrupted after a checkpoint has been applied, then the table remains in an unusable state. While the table is unusable, the only operations allowed on it are `DROP TABLE`, `TRUNCATE TABLE`, and `ALTER TABLE DROP` ... `COLUMNS CONTINUE` (described in sections that follow).

You cannot use this clause with SET UNUSED, because that clause does not remove column data.

DROP COLUMNS CONTINUE Clause

Specify DROP COLUMNS CONTINUE to continue the drop column operation from the point at which it was interrupted. Submitting this statement while the table is in an invalid state results in an error.

Restrictions on Dropping Columns

Dropping columns is subject to the following restrictions:

- Each of the parts of this clause can be specified only once in the statement and cannot be mixed with any other ALTER TABLE clauses. For example, the following statements are not allowed:

  ```
  ALTER TABLE t1 DROP COLUMN f1 DROP (f2);
  ALTER TABLE t1 DROP COLUMN f1 SET UNUSED (f2);
  ALTER TABLE t1 DROP (f1) ADD (f2 NUMBER);
  ALTER TABLE t1 SET UNUSED (f3)
     ADD (CONSTRAINT ck1 CHECK (f2 > 0));
  ```

- You can drop an object type column only as an entity. To drop an attribute from an object type column, use the ALTER TYPE ... DROP ATTRIBUTE statement with the CASCADE INCLUDING TABLE DATA clause. Be aware that dropping an attribute affects all dependent objects. See *Oracle Database PL/SQL Language Reference* for more information.
- You can drop a column from an index-organized table only if it is not a primary key column. The primary key constraint of an index-organized table can never be dropped, so you cannot drop a primary key column even if you have specified CASCADE CONSTRAINTS.
- You can export tables with dropped or unused columns. However, you can import a table only if all the columns specified in the export files are present in the table (none of those columns has been dropped or marked unused). Otherwise, Oracle Database returns an error.
- You can set unused a column from a table that uses COMPRESS BASIC, but you cannot drop the column. However, all clauses of the *drop_column_clause* are valid for tables that use ROW STORE COMPRESS ADVANCED. See the semantics for *table_compression* for more information.
- You cannot drop a column on which a domain index has been built.
- You cannot drop a SCOPE table constraint or a WITH ROWID constraint on a REF column.
- You cannot use this clause to drop:
  - A pseudocolumn, cluster column, or partitioning column. You can drop nonpartitioning columns from a partitioned table if all the tablespaces where the partitions were created are online and in read/write mode.
  - A column from a nested table, an object table, a duplicated table, or a table owned by SYS.

See Also:

"Dropping a Column: Example"

add_period_clause

Use the `add_period_clause` to add a valid time dimension to `table`.

The `period_definition` clause of `ALTER TABLE` has the same semantics as in `CREATE TABLE`, with the following exceptions and additions:

- `valid_time_column` must not already exist in `table`.
- If you specify `start_time_column` and `end_time_column`, then these columns must already exist in `table` or you must specify the `add_column_clause` for each of these columns.
- If you specify `start_time_column` and `end_time_column` and these columns already exist in `table` and are populated with data, then for all rows where both columns have non-NULL values, the value of `start_time_column` must be earlier than the value of `end_time_column`.

> **See Also:**
>
> `CREATE TABLE` *period_definition* for the full semantics of this clause

drop_period_clause

Use the `drop_period_clause` to drop a valid time dimension from `table`.

For `valid_time_column`, specify the name of the valid time dimension you want to drop.

This clause has the following effects:

- The `valid_time_column` will be dropped from `table`.
- If the start time column and end time column were automatically created by Oracle Database when the valid time dimension was created, either with `CREATE TABLE` ... `period_definition` or `ALTER TABLE` ... `add_period_clause`, then they will be dropped. Otherwise, these columns will remain in `table` and revert to regular table columns.

> **See Also:**
>
> `CREATE TABLE` *period_definition* for more information on the `valid_time_column`, start time column, and end time column

rename_column_clause

Use the `rename_column_clause` to rename a column of `table`. The new column name must not be the same as any other column name in `table`.

When you rename a column, Oracle Database handles dependent objects as follows:

- Function-based indexes and check constraints that depend on the renamed column remain valid.
- Dependent views, triggers, functions, procedures, and packages are invalidated. Oracle Database attempts to revalidate them when they are next accessed, but you may need to alter these objects with the new column name if revalidation fails.

- If a domain index is defined on the column being renamed, then the database invokes the ODCIIndexAlter method with the `RENAME` option. This operation establishes correspondence between the indextype metadata and the base table

Restrictions on Renaming Columns

Renaming columns is subject to the following restrictions:

- You cannot combine this clause with any of the other *`column_clauses`* in the same statement.
- You cannot rename a column that is used to define a join index. Instead you must drop the index, rename the column, and re-create the index.
- You cannot rename a column in a duplicated table.

See Also:

"Renaming a Column: Example"

modify_collection_retrieval

Use the *`modify_collection_retrieval`* clause to change what Oracle Database returns when a collection item is retrieved from the database.

collection_item

Specify the name of a column-qualified attribute whose type is nested table or varray.

RETURN AS

Specify what Oracle Database should return as the result of a query:

- `LOCATOR` specifies that a unique locator for the nested table is returned.
- `VALUE` specifies that a copy of the nested table itself is returned.

See Also:

"Collection Retrieval: Example"

modify_LOB_storage_clause

The *`modify_LOB_storage_clause`* lets you change the physical attributes of *`LOB_item`*. You can specify only one *`LOB_item`* for each *`modify_LOB_storage_clause`*.

The sections that follow describe the semantics of parameters specific to modify_LOB_parameters. Unless otherwise documented in this section, the remaining LOB parameters have the same semantics when altering a table that they have when you are creating a table. Refer to the restrictions at the end of this section and to the `CREATE TABLE` clause *LOB_storage_parameters* for more information.

Note:

- You can modify LOB storage with an ALTER TABLE statement or with online redefinition by using the DBMS_REDEFINITION package. If you have not enabled LOB encryption, compression, or deduplication at create time, Oracle recommends that you use online redefinition to enable them after creation, as this process is more disk space efficient for changes to these three parameters. See *Oracle Database PL/SQL Packages and Types Reference* for more information on DBMS_REDEFINITION.
- 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.

PCTVERSION *integer*

Refer to the CREATE TABLE clause PCTVERSION integer for information on this clause.

LOB_retention_clause

If the database is in automatic undo mode, then you can specify RETENTION instead of PCTVERSION to instruct Oracle Database to retain old versions of this LOB. This clause overrides any prior setting of PCTVERSION.

FREEPOOLS *integer*

For BasicFiles LOBs, if the database is in automatic undo mode, then you can use this clause to specify the number of freelist groups for this LOB. This clause overrides any prior setting of FREELIST GROUPS. Refer to the CREATE TABLE clause FREEPOOLS integer for a full description of this parameter. The database ignores this parameter for SecureFiles LOBs.

REBUILD FREEPOOLS

This clause applies only to BasicFiles LOBs, not to SecureFiles LOBs. The REBUILD FREEPOOLS clause removes all the old versions of data from the LOB column. This clause is useful for removing all retained old version space in a LOB segment, freeing that space to be used immediately by new LOB data.

LOB_deduplicate_clause

This clause is valid only for SecureFiles LOBs. KEEP_DUPLICATES disables LOB deduplication. DEDUPLICATE enables LOB deduplication. All lobs in the segment are read, and any matching LOBs are deduplicated before returning.

LOB_compression_clause

This clause is valid only for SecureFiles LOBs. COMPRESS compresses all LOBs in the segment and then returns. NOCOMPRESS uncompresses all LOBs in the segment and then returns.

ENCRYPT | DECRYPT

LOB encryption has the same semantics as column encryption in general. See "ENCRYPT encryption_spec | DECRYPT" for more information.

CACHE, NOCACHE, CACHE READS

When you modify a LOB column from CACHE or NOCACHE to CACHE READS, or from CACHE READS to CACHE or NOCACHE, you can change the logging attribute. If you do not specify LOGGING or

NOLOGGING, then this attribute defaults to the current logging attribute of the LOB column. If you do not specify CACHE, NOCACHE, or CACHE READS, then Oracle Database retains the existing values of the LOB attributes.

Restrictions on Modifying LOB Storage

Modifying LOB storage is subject to the following restrictions:

- You cannot modify the value of the INITIAL parameter in the *storage_clause* when modifying the LOB storage attributes.
- You cannot specify both the *allocate_extent_clause* and the *deallocate_unused_clause* in the same statement.
- You cannot specify both the PCTVERSION and RETENTION parameters.
- You cannot specify the *shrink_clause* for SecureFiles LOBs.

See Also:

LOB_storage_clause (in CREATE TABLE) for information on setting LOB parameters and "LOB Columns: Examples"

alter_varray_col_properties

The *alter_varray_col_properties* clause lets you change the storage characteristics of an existing LOB in which a varray is stored.

Restriction on Altering Varray Column Properties

You cannot specify the TABLESPACE clause of *LOB_parameters* as part of this clause. The LOB tablespace for a varray defaults to the tablespace of the containing table.

REKEY *encryption_spec*

The REKEY clause causes the database to generate a new encryption key. All encrypted columns in the table are reencrypted using the new key and, if you specify the USING clause of the *encryption_spec*, a new encryption algorithm. You cannot combine this clause with any other clauses in this ALTER TABLE statement.

See Also:

Transparent Data Encryption for more information on transparent column encryption

constraint_clauses

Use the *constraint_clauses* to add a new constraint using out-of-line declaration, modify the state of an existing constraint, or drop a constraint. Refer to *constraint* for a description of all the keywords and parameters of out-of-line constraints and *constraint_state*.

Adding a Constraint

The ADD clause lets you add a new out-of-line constraint or out-of-line REF constraint to the table.

Restrictions on Adding a Constraint

Adding constraints is subject to the following restrictions:

- You cannot add a constraint to a duplicated table.
- You cannot add a foreign key constraint to a sharded table.

See Also:

"Disabling a CHECK Constraint: Example", "Specifying Object Identifiers: Example", and "REF Columns: Examples"

Modifying a Constraint

The `MODIFY CONSTRAINT` clause lets you change the state of an existing constraint.

The `CASCADE` keyword is valid only when you are disabling a unique or primary key constraint on which a foreign key constraint is defined. In this case, you must specify `CASCADE` so that the unique or primary key constraint and all of its dependent foreign key constraints are disabled.

Like the other constraint states you can set the precheck state of a constraint to `PRECHECK` and unset it with `NOPRECHECK`.

Restrictions on Modifying Constraints

Modifying constraints is subject to the following restrictions:

- You cannot change the state of a `NOT DEFERRABLE` constraint to `INITIALLY DEFERRED`.
- If you specify this clause for an index-organized table, then you cannot specify any other clauses in the same statement.
- You cannot change the `NOT NULL` constraint on a foreign key column of a reference-partitioned table, and you cannot change the state of a partitioning referential constraint of a reference-partitioned table.
- You cannot modify a constraint on a duplicated table.
- You can specify *`precheck_state`* only with *`constraint_name`*. Primary key and unique constraints are not supported.

See Also:

- "Adding and Modifying Precheck State Constraint: Example"
- "Changing the State of a Constraint: Examples"
- *Explicitly Declaring Column Check Constraints Precheckable or Not*

Renaming a Constraint

The `RENAME CONSTRAINT` clause lets you rename any existing constraint on *`table`*. The new constraint name cannot be the same as any existing constraint on any object in the same schema. All objects that are dependent on the constraint remain valid.

See Also:

"Renaming Constraints: Example"

drop_constraint_clause

The *drop_constraint_clause* lets you drop an integrity constraint from the database. Oracle Database stops enforcing the constraint and removes it from the data dictionary. You can specify only one constraint for each *drop_constraint_clause*, but you can specify multiple *drop_constraint_clause* in one statement.

PRIMARY KEY

Specify PRIMARY KEY to drop the primary key constraint of *table*.

UNIQUE

Specify UNIQUE to drop the unique constraint on the specified columns.

If you drop the primary key or unique constraint from a column on which a bitmap join index is defined, then Oracle Database invalidates the index. See CREATE INDEX for information on bitmap join indexes.

CONSTRAINT

Specify CONSTRAINT *constraint_name* to drop an integrity constraint other than a primary key or unique constraint.

CASCADE

Specify CASCADE if you want all other integrity constraints that depend on the dropped integrity constraint to be dropped as well.

KEEP INDEX | DROP INDEX

Specify KEEP INDEX or DROP INDEX to indicate whether Oracle Database should preserve or drop the index it has been using to enforce the PRIMARY KEY or UNIQUE constraint.

ONLINE

Specify ONLINE to indicate that DML operations on the table will be allowed while dropping the constraint.

Restrictions on Dropping Constraints

Dropping constraints is subject to the following restrictions:

- You cannot drop a primary key or unique key constraint that is part of a referential integrity constraint without also dropping the foreign key. To drop the referenced key and the foreign key together, use the CASCADE clause. If you omit CASCADE, then Oracle Database does not drop the primary key or unique constraint if any foreign key references it.
- You cannot drop a primary key constraint (even with the CASCADE clause) on a table that uses the primary key as its object identifier (OID).
- If you drop a referential integrity constraint on a REF column, then the REF column remains scoped to the referenced table.
- You cannot drop the scope of a REF column.

- You cannot drop the `NOT NULL` constraint on a foreign key column of a reference-partitioned table, and you cannot drop a partitioning referential constraint of a reference-partitioned table.
- You cannot drop the `NOT NULL` constraint on a column that is defined with a default column value using the `ON NULL` clause.
- You cannot specify the `ONLINE` clause when dropping a `DEFERRABLE` constraint.

See Also:

"Dropping Constraints: Examples"

alter_external_table

Use the *`alter_external_table`* clauses to change the characteristics of an external table. This clause has no affect on the external data itself. The syntax and semantics of the *`parallel_clause`*, *`enable_disable_clause`*, *`external_table_data_props`*, and `REJECT LIMIT` clause are the same as described for `CREATE TABLE`. See the *external_table_clause* (in `CREATE TABLE`).

PROJECT COLUMN Clause

This clause lets you determine how the access driver validates the rows of an external table in subsequent queries. The default is `PROJECT COLUMN ALL`, which means that the access driver processes all column values, regardless of which columns are selected, and validates only those rows with fully valid column entries. If any column value would raise an error, such as a data type conversion error, then the row is rejected even if that column was not referenced in the select list. If you specify `PROJECT COLUMN REFERENCED`, then the access driver processes only those columns in the select list.

The `ALL` setting guarantees consistent result sets. The `REFERENCED` setting can result in different numbers of rows returned, depending on the columns referenced in subsequent queries, but is faster than the `ALL` setting. If a subsequent query selects all columns of the external table, then the settings behave identically.

Restrictions on Altering External Tables

Altering external tables is subject to the following restrictions:

- You cannot modify an external table using any clause outside of this clause.
- You cannot add a `LONG`, varray, or object type column to an external table, nor can you change the data type of an external table column to any of these data types.
- You cannot modify the storage parameters of an external table.

alter_table_partitioning

The clauses in this section apply only to partitioned tables. You cannot combine partition operations with other partition operations or with operations on the base table in the same `ALTER TABLE` statement.

Notes on Changing Table Partitioning

The following notes apply when changing table partitioning:

- If you drop, exchange, truncate, move, modify, or split a partition on a table that is a master table for one or more materialized views, then existing bulk load information about the table will be deleted. Therefore, be sure to refresh all dependent materialized views before performing any of these operations.
- If a bitmap join index is defined on *table*, then any operation that alters a partition of *table* causes Oracle Database to mark the index UNUSABLE.
- The only *alter_table_partitioning* clauses you can specify for a reference-partitioned table are *modify_table_default_attrs*, *move_table_[sub]partition*, *truncate_partition_subpart*, and *exchange_partition_subpart*. None of these operations cascade to any child table of the reference-partitioned table. No other partition maintenance operations are valid on a reference-partitioned table, but you can specify the other partition maintenance operations on the parent table of a reference-partitioned table, and the operation will cascade to the child reference-partitioned table.
- When adding partitions and subpartitions, bear in mind that you can specify up to a total of 1024K-1 partitions and subpartitions for each table.
- When you add a table partition or subpartition and you omit the partition name, the database generates a name using the rules described in "Notes on Partitioning in General".
- When you move, add (hash only), coalesce, drop, split, merge, rename, or truncate a table partition or subpartition, the procedures, functions, packages, package bodies, views, type bodies, and triggers that reference the table remain valid. All other dependent objects are invalidated.
- Deferred segment creation is not supported for partition maintenance operations that create new segments on tables with LOB columns; segments will always be created for the involved (sub)partitions.
- For sharded tables, the only clauses you can specify for modifying table partitions and subpartitions are UNUSABLE LOCAL INDEXES and REBUILD UNUSABLE LOCAL INDEXES. You cannot perform any other modifications for individual partitions and subpartitions on a system sharded table.
- For user-defined sharded tables the following operations on partitions and subpartitions are supported:
  - add partition, add subpartition
  - drop partition, drop subpartition
  - split partition
  - modify partition to add or drop values to a list partition
- For sharded tables, the only supported partition maintenance operations are truncating partitions and subpartitions. You cannot perform any other partition maintenance operations on a sharded table.

For additional information on partition operations on tables with an associated CONTEXT domain index, refer to *Oracle Text Reference*.

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.

modify_table_default_attrs

The *modify_table_default_attrs* clause lets you specify new default values for the attributes of *table*. Only attributes named in the statement are affected. Partitions and LOB partitions you create subsequently will inherit these values unless you override them explicitly when creating the partition or LOB partition. Existing partitions and LOB partitions are not affected by this clause.

Only attributes named in the statement are affected, and the default values specified are overridden by any attributes specified at the individual partition or LOB partition level.

- `FOR` *`partition_extended_name`* applies only to composite-partitioned tables. This clause specifies new default values for the attributes of the partition identified in *`partition_extended_name`*. Subpartitions and LOB subpartitions of that partition that you create subsequently will inherit these values unless you override them explicitly when creating the subpartition or LOB subpartition. Existing subpartitions are not affected by this clause.

  If you are modifying the default directory, you can save its location using `DEFAULT DIRECTORY` *`directory`*.
- `PCTTHRESHOLD`, *`prefix_compression`*, and the *`alter_overflow_clause`* are valid only for partitioned index-organized tables.
- You can specify the *`prefix_compression`* clause only if prefix compression is already specified at the table level. Further, you cannot specify an integer after the `COMPRESS` keyword. Prefix length can be specified only when you create the table.
- You cannot specify the `PCTUSED` parameter in *`segment_attributes`* for the index segment of an index-organized table.
- The *`read_only_clause`* lets you modify the default read-only or read/write mode for the table. The new default mode will be assigned to partitions or subpartitions that are subsequently added to the table, unless you override this behavior by specifying the mode for the new partition or subpartition. When you modify the default read-only or read/write mode of a table, you do not change the mode of the existing partitions and subpartitions in the table. Refer to the *read_only_clause* of `CREATE TABLE` for the full semantics of this clause.
- The *`indexing_clause`* lets you modify the default indexing property for the table. The new default indexing property will be assigned to partitions or subpartitions that are subsequently added to the table, unless you override this behavior by specifying the indexing property for the new partition or subpartition. When you modify the default indexing property of a table, you do not change the indexing property of the existing partitions and subpartitions in the table. Refer to the *indexing_clause* of `CREATE TABLE` for the full semantics of this clause.

alter_automatic_partitioning

This clause allows you to manage automatic list-partitioned tables, as follows:

- Use the `SET PARTITIONING AUTOMATIC` clause to convert a regular list-partitioned table to an automatic list-partitioned table.
- Use the `SET PARTITIONING MANUAL` clause to convert an automatic list-partitioned table to a regular list-partitioned table.
- You can specify the `SET STORE IN` clause only for automatic list-partitioned tables. It lets you specify one or more tablespaces into which the database will store data for any subsequent automatically created list partitions. This clause overrides any tablespaces that might have been set for the table by a previously issued `SET STORE IN` clause.

To determine whether an existing table is an automatic list-partitioned table, you can query the `AUTOLIST` column of the `USER_`, `DBA_`, `ALL_PART_TABLES` data dictionary views.

Restriction on *alter_automatic_partitioning*

You cannot convert a regular list-partitioned table that contains a `DEFAULT` partition to an automatic list-partitioned table.

> **See Also:**
>
> The AUTOMATIC clause in the documentation on `CREATE TABLE` for more information on automatic list-partitioned tables

alter_interval_partitioning

Use this clause:

- To convert an existing range-partitioned table to interval partitioning. The database automatically creates partitions of the specified numeric range or datetime interval as needed for data beyond the highest value allowed for the last range partition. If the table has reference-partitioned child tables, then the child tables are converted to interval reference-partitioned child tables.
- To change the interval of an existing interval-partitioned table. The database first converts existing interval partitions to range partitions and determines the high value of the defined range partitions. The database then automatically creates partitions of the specified numeric range or datetime interval as needed for data that is beyond that high value.
- To change the tablespace storage for an existing interval-partitioned table. If the table has interval reference-partitioned child tables, then the new tablespace storage is inherited by any child table that does not have its own table-level default tablespace.
- To change an interval-partitioned table back to a range-partitioned table. Use `SET INTERVAL ()` to disable interval partitioning. The database converts existing interval partitions to range partitions, using the higher boundaries of created interval partitions as upper boundaries for the range partitions to be created. If the table has interval reference-partitioned child tables, then the child tables are converted to ordinary reference-partitioned child tables.

For *`expr`*, specify a valid number or interval expression.

> **See Also:**
>
> The `CREATE TABLE` "INTERVAL Clause" and *Oracle Database VLDB and Partitioning Guide* for more information on interval partitioning

set_subpartition_template

Use the *`set_subpartition_template`* clause to create or replace existing default range, list, or hash subpartition definitions for each table partition. This clause is valid only for composite-partitioned tables. It replaces the existing subpartition template or creates a new template if you have not previously created one. Existing subpartitions are not affected, nor are existing local and global indexes. However, subsequent partitioning operations (such as add and merge operations) will use the new template.

You can drop an existing subpartition template by specifying `ALTER TABLE` *`table`* `SET SUBPARTITION TEMPLATE ()`.

The *set_subpartition_template* clause has the same semantics as the *subpartition_template* clause of CREATE TABLE. Refer to the *subpartition_template* clause of CREATE TABLE for more information.

modify_table_partition

The *modify_table_partition* clause lets you change the real physical attributes of a range, hash, list partition, or system partition. This clause optionally modifies the storage attributes of one or more LOB items for the partition. You can specify new values for physical attributes (with some restrictions, as noted in the sections that follow), logging, and storage parameters.

For all types of partitions, you can also specify how Oracle Database should handle local indexes that become unusable as a result of the modification to the partition. See "UNUSABLE LOCAL INDEXES Clauses".

For partitioned index-organized tables, you can also update the mapping table in conjunction with partition changes. See the *alter_mapping_table_clauses* .

read_only_clause

Use the *read_only_clause* to put a table partition in read-only or read/write mode. Refer to the *read_only_clause* of CREATE TABLE for the full semantics of this clause.

indexing_clause

Use the *indexing_clause* to modify the indexing property of a table partition. The indexing property determines whether the partition is included in partial indexes on the table. You can specify the *indexing_clause* in the *modify_range_partition*, *modify_hash_partition*, and *modify_list_partition* clauses.

Specify INDEXING ON to change the indexing property for a table partition to ON. This operation has no effect on full indexes on the table. It has the following effects on partial indexes on the table:

- Local partial indexes: The table partition is included in the index. The corresponding index partition is rebuilt and marked USABLE.
- Global partial indexes: The table partition is included in the index. Index entries for the table partition are added to the index as part of routine index maintenance.

Specify INDEXING OFF to change the indexing property for a table partition to OFF. This operation has no effect on full indexes on the table. It has the following effects on partial indexes on the table:

- Local partial indexes: The table partition is excluded from the index. The corresponding index partition is marked UNUSABLE.
- Global partial indexes: The table partition is excluded from the index. Index entries for the table partition are removed from the index. This is a metadata-only operation and the index entries will continue to be physically stored in the index. You can remove these orphaned index entries by specifying COALESCE CLEANUP in the ALTER INDEX statement or in the *modify_index_partition* clause.

Restriction on column of type object

You cannot partition a table that has an object type. The alter table modification to a partitioned state is only supported for non-partitioned heap tables with zero columns of type object.

Restriction on the *indexing_clause*

You can specify this clause only for partitions of a simple partitioned table. For composite-partitioned tables, you can specify the *indexing_clause* at the table subpartition level. Refer to *modify_table_subpartition* for more information.

Notes on Modifying Table Partitions

The following notes apply to operations on range, list, and hash table partitions:

- For all types of table partition, in the *partition_attributes* clause, the *shrink_clause* lets you compact an individual partition segment. Refer to *shrink_clause* for additional information on this clause.
- The syntax and semantics for modifying a system partition are the same as those for modifying a hash partition. Refer to *modify_hash_partition*.
- If *table* is composite partitioned, then:
  - If you specify the *allocate_extent_clause*, then Oracle Database allocates an extent for each subpartition of *partition*.
  - If you specify the *deallocate_unused_clause*, then Oracle Database deallocates unused storage from each subpartition of *partition*.
  - Any other attributes changed in this clause will be changed in subpartitions of *partition* as well, overriding existing values. To avoid changing the attributes of existing subpartitions, use the `FOR PARTITION` clause of *modify_table_default_attrs*.
- When you modify the *partition_attributes* of a table partition with equipartitioned nested tables, the changes do not apply to the nested table partitions corresponding to the table partition being modified. However, you can modify the storage table of the nested table partition directly with an `ALTER TABLE` statement.
- Unless otherwise documented, the remaining clauses of *partition_attributes* have the same behavior they have when you are creating a partitioned table. Refer to the `CREATE TABLE` *table_partitioning_clauses* for more information.

See Also:

"Modifying Table Partitions: Examples"

modify_range_partition

Use this clause to modify the characteristics of a range partition.

add_range_subpartition

This clause is valid only for range-range composite partitions. It lets you add one or more range subpartitions to *partition*.

Starting with Oracle Database 12*c* Release 2 (12.2), you can use this clause to add a subpartition to composite-partitioned external table. In this case, you can specify the optional *external_part_subpart_data_props* clause of the *range_subpartition_desc* clause. Refer to external_part_subpart_data_props for the full semantics of this clause.

Restriction on Adding Range Subpartitions

If *table* is an index-organized table, then you can add only one range subpartition at a time.

add_hash_subpartition

This clause is valid only for range-hash composite partitions. The `add_hash_subpartition` clause lets you add a hash subpartition to `partition`. Oracle Database populates the new subpartition with rows rehashed from the other subpartition(s) of `partition` as determined by the hash function. For optimal load balancing, the total number of subpartitions should be a power of 2.

In the `partitioning_storage_clause`, the only clause you can specify for subpartitions is the `TABLESPACE` clause. If you do not specify `TABLESPACE`, then the new subpartition will reside in the default tablespace of `partition`.

Oracle Database adds local index partitions corresponding to the selected partition.

Oracle Database marks `UNUSABLE` the local index partitions corresponding to the added partitions. The database invalidates any indexes on heap-organized tables. You can update these indexes during this operation using the *update_index_clauses*.

add_list_subpartition

This clause is valid only for range-list and list-list composite partitions. It lets you add one or more list subpartitions to `partition`, and only if you have not already created a `DEFAULT` subpartition.

- The `list_values_clause` is required in this operation, and the values you specify in the `list_values_clause` cannot exist in any other subpartition of `partition`. However, these values can duplicate values found in subpartitions of other partitions.
- In the `partitioning_storage_clause`, the only clauses you can specify for subpartitions are the `TABLESPACE` clause and table compression.
- Starting with Oracle Database 12*c* Release 2 (12.2), you can use this clause to add a subpartition to composite-partitioned external table. In this case, you can specify the optional `external_part_subpart_data_props` clause of the `list_subpartition_desc` clause. Refer to external_part_subpart_data_props for the full semantics of this clause.

For each added subpartition, Oracle Database also adds a subpartition with the same value list to all local index partitions of the table. The status of existing local and global index partitions of `table` are not affected.

Restrictions on Adding List Subpartitions

The following restrictions apply to adding list subpartitions:

- You cannot specify this clause if you have already created a `DEFAULT` subpartition for this partition. Instead you must split the `DEFAULT` partition using the `split_list_subpartition` clause.
- If `table` is an index-organized table, then you can add only one list subpartition at a time.

coalesce_table_subpartition

`COALESCE SUBPARTITION` applies only to hash subpartitions. Use the `COALESCE SUBPARTITION` clause if you want Oracle Database to select the last hash subpartition, distribute its contents into one or more remaining subpartitions (determined by the hash function), and then drop the last subpartition.

- Oracle Database drops local index partitions corresponding to the selected partition.

- Oracle Database marks UNUSABLE the local index partitions corresponding to one or more absorbing partitions. The database invalidates any global indexes on heap-organized tables. You can update these indexes during this operation using the *update_index_clauses*.

modify_hash_partition

When modifying a hash partition, in the *partition_attributes* clause, you can specify only the *allocate_extent_clause* and *deallocate_unused_clause*. All other attributes of the partition are inherited from the table-level defaults except TABLESPACE, which stays the same as it was at create time.

modify_list_partition

Clauses available to you when modifying a list partition have the same semantics as when you are modifying a range partition. When modifying a list partition, the following additional clauses are available:

ADD | DROP VALUES Clauses

These clauses are valid only when you are modifying composite partitions. Local and global indexes on the table are not affected by either of these clauses.

- Use the ADD VALUES clause to extend the *partition_key_value* list of *partition* to include additional values. The added partition values must comply with all rules and restrictions listed in the CREATE TABLE clause *list_partitions* .
- Use the DROP VALUES clause to reduce the *partition_key_value* list of *partition* by eliminating one or more *partition_key_value*. When you specify this clause, Oracle Database checks to ensure that no rows with this value exist. If such rows do exist, then Oracle Database returns an error.

> **Note:**
>
> ADD VALUES and DROP VALUES operations on a table with a DEFAULT list partition are enhanced if you have defined a local prefixed index on the table.

Restrictions on Adding and Dropping List Values

Adding and dropping list values are subject to the following restrictions:

- You cannot add values to or drop values from a DEFAULT list partition.
- If *table* contains a DEFAULT partition and you attempt to add values to a nondefault partition, then Oracle Database will check that the values being added do not already exist in the DEFAULT partition. If the values do exist in the DEFAULT partition, then Oracle Database returns an error.

modify_table_subpartition

This clause applies only to composite-partitioned tables. Its subclauses let you modify the characteristics of an individual range, list, or hash subpartition.

The *shrink_clause* lets you compact an individual subpartition segment. Refer to *shrink_clause* for additional information on this clause.

You can also specify how Oracle Database should handle local indexes that become unusable as a result of the modification to the partition. See "UNUSABLE LOCAL INDEXES Clauses".

Use the `read_only_clause` to put a table subpartition in read-only or read/write mode. Refer to the *read_only_clause* of `CREATE TABLE` for the full semantics of this clause.

Use the `indexing_clause` to modify the indexing property of a table subpartition. The indexing property determines whether the subpartition is included in partial indexes on the table. Modifying the indexing property of table subpartitions has the same effect on index subpartitions as modifying the indexing property of table partitions has on index partitions. Refer to the indexing_clause of `modify_table_partition` for details.

Restriction on Modifying Hash Subpartitions

The only `modify_LOB_parameters` you can specify for `subpartition` are the `allocate_extent_clause` and `deallocate_unused_clause`.

ADD | DROP VALUES Clauses

These clauses are valid only when you are modifying list subpartitions. Local and global indexes on the table are not affected by either of these clauses.

- Use the `ADD VALUES` clause to extend the `subpartition_key_value` list of `subpartition` to include additional values. The added partition values must comply with all rules and restrictions listed in the `CREATE TABLE` clause *list_partitions* .
- Use the `DROP VALUES` clause to reduce the `subpartition_key_value` list of `subpartition` by eliminating one or more `subpartition_key_value`. When you specify this clause, Oracle Database checks to ensure that no rows with this value exist. If such rows do exist, then Oracle Database returns an error.

You can also specify how Oracle Database should handle local indexes that become unusable as a result of the modification to the partition. See "UNUSABLE LOCAL INDEXES Clauses".

Restriction on Modifying List Subpartitions

The only `modify_LOB_parameters` you can specify for `subpartition` are the `allocate_extent_clause` and `deallocate_unused_clause`.

move_table_partition

Use the `move_table_partition` clause to move `partition` to another segment. You can move partition data to another tablespace, recluster data to reduce fragmentation, or change create-time physical attributes.

If the table contains LOB columns, then you can use the `LOB_storage_clause` to move the LOB data and LOB index segments associated with this partition. Only the LOBs named are affected. If you do not specify the `LOB_storage_clause` for a particular LOB column, then its LOB data and LOB index segments are not moved.

If the table contains nested table columns, then you can use the `nested_table_col_properties clause` of the `table_partition_description` to move the nested table segments associated with this partition. Only the nested table items named are affected. If you do not specify the `nested_table_col_properties clause` of the `table_partition_description` for a particular nested table column, then its segments are not moved.

Oracle Database moves local index partitions corresponding to the specified partition. If the moved partitions are not empty, then the database marks them `UNUSABLE`. The database invalidates global indexes on heap-organized tables. You can update these indexes during this operation using the *update_index_clauses*.

When you move a LOB data segment, Oracle Database drops the old data segment and corresponding index segment and creates new segments even if you do not specify a new tablespace.

The move operation obtains its parallel attribute from the *parallel_clause*, if specified. When it is not specified, the default parallel attributes of the table, if any, are used. If neither is specified, then Oracle Database performs the move serially.

Specifying the *parallel_clause* in MOVE PARTITION does not change the default parallel attributes of *table*.

Note:

For index-organized tables, Oracle Database uses the address of the primary key, as well as its value, to construct logical rowids. The logical rowids are stored in the secondary index of the table. If you move a partition of an index-organized table, then the address portion of the rowids will change, which can hamper performance. To ensure optimal performance, rebuild the secondary index(es) on the moved partition to update the rowids.

See Also:

"Moving Table Partitions: Example"

MAPPING TABLE

The MAPPING TABLE clause is relevant only for an index-organized table that already has a mapping table defined for it. Oracle Database moves the mapping table along with the moved index-organized table partition. The mapping table partition inherits the physical attributes of the moved index-organized table partition. This is the only way you can change the attributes of the mapping table partition. If you omit this clause, then the mapping table partition retains its original attributes.

Oracle Database marks UNUSABLE all corresponding bitmap index partitions.

Refer to the *mapping_table_clauses* (in CREATE TABLE) for more information on this clause.

ONLINE

Specify ONLINE to indicate that DML operations on the table partition will be allowed while moving the table partition.

Restrictions on the ONLINE Clause

The ONLINE clause is subject to the following restrictions when moving table partitions:

- You cannot specify the ONLINE clause for tables owned by SYS.
- You cannot specify the ONLINE clause for index-organized tables.
- You cannot specify the ONLINE clause for heap-organized tables that contain object types or on which bitmap join indexes or domain indexes are defined.

- Parallel DML and direct path `INSERT` operations require an exclusive lock on the table. Therefore, these operations are not supported concurrently with an ongoing online partition `MOVE`, due to conflicting locks.

Restrictions on Moving Table Partitions

Moving table partitions is subject to the following restrictions:

- If *`partition`* is a hash partition, then the only attribute you can specify in this clause is `TABLESPACE`.
- You cannot specify this clause for a partition containing subpartitions. However, you can move subpartitions using the *`move_table_subpartition`* clause.

move_table_subpartition

Use the *`move_table_subpartition`* clause to move the subpartition identified by *`subpartition_extended_name`* to another segment. If you do not specify `TABLESPACE`, then the subpartition remains in the same tablespace.

If the subpartition is not empty, then Oracle Database marks `UNUSABLE` all local index subpartitions corresponding to the subpartition being moved. You can update all indexes on heap-organized tables during this operation using the *update_index_clauses*.

If the table contains LOB columns, then you can use the *`LOB_storage_clause`* to move the LOB data and LOB index segments associated with this subpartition. Only the LOBs specified are affected. If you do not specify the *`LOB_storage_clause`* for a particular LOB column, then its LOB data and LOB index segments are not moved.

When you move a LOB data segment, Oracle Database drops the old data segment and corresponding index segment and creates new segments even if you do not specify a new tablespace.

ONLINE

Specify `ONLINE` to indicate that DML operations on the table subpartition will be allowed while moving the table subpartition.

Restrictions on the ONLINE Clause

The `ONLINE` clause for moving table subpartitions is subject to the same restrictions as the `ONLINE` clause for moving table partitions. Refer to "Restrictions on the ONLINE Clause."

Restriction on Moving Table Subpartitions

The only clauses of the *`partitioning_storage_clause`* you can specify are the `TABLESPACE` clause and *`table_compression`*.

add_external_partition_attrs

Use this clause to add external parameters to a partitioned table.

add_table_partition

Use the *`add_table_partition`* clause to add one or more range, list, or system partitions to *`table`*, or to add one hash partition to *`table`*.

For each partition added, Oracle Database adds to any local index defined on *`table`* a new partition with the same name as that of the base table partition. If the index already has a partition with such a name, then Oracle Database generates a partition name of the form `SYS_P`*`n`*.

If `table` is index organized, then for each partition added Oracle Database adds a partition to any mapping table and overflow area defined on the table as well.

If `table` is the parent table of a reference-partitioned table, then you can use the `dependent_tables_clause` to propagate the partition maintenance operation you are specifying in this statement to all the reference-partitioned child tables.

The default indexing property of `table` is inherited by the new table partition(s). You can override this by setting the indexing property of a list, range, or system partition using the `indexing_clause` in the `table_partition_description` clause, or a hash partition using the `indexing_clause` in the `add_hash_partition_clause`.

For each partition added to a composite-partitioned table, Oracle Database adds a new index partition with the same subpartition descriptions to all local indexes defined on `table`. Global indexes defined on `table` are not affected. If you specify the indexing property for the new table partition, then the new subpartitions inherit the indexing property for the partition. Otherwise, the new subpartitions inherit the default indexing property for the table. You can override this by setting the indexing property of a subpartition using the `indexing_clause` in the `range_subpartition_desc`, `individual_hash_subparts`, and `list_subpartition_desc` clauses.

BEFORE Clause

You can specify the optional `BEFORE` clause only when adding system partitions to `table`. This clause lets you specify where the new partition(s) should be added in relation to existing partitions. You cannot split a system partition. Therefore, this clause is useful if you want to divide the contents of one existing partition among multiple new partitions. If you omit this clause, then the database adds the new partition(s) after the existing partitions.

Restriction on Adding Table Partitions

If `table` is an index-organized table, or if a local domain index is defined on `table`, then you can add only one partition at a time.

See Also:

"Adding a Table Partition with a LOB and Nested Table Storage: Examples" and "Adding Multiple Partitions to a Table: Example"

add_range_partition_clause

The `add_range_partition_clause` lets you add a new range partition to the high end of a range-partitioned or composite range-partitioned table (after the last existing partition).

If a domain index is defined on `table`, then the index must not be marked `IN_PROGRESS` or `FAILED`.

Restrictions on Adding Range Partitions

Adding range partitions is subject to the following restrictions:

- If the upper partition bound of each partitioning key in the existing high partition is `MAXVALUE`, then you cannot add a partition to the table. Instead, use the `split_table_partition` clause to add a partition at the beginning or the middle of the table.

- The *prefix_compression* and OVERFLOW clauses, are valid only for a partitioned index-organized table. You can specify *prefix_compression* only if prefix compression is enabled at the table level. You can specify OVERFLOW only if the partitioned table already has an overflow segment.
- You cannot specify the PCTUSED parameter for the index segment of an index-organized table.

range_values_clause

Specify the upper bound for the new partition. The *value_list* is a comma-delimited, ordered list of literal values corresponding to the partitioning key columns. The *value_list* must collate greater than the partition bound for the highest existing partition in the table.

table_partition_description

Use this clause to specify any create-time physical attributes for the new partition. If the table contains LOB columns, then you can also specify partition-level attributes for one or more LOB items.

external_part_subpart_data_props

Starting with Oracle Database 12*c* Release 2 (12.2), Oracle supports partitioned and composite-partitioned external tables. When adding a partition to such a table, you can optionally use this clause to specify the DEFAULT DIRECTORY and LOCATION for the partition. Refer to DEFAULT DIRECTORY and LOCATION in the documentation on CREATE TABLE for the full semantics of these clauses.

Subpartition Descriptions

These clauses are valid only for composite-partitioned tables. Use the *range_subpartition_desc*, *list_subpartition_desc*, *individual_hash_subparts*, or *hash_subparts_by_quantity* clause as appropriate, if you want to specify subpartitions for the new partition. This clause overrides any subpartition descriptions defined in *subpartition_template* at the table level.

add_hash_partition_clause

The *add_hash_partition_clause* lets you add a new hash partition to the high end of a hash-partitioned table. Oracle Database populates the new partition with rows rehashed from other partitions of *table* as determined by the hash function. For optimal load balancing, the total number of partitions should be a power of 2.

You can specify a name for the partition, and optionally a tablespace where it should be stored. If you do not specify a name, then the database assigns a partition name of the form SYS_P*n*. If you do not specify TABLESPACE, then the new partition is stored in the default tablespace of the table. Other attributes are always inherited from table-level defaults.

If this operation causes data to be rehashed among partitions, then the database marks UNUSABLE any corresponding local index partitions. You can update all indexes on heap-organized tables during this operation using the *update_index_clauses*.

Use the *parallel_clause* to specify whether to parallelize the creation of the new partition.

Use the *read_only_clause* to put a table partition in read-only or read/write mode. Refer to the *read_only_clause* of CREATE TABLE for the full semantics of this clause.

Use the *indexing_clause* to specify the indexing property for the partition. If you do not specify this clause, then the partition inherits the default indexing property of *table*.

See Also:

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

add_list_partition_clause

The *add_list_partition_clause* lets you add a new partition to *table* using a new set of partition values. You can specify any create-time physical attributes for the new partition. If the table contains LOB columns, then you can also specify partition-level attributes for one or more LOB items.

Restrictions on Adding List Partitions

You cannot add a list partition if you have already defined a DEFAULT partition for the table. Instead, you must use the *split_table_partition* clause to split the DEFAULT partition.

See Also:

- *list_partitions* of CREATE TABLE for more information and restrictions on list partitions
- "Working with Default List Partitions: Example"

add_system_partition_clause

Use this clause to add a partition to a system-partitioned table. Oracle Database adds a corresponding index partition to all local indexes defined on the table.

The *table_partition_description* lets you specify partition-level attributes of the new partition. The values of any unspecified attributes are inherited from the table-level values.

Restriction on Adding System Partitions

You cannot specify the OVERFLOW clause when adding a system partition.

See Also:

The CREATE TABLE clause *system_partitioning* for more information on system partitions

coalesce_table_partition

COALESCE applies only to hash partitions. Use the *coalesce_table_partition* clause to indicate that Oracle Database should select the last hash partition, distribute its contents into one or more remaining partitions as determined by the hash function, and then drop the last partition.

Oracle Database drops local index partitions corresponding to the selected partition. The database marks UNUSABLE the local index partitions corresponding to one or more absorbing partitions. The database invalidates any indexes on heap-organized tables. You can update all indexes during this operation using the *update_index_clauses*.

Restriction on Coalescing Table Partitions

If you update global indexes using the *update_all_indexes_clause*, then you can specify only the keywords `UPDATE INDEXES`, not the subclause.

drop_external_partition_attrs
Use this clause to drop external parameters in a partitioned table.

drop_table_partition

The *drop_table_partition* clause removes partitions, and the data in those partitions, from a partitioned table. If you want to drop a partition but keep its data in the table, then you must merge the partition into one of the adjacent partitions.

Starting with Oracle Database 12*c* Release 2 (12.2), you can use this clause to drop a partition from a partitioned table or composite-partitioned external table.

See Also:

merge_table_partitions

Use the *partition_extended_names* clause to specify one or more partitions to be dropped. When specifying multiple partitions, you must specify all partitions by name, as shown in the upper branch of the syntax diagram, or all partitions using the `FOR` clause, as shown in the lower branch of the syntax diagram. You cannot use both types of syntax in one drop operation.

- If *table* has LOB columns, then Oracle Database also drops the LOB data and LOB index partitions and any subpartitions corresponding to the table partition(s) being dropped.
- If *table* has equipartitioned nested table columns, then Oracle Database also drops the nested table partitions corresponding to the table partition(s) being dropped.
- If *table* is index organized and has a mapping table defined on it, then the database drops the corresponding mapping table partition(s) as well.
- Oracle Database drops local index partitions and subpartitions corresponding to the dropped partition(s), even if they are marked `UNUSABLE`.

You can update indexes on *table* during this operation using the *update_index_clauses*. Updates to global indexes are metadata-only and the index entries for records that are dropped by the drop operation will continue to be physically stored in the index. You can remove these orphaned index entries by specifying `COALESCE CLEANUP` in the ALTER INDEX statement or in the *modify_index_partition* clause.

If you specify the *parallel_clause* with the *update_index_clauses*, then the database parallelizes the index update, not the drop operation.

If you drop a range partition and later insert a row that would have belonged to the dropped partition, then the database stores the row in the next higher partition. However, if that partition is the highest partition, then the insert will fail, because the range of values represented by the dropped partition is no longer valid for the table.

Restrictions on Dropping Table Partitions

Dropping table partitions is subject to the following restrictions:

- You cannot drop a partition of a hash-partitioned table. Instead, use the *coalesce_table_partition* clause.
- You cannot drop all of the partitions in a table. Instead, drop the table.
- If you update global indexes using the update_all_indexes_clause, then you can specify only the UPDATE INDEXES keywords but not the subclause.
- If *table* is an index-organized table, or if a local domain index is defined on *table*, then you can drop only one partition at a time.
- You cannot drop a partition of a duplicated table.
- Dropping a partition does not place the partition in the Oracle Database recycle bin, regardless of the setting of the recycle bin. Dropped partitions are immediately removed from the system.

See Also:

"Dropping a Table Partition: Example"

drop_table_subpartition

Use this clause to drop range or list subpartitions from a range, list, or hash composite-partitioned table. Oracle Database deletes any rows in the dropped subpartition(s).

Starting with Oracle Database 12*c* Release 2 (12.2), you can use this clause to drop a subpartition from a composite-partitioned external table.

Use the *subpartition_extended_names* clause to specify one or more subpartitions to be dropped. When specifying multiple subpartitions, you must specify all subpartitions by name, as shown in the upper branch of the syntax diagram, or all subpartitions using the FOR clause, as shown in the lower branch of the syntax diagram. You cannot use both types of syntax in one drop operation.

Oracle Database drops the corresponding subpartition(s) of any local index. Other index subpartitions are not affected. Any global indexes are marked UNUSABLE unless you specify the *update_global_index_clause* or *update_all_indexes_clause*. Updates to global indexes are metadata-only and the index entries for records that are dropped by the drop operation will continue to be physically stored in the index. You can remove these orphaned index entries by specifying COALESCE CLEANUP in the ALTER INDEX statement or in the *modify_index_partition* clause.

Restrictions on Dropping Table Subpartitions

Dropping table subpartitions is subject to the following restrictions:

- You cannot drop a hash subpartition. Instead use the MODIFY PARTITION ... COALESCE SUBPARTITION syntax.
- You cannot drop all of the subpartitions in a partition. Instead, use the *drop_table_partition* clause.
- If you update the global indexes, then you cannot specify the optional subclause of the *update_all_indexes_clause*.
- If *table* is an index-organized table, then you can drop only one subpartition at a time.

- When dropping multiple subpartitions, all of the subpartitions must be in the same partition.
- You cannot drop a subpartition of a duplicated table.

rename_partition_subpart

Use the *rename_partition_subpart* clause to rename a table partition or subpartition to *new_name*. For both partitions and subpartitions, *new_name* must be different from all existing partitions and subpartitions of the same table.

If *table* is index organized, then Oracle Database assigns the same name to the corresponding primary key index partition as well as to any existing overflow partitions and mapping table partitions.

Starting with Oracle Database 12*c* Release 2 (12.2), you can use this clause to rename a partition or subpartition in a partitioned or composite-partitioned external table.

See Also:

"Renaming Table Partitions: Examples"

truncate_partition_subpart

Specify TRUNCATE *partition_extended_names* to remove all rows from the partition(s) identified by *partition_extended_names* or, if the table is composite partitioned, all rows from the subpartitions of those partitions. Specify TRUNCATE *subpartition_extended_names* to remove all rows from individual subpartitions. If *table* is index organized, then Oracle Database also truncates any corresponding mapping table partitions and overflow area partitions.

When specifying multiple partitions, you must specify all partitions by name, as shown in the upper branch of the *partition_extended_names* syntax diagram, or all partitions using the FOR clause, as shown in the lower branch of the syntax diagram. You cannot use both types of syntax in one truncate operation. The same rule applies when specifying multiple subpartitions with the subpartition_extended_names clause.

For each specified partition or subpartition:

- If the partition or subpartition to be truncated contain data, then you must first disable any referential integrity constraints on the table. Alternatively, you can delete the rows and then truncate the partition.
- If *table* contains any LOB columns, then the LOB data and LOB index segments for the partition are also truncated. If *table* is composite partitioned, then the LOB data and LOB index segments for the subpartitions of the partition are truncated.
- If *table* contains any equipartitioned nested tables, then you cannot truncate the parent partition unless its corresponding nested table partition is empty.
- If a domain index is defined on *table*, then the index must not be marked IN_PROGRESS or FAILED, and the index partition corresponding to the table partition being truncated must not be marked IN_PROGRESS.

For each partition or subpartition truncated, Oracle Database also truncates corresponding local index partitions and subpartitions. If those index partitions or subpartitions are marked UNUSABLE, then the database truncates them and resets the UNUSABLE marker to VALID.

You can update indexes on *table* during this operation using the *update_index_clauses*. Updates to global indexes are metadata-only and the index entries for records that are dropped by the truncate operation will continue to be physically stored in the index. You can remove these orphaned index entries by specifying `COALESCE CLEANUP` in the ALTER INDEX statement or in the *modify_index_partition* clause.

If you specify the *`parallel_clause`* with the *`update_index_clauses`*, then the database parallelizes the index update, not the truncate operation.

DROP STORAGE

Specify `DROP STORAGE` to deallocate all space from the deleted rows, except the space allocated by the `MINEXTENTS` parameter. This space can subsequently be used by other objects in the tablespace.

DROP ALL STORAGE

Specify `DROP ALL STORAGE` to deallocate all space from the deleted rows, including the space allocated by the `MINEXTENTS` parameter. All segments for the partition(s) or subpartition(s), as well as all segments for their dependent objects, will be deallocated.

Restrictions on DROP ALL STORAGE

This clause is subject to the same restrictions as described in "Restrictions on Deferred Segment Creation".

REUSE STORAGE

Specify `REUSE STORAGE` to keep space from the deleted rows allocated to the partition(s) or subpartition(s). The space is subsequently available only for inserts and updates to the same partition(s) or subpartition(s).

CASCADE

Specify `CASCADE` to truncate the corresponding partition(s) or subpartition(s) in all reference-partitioned child tables of *`table`*.

Restrictions on Truncating Table Partitions and Subpartitions

Truncating table partitions and subpartitions is subject to the following restrictions:

- If you update global indexes using the *`update_all_indexes_clause`*, then you can specify only the `UPDATE INDEXES` keywords, not the subclause.
- If *`table`* is an index-organized table, or if a local domain index is defined on *`table`*, then you can truncate only one table partition or one table subpartition at a time.
- You cannot truncate partitions or subpartitions in a duplicated table.

See Also:

"Truncating Table Partitions: Example"

split_table_partition

The *`split_table_partition`* clause lets you create, from the partition identified by *`partition_extended_name`*, multiple new partitions, each with a new segment, new physical attributes, and new initial extents. The segment associated with the current partition is discarded.

The new partitions inherit all unspecified physical attributes from the current partition.

Note:

Oracle Database can optimize and speed up `SPLIT PARTITION` and `SPLIT SUBPARTITION` operations if specific conditions are met. Refer to *Oracle Database VLDB and Partitioning Guide* for information on optimizing these operations.

- If you split a `DEFAULT` list partition, then the last resulting partition will have the `DEFAULT` value. All other resulting partitions will have the specified split values.
- If *`table`* is index organized, then Oracle Database splits any corresponding mapping table partition and places it in the same tablespace as the parent index-organized table partition. The database also splits any corresponding overflow area, and you can use the `OVERFLOW` clause to specify segment attributes for the new overflow areas.
- If *`table`* contains LOB columns, then you can use the *`LOB_storage_clause`* to specify separate LOB storage attributes for the LOB data segments resulting from the split. The database drops the LOB data and LOB index segments of the current partition and creates new segments for each LOB column, for each partition, even if you do not specify a new tablespace.
- If *`table`* contains nested table columns, then you can use the *`split_nested_table_part`* clause to specify the storage table names and segment attributes of the nested table segments resulting from the split. The database drops the nested table segments of the current partition and creates new segments for each nested table column, for each partition. This clause allows for multiple nested table columns in the parent table as well as multilevel nested table columns.

Oracle Database splits the corresponding local index partition, even if it is marked `UNUSABLE`. The database marks `UNUSABLE`, and you must rebuild the local index partitions corresponding to the split partitions. The new index partitions inherit their attributes from the partition being split. The database stores the new index partitions in the default tablespace of the index partition being split. If that index partition has no default tablespace, then the database uses the tablespace of the new underlying table partitions.

AT Clause

The `AT` clause applies only to range partitions and lets you split one range partition into two range partitions. Specify the new noninclusive upper bound for the first of the two new partitions. The value list must compare less than the original partition bound for the current partition and greater than the partition bound for the next lowest partition (if there is one).

VALUES Clause

The `VALUES` clause applies only to list partitions and allows you to split one list partition into two list partitions. 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. You can specify `NULL` if you have not already specified `NULL` for another partition in the table. 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. Oracle Database creates the first new partition using the *`list_values`* you specify and creates the second new partition using the remaining partition values from the current partition. Therefore, the value list cannot contain all of the partition values of the current partition, nor can it contain any partition values that do not already exist for the current partition.

INTO Clause

The `INTO` clause lets you describe the new partitions resulting from the split.

- The `AT ... INTO` clause lets you describe the partitions resulting from splitting one range partition into two range partitions. In *`range_partition_desc`*, the keyword `PARTITION` is required even if you do not specify the optional names and physical attributes of the two partitions resulting from the split. If you do not specify new partition names, then Oracle Database assigns names of the form `SYS_P`*`n`*. Any attributes you do not specify are inherited from the current partition.
- The `VALUES ... INTO` clause lets you describe the partitions resulting from splitting one list partition into two list partitions. In *`list_partition_desc`*, the keyword `PARTITION` is required even if you do not specify the optional names and physical attributes of the two partitions resulting from the split. If you do not specify new partition names, then Oracle Database assigns names of the form `SYS_P`*`n`*. Any attributes you do not specify are inherited from the current partition.
- The `INTO` clause lets you split one range partition into two or more range partitions, or one list partition into two or more list partitions. If you do not specify new partition names, then Oracle Database assigns names of the form `SYS_P`*`n`*. Any attributes you do not specify are inherited from the current partition.
  - You must specify range partitions in ascending order of their partition bounds. The partition bound of the first specified range partition must be greater than the partition bound for the next lowest partition in the table (if there is one). Do not specify a partition bound for the last range partition; it will inherit the partition bound of the current partition.
  - For list partitions, all specified partition values for the new partitions must exist in the current partition. Do not specify any partition values for the last partition. Oracle Database creates the last partition using the remaining partition values from the current partition.

For range-hash composite-partitioned tables, if you specify subpartitioning for the new partitions, then you can specify only `TABLESPACE` and table compression for the subpartitions. All other attributes are inherited from the current partition. If you do not specify subpartitioning for the new partitions, then their tablespace is also inherited from the current partition.

For range-list and list-list composite-partitioned tables, you cannot specify subpartitions for the new partitions at all. The list subpartitions of the split partition inherit the number of subpartitions and value lists from the current partition.

For all composite-partitioned tables for which you do not specify subpartition names for the newly created subpartitions, the newly created subpartitions inherit their names from the parent partition as follows:

- For those subpartitions in the parent partition with names of the form *`partition_name`* underscore (_) *`subpartition_name`* (for example, `P1_SUBP1`), Oracle Database generates corresponding names in the newly created subpartitions using the new partition names (for example `P1A_SUB1` and `P1B_SUB1`).
- For those subpartitions in the parent partition with names of any other form, Oracle Database generates subpartition names of the form `SYS_SUBP`*`n`*.

Oracle Database splits the corresponding partition(s) in each local index defined on *`table`*, even if the index is marked `UNUSABLE`.

If *`table`* is the parent table of a reference-partitioned table, then you can use the *`dependent_tables_clause`* to propagate the partition maintenance operation you are specifying in this statement to all the reference-partitioned child tables.

Oracle Database invalidates any indexes on heap-organized tables. You can update these indexes during this operation using the *update_index_clauses*.

The *parallel_clause* lets you parallelize the split operation but does not change the default parallel attributes of the table.

ONLINE

Specify ONLINE to indicate that DML operations on the table will be allowed while splitting the table partition.

Restrictions on the ONLINE Clause

The ONLINE clause is subject to the following restrictions when splitting table partitions:

- You cannot specify the ONLINE clause for tables owned by SYS.
- You cannot specify the ONLINE clause for index-organized tables.
- You cannot specify the ONLINE clause if a domain index is defined on the table.
- You cannot specify the ONLINE clause for heap-organized tables that contain object types or on which bitmap join indexes are defined.
- Parallel DML and direct path INSERT operations require an exclusive lock on the table. Therefore, these operations are not supported concurrently with an ongoing online partition split, due to conflicting locks.

Restrictions on Splitting Table Partitions

Splitting table partitions is subject to the following restrictions:

- You cannot specify this clause for a hash partition.
- You cannot specify the *parallel_clause* for index-organized tables.
- If *table* is an index-organized table, or if a local domain index is defined on *table*, then you can split the partition into only two new partitions.

See Also:

"Splitting Table Partitions: Examples"

split_table_subpartition

Use this clause to split a subpartition into multiple new subpartitions with nonoverlapping value lists.

Note:

Oracle Database can optimize and speed up SPLIT PARTITION and SPLIT SUBPARTITION operations if specific conditions are met. Refer to *Oracle Database VLDB and Partitioning Guide* for information on optimizing these operations.

AT Clause

The AT clause is valid only for range subpartitions. Specify the new noninclusive upper bound for the first of the two new subpartitions. The value list must compare less than the original subpartition bound for the subpartition identified by *subpartition_extended_name* and greater than the partition bound for the next lowest subpartition (if there is one).

VALUES Clause

The VALUES clause is valid only for list subpartitions. If the table is subpartitioned on one key column, then use the upper branch of the *list_values* syntax to specify a list of values for that column. You can specify NULL if you have not already specified NULL for another subpartition in the same partition. If the table is subpartitioned 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. Oracle Database creates the first new subpartition using the subpartition value list you specify and creates the second new partition using the remaining partition values from the current subpartition. Therefore, the value list cannot contain all of the partition values of the current subpartition, nor can it contain any partition values that do not already exist for the current subpartition.

INTO Clause

The INTO clause lets you describe the new subpartitions resulting from the split.

- The AT ... INTO clause lets you describe the two subpartitions resulting from splitting one range partition into two range partitions. In *range_subpartition_desc*, the keyword SUBPARTITION is required even if you do not specify the optional names and attributes of the two new subpartitions. If you do not specify new subpartition names, then Oracle Database assigns names of the form SYS_SUBP*n* Any attributes you do not specify are inherited from the current subpartition.
- The VALUES ... INTO clause lets you describe the two subpartitions resulting from splitting one list partition into two list partitions. In *list_subpartition_desc*, the keyword SUBPARTITION is required even if you do not specify the optional names and attributes of the two new subpartitions. If you do not specify new subpartition names, then Oracle Database assigns names of the form SYS_SUBP*n* Any attributes you do not specify are inherited from the current subpartition.
- The INTO clause lets you split one range subpartition into two or more range subpartitions, or one list subpartition into two or more list subpartitions. If you do not specify new subpartition names, then Oracle Database assigns names of the form SYS_SUBP*n*. Any attributes you do not specify are inherited from the current subpartition.
  - You must specify range subpartitions in ascending order of their subpartition bounds. The subpartition bound of the first specified range subpartition must be greater than the subpartition bound for the next lowest subpartition (if there is one). Do not specify a subpartition bound for the last range subpartition; it will inherit the partition bound of the current subpartition.
  - For list subpartitions, all specified subpartition values for the new subpartitions must exist in the current subpartition. Do not specify any subpartition values for the last subpartition. Oracle Database creates the last subpartition using the remaining partition values from the current subpartition.

Oracle Database splits any corresponding local subpartition index, even if it is marked UNUSABLE. The new index subpartitions inherit the names of the new table subpartitions unless those names are already held by index subpartitions. In that case, the database assigns new index subpartition names of the form SYS_SUBPn. The new index subpartitions inherit physical attributes from the parent subpartition. However, if the parent subpartition does not have a default TABLESPACE attribute, then the new subpartitions inherit the tablespace of the corresponding new table subpartitions.

Oracle Database invalidates indexes on heap-organized tables. You can update these indexes by using the *update_index_clauses*.

ONLINE

Specify `ONLINE` to indicate that DML operations on the table will be allowed while splitting the table subpartition.

Restrictions on the ONLINE Clause

The `ONLINE` clause for splitting table subpartitions is subject to the same restrictions as the `ONLINE` clause for splitting table partitions. Refer to Restrictions on the ONLINE Clause.

Restrictions on Splitting Table Subpartitions

Splitting table subpartitions is subject to the following restrictions:

- You cannot specify this clause for a hash subpartition.
- In subpartition descriptions, the only clauses of *`partitioning_storage_clause`* you can specify are `TABLESPACE` and table compression.
- You cannot specify the *`parallel_clause`* for index-organized tables.
- If *`table`* is an index-organized table, then you can split the subpartition into only two new subpartitions.

merge_table_partitions

The *`merge_table_partitions`* clause lets you merge the contents of two or more range, list, or system partitions of *`table`* into one new partition and then drop the original partitions. This clause is not valid for hash partitions. Use the *`coalesce_table_partition`* clause instead.

Specify a comma-separated list of two or more range, list, or system partitions to be merged. You can use the `TO` clause to specify two or more adjacent range partitions to be merged.

For each partition, use *`partition`* to specify a partition name or the `FOR` clause to specify a partition without using its name. See "References to Partitioned Tables and Indexes " for more information on the `FOR` clause.

- The partitions to be merged must be adjacent and must be specified in ascending order of their partition bounds if they are range partitions. List partitions and system partitions need not be adjacent in order to be merged.
- When you merge range partitions, the new partition inherits the partition bound of the highest of the original partitions.
- When you merge list partitions, the resulting partition value list is the union of the set of the partition values lists of the partitions being merged. If you merge a `DEFAULT` list partition with other list partitions, then the resulting partition will be the `DEFAULT` partition and will have the `DEFAULT` value.
- When you merge composite range partitions or composite list partitions, range-list or list-list composite partitions, you cannot specify subpartition descriptions. Oracle Database obtains the subpartitioning information from the subpartition template. If you have not specified a subpartition template, then the database creates one `MAXVALUE` subpartition from range subpartitions or one `DEFAULT` subpartition from list subpartitions.

Any attributes you do not specify explicitly for the new partition are inherited from table-level defaults. However, if you reuse one of the partition names for the new partition, then the new partition inherits values from the partition whose name is being reused rather than from table-level default values.

Oracle Database drops local index partitions corresponding to the selected partitions and marks UNUSABLE the local index partition corresponding to merged partition. The database also marks UNUSABLE any global indexes on heap-organized tables. You can update all these indexes during this operation using the *update_index_clauses*.

If *table* is the parent table of a reference-partitioned table, then you can use the *dependent_tables_clause* to propagate the partition maintenance operation you are specifying in this statement to all the reference-partitioned child tables.

ONLINE

Specify ONLINE to allow DML operations on the table partitions during the merge partitions operation.

Restriction on Merging Table Partitions

If *table* is an index-organized table, or if a local domain index is defined on *table*, then you can merge only two partitions at a time.

See Also:

"Merging Two Table Partitions: Example", "Merging Four Adjacent Range Partitions: Example", and "Working with Default List Partitions: Example"

merge_table_subpartitions

The *merge_table_subpartitions* clause lets you merge the contents of two or more range or list subpartitions of *table* into one new subpartition and then drop the original subpartitions. This clause is not valid for hash subpartitions. Use the *coalesce_hash_subpartition* clause instead.

Specify a comma-separated list of two or more range or list subpartitions to be merged. You can use the TO clause to specify two or more adjacent range subpartitions to be merged.

For each subpartition, use *subpartition* to specify a subpartition name or the FOR clause to specify a subpartition without using its name. See "References to Partitioned Tables and Indexes " for more information on the FOR clause.

The subpartitions to be merged must belong to the same partition. If they are range subpartitions, then they must be adjacent. If they are list subpartitions, then they need not be adjacent. The data in the resulting subpartition consists of the combined data from the merged subpartitions.

If you specify the INTO clause, then in the *range_subpartition_desc* or *list_subpartition_desc* you cannot specify the *range_values_clause* or *list_values_clause*, respectively. Further, the only clauses you can specify in the *partitioning_storage_clause* are the TABLESPACE clause and *table_compression*.

Any attributes you do not specify explicitly for the new subpartition are inherited from partition-level values. However, if you reuse one of the subpartition names for the new subpartition, then the new subpartition inherits values from the subpartition whose name is being reused rather than from partition-level default values.

Oracle Database merges corresponding local index subpartitions and marks the resulting index subpartition UNUSABLE. The database also marks UNUSABLE both partitioned and nonpartitioned global indexes on heap-organized tables. You can update all indexes during this operation using the *update_index_clauses*.

ONLINE

Specify `ONLINE` to allow DML operations on the table subpartitions during the merge subpartitions operation.

Restriction on Merging Table Subpartitions

If *`table`* is an index-organized table, then you can merge only two subpartitions at a time.

exchange_partition_subpart

Use the `EXCHANGE PARTITION` or `EXCHANGE SUBPARTITION` clause to exchange the data and index segments of:

- One nonpartitioned table with:
  - one range, list, or hash partition
  - one range, list, or hash subpartition
- One range-partitioned table with the range subpartitions of a range-range or list-range composite-partitioned table partition
- One hash-partitioned table with the hash subpartitions of a range-hash or list-hash composite-partitioned table partition
- One list-partitioned table with the list subpartitions of a range-list or hash-list composite-partitioned table partition

In all cases, the structure of the table and the partition or subpartition being exchanged, including their partitioning keys, must be identical. In the case of list partitions and subpartitions, the corresponding value lists must also match.

This clause facilitates high-speed data loading when used with transportable tablespaces.

See Also:

Oracle Database Administrator's Guide for information on transportable tablespaces

If *`table`* contains LOB columns, then for each LOB column Oracle Database exchanges LOB data and LOB index partition or subpartition segments with corresponding LOB data and LOB index segments of *`table`*.

If *`table`* has nested table columns, then for each such column Oracle Database exchanges nested table partition segments with corresponding nested table segments of the nonpartitioned table.

If *`table`* contains an identity column, then so must the partition or subpartition being exchanged, and vice versa. The sequence generators must both be increasing or decreasing. The sequence generators are not exchanged, so *`table`* and the partition or subpartition will continue to use the same sequence generators. The high water mark for both sequence generators will be adjusted so that new identity column values will not conflict with existing values.

All of the segment attributes of the two objects (including tablespace and logging) are also exchanged.

Existing statistics for the table being exchanged into the partitioned table will be exchanged. However, the global statistics for the partitioned table will not be altered. Use the `DBMS_STATS.GATHER_TABLE_STATS` procedure to re-create global statistics. You can set the `GRANULARITY` attribute equal to '`APPROX_GLOBAL AND PARTITION`' to speed up the process and aggregate new global statistics based on the existing partition statistics. See *Oracle Database PL/SQL Packages and Types Reference* for more information on this packaged procedure.

Oracle Database invalidates any global indexes on the objects being exchanged. You can update the global indexes on the table whose partition is being exchanged by using either the update_global_index_clause or the update_all_indexes_clause. For the *`update_all_indexes_clause`*, you can specify only the keywords `UPDATE INDEXES`, not the subclause. Global indexes on the table being exchanged remain invalidated. The *`update_global_index_clause`* and *`update_all_indexes_clause`* do not update local indexes during an exchange operation. You can specify local index maintenance by using the INCLUDING | EXCLUDING INDEXES clause. If you specify the *`parallel_clause`* with either of these clauses, then the database parallelizes the index update, not the exchange operation.

See Also:

"Notes on Exchanging Partitions and Subpartitions"

WITH TABLE

Specify the *`table`* with which the partition or subpartition will be exchanged. If you omit *`schema`*, then Oracle Database assumes that *`table`* is in your own schema.

INCLUDING | EXCLUDING INDEXES

Specify `INCLUDING INDEXES` if you want local index partitions or subpartitions to be exchanged with the corresponding table index (for a nonpartitioned table) or local indexes (for a hash-partitioned table). Specify `EXCLUDING INDEXES` if you want all index partitions or subpartitions corresponding to the partition and all the regular indexes and index partitions on the exchanged table to be marked `UNUSABLE`. If you omit this clause, then the default is `EXCLUDING INDEXES`.

WITH | WITHOUT VALIDATION

Specify `WITH VALIDATION` if you want Oracle Database to return an error if any rows in the exchanged table do not map into partitions or subpartitions being exchanged. Specify `WITHOUT VALIDATION` if you do not want Oracle Database to check the proper mapping of rows in the exchanged table. If you omit this clause, then the default is `WITH VALIDATION`.

exceptions_clause

See "Handling Constraint Exceptions " for information on this clause. In the context of exchanging partitions, this clause is valid only if the partitioned table has been defined with a `UNIQUE` constraint, and that constraint must be in `DISABLE VALIDATE` state. This clause is valid only for exchanging partition, not subpartitions.

CASCADE

Specify `CASCADE` to exchange the corresponding partition or subpartition in all reference-partitioned child tables of *`table`*. The reference-partitioned table hierarchies of the source and target must match.

Restrictions on CASCADE

The following restrictions apply to the `CASCADE` clause:

- You cannot specify `CASCADE` if a parent key in the reference-partitioned table hierarchy is referenced by multiple partitioning constraints.
- You cannot specify `CASCADE` if a domain index or an `XMLIndex` index is defined on any of the reference-partitioned child tables of *`table`*.

See Also:

- The `DBMS_IOT` package in *Oracle Database PL/SQL Packages and Types Reference* for information on the SQL scripts
- *Oracle Database Administrator's Guide* for information on eliminating migrated and chained rows
- *constraint* for more information on constraint checking and "Creating an Exceptions Table for Index-Organized Tables: Example"

Notes on Exchanging Partitions and Subpartitions

The following notes apply when exchanging partitions and subpartitions:

- Both tables involved in the exchange must have the same primary key, and no validated foreign keys can be referencing either of the tables unless the referenced table is empty.
- When exchanging partitioned index-organized tables:
  - The source and target table or partition must have their primary key set on the same columns, in the same order.
  - If prefix compression is enabled, then it must be enabled for both the source and the target, and with the same prefix length.
  - Both the source and target must be index organized.
  - Both the source and target must have overflow segments, or neither can have overflow segments. Also, both the source and target must have mapping tables, or neither can have a mapping table.
  - Both the source and target must have identical storage attributes for any LOB columns.

See Also:

"Exchanging Table Partitions: Example"

dependent_tables_clause

This clause is valid only when you are altering the parent table of a reference-partitioned table. The clause lets you specify attributes of partitions that are created by the operation for reference-partitioned child tables of the parent table.

- If the parent table is not composite partitioned, then specify one or more child tables, and for each child table specify one *`partition_spec`* for each partition created in the parent table.

- If the parent table is composite, then specify one or more child tables, and for each child table specify one *partition_spec* for each subpartition created in the parent table.

> **See Also:**
>
> The CREATE TABLE clause *reference_partitioning* for information on creating reference-partitioned tables and *Oracle Database VLDB and Partitioning Guide* for information on partitioning by reference in general

UNUSABLE LOCAL INDEXES Clauses

These two clauses modify the attributes of local index partitions and index subpartitions corresponding to *partition*, depending on whether you are modifying a partition or subpartition.

- UNUSABLE LOCAL INDEXES marks UNUSABLE the local index partition or index subpartition associated with *partition*.
- REBUILD UNUSABLE LOCAL INDEXES rebuilds the unusable local index partition or index subpartition associated with *partition*.

Restrictions on UNUSABLE LOCAL INDEXES

This clause is subject to the following restrictions:

- You cannot specify this clause with any other clauses of the *modify_table_partition* clause.
- You cannot specify this clause in the *modify_table_partition* clause for a partition that has subpartitions. However, you can specify this clause in the *modify_table_subpartition* clause.

update_index_clauses

Use the *update_index_clauses* to update the indexes on *table* as part of the table partitioning operation. When you perform DDL on a table partition, if an index is defined on *table*, then Oracle Database invalidates the entire index, not just the partitions undergoing DDL. This clause lets you update the index partition you are changing during the DDL operation, eliminating the need to rebuild the index after the DDL.

The *update_index_clauses* are not needed, and are not valid, for partitioned index-organized tables. Index-organized tables are primary key based, so Oracle can keep global indexes USABLE during operations that move data but do not change its value.

update_global_index_clause

Use this clause to update only global indexes on *table*. Oracle Database marks UNUSABLE all local indexes on *table*.

UPDATE GLOBAL INDEXES

Specify UPDATE GLOBAL INDEXES to update the global indexes defined on *table*.

Restriction on Updating Global Indexes

If the global index is a global domain index defined on a LOB column, then Oracle Database marks the domain index UNUSABLE instead of updating it.

INVALIDATE GLOBAL INDEXES

Specify `INVALIDATE GLOBAL INDEXES` to invalidate the global indexes defined on *`table`*.

If you specify neither, then Oracle Database invalidates the global indexes.

Restrictions on Invalidating Global Indexes

This clause is supported only for global indexes. It is not supported for index-organized tables. In addition, this clause updates only indexes that are `USABLE` and `VALID`. `UNUSABLE` indexes are left unusable, and `INVALID` global indexes are ignored.

update_all_indexes_clause

Use this clause to update all indexes on *`table`*.

update_index_partition

This clause is valid only for operations on table partitions and affects only local indexes.

- The *`index_partition_description`* lets you specify physical attributes, tablespace storage, and logging for each partition of each local index. If you specify only the `PARTITION` keyword, then Oracle Database updates the index partition as follows:
  - For operations on a single table partition (such as `MOVE PARTITION` and `SPLIT PARTITION`), the corresponding index partition inherits the attributes of the affected index table partition, Oracle Database does not generate names for new index partitions, so any new index partitions resulting from this operation inherit their names from the corresponding new table partition.
  - For `MERGE PARTITION` operations, the resulting local index partition inherits its name from the resulting table partition and inherits its attributes from the local index.

  For a domain index, you can use the `PARAMETERS` clause to specify the parameter string that is passed uninterpreted to the appropriate ODCI indextype routine. The `PARAMETERS` clause is valid only for domain indexes, and is the only part of the *`index_partition_description`* you can specify for a domain index.

  See Also:

  Oracle Database Data Cartridge Developer's Guide for more information on domain indexes

- For a composite-partitioned index, the *`index_subpartition_clause`* lets you specify tablespace storage for each subpartition. Refer to the index_subpartition_clause (in `CREATE INDEX`) for more information on this component of the *`update_index_partition`* clause.

For information on the `USABLE` and `UNUSABLE` keywords, refer to `ALTER INDEX` ... USABLE | UNUSABLE.

update_index_subpartition

This clause is valid only for operations on subpartitions of composite-partitioned tables and affects only local indexes on composite-partitioned tables. It lets you specify tablespace storage for one or more subpartitions.

Restrictions on Updating All Indexes

The following restrictions apply to the *`update_all_indexes_clause`*:

- You cannot specify this clause for index-organized tables.
- When you exchange a partition or subpartition with the *exchange_partition_subpart* clause, the *update_all_indexes_clause* is applicable only to global indexes. Therefore, you cannot specify the *update_index_partition* or *update_index_subpartition* clauses. You can, however, specify local index maintenance during an exchange operation by using the INCLUDING | EXCLUDING INDEXES clause.

See Also:

"Updating Global Indexes: Example" and "Updating Partitioned Indexes: Example"

parallel_clause

The *parallel_clause* lets you change the default degree of parallelism for queries and DML on the table.

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

Restrictions on Changing Table Parallelization

Changing parallelization is subject to the following restrictions:

- If *table* contains any columns of LOB or user-defined object type, then subsequent `INSERT`, `UPDATE`, and `DELETE` operations on *table* are executed serially without notification. Subsequent queries, however, are executed in parallel.
- If you specify the *parallel_clause* in conjunction with the *move_table_clause*, then the parallelism applies only to the move, not to subsequent DML and query operations on the table.

See Also:

"Specifying Parallel Processing: Example"

alter_table_partitionset

The clauses of *alter_table_partitionset* only apply to sharded tables within a composite sharding setup.

The following notes apply when changing table partitioning of sharded tables:

- Specify *add_partitionset* only to root sharded tables.
- For add and split partitionset operations you must deploy a new primary shardspace per partitionset before specifying *add_partitionset* .
- For add and split partitionset operations partition names do not have a value list after them. This is different from partition by list.
- Specify *modify_partitionset* only to root sharded tables that are partitioned by list.
- You must provide all the partitionset names as these are not generated by the system.

split_partitionset

Use *split_partitionset* to split an existing partitionset into one or more partitionsets. This clause is valid only for root sharded table in a composite sharding setup. New primary shardspaces must be deployed per new partitionset before executing *split_partitionset*.

See Operations on Directory-Based Partitioned Table for examples.

add_partitionset

add_partitionset clause is only valid for a root sharded table in a composite sharding setup. When a new primary shardspace is created, *add_partitionset* needs to be used to create new partitionsets.

See Operations on Directory-Based Partitioned Table for examples.

modify_partitionset

modify_partitionset clause is only value for a root sharded table that is partitionset by LIST. Use this clause to add a new list value to an existing partitionset.

move_partitionset

Use *move_partitionset* to move all existing partitions of a sharded table in a partitionset to new tablespace sets.

filter_condition

This clause lets you specify which rows to preserve during the following `ALTER TABLE` operations: moving, splitting, or merging table partitions or subpartitions; moving a table; or converting a nonpartitioned table to a partitioned table. The database preserves only the rows that satisfy the condition specified in the *where_clause*. Refer to the *where_clause* in the documentation on `SELECT` for the full semantics of this clause.

Restrictions on Filter Conditions

The following restrictions apply to the *filter_condition* clause:

- Filter conditions are supported only for heap-organized tables.
- Filter conditions can refer only to columns in the table being altered. Filter conditions cannot contain operations, such as joins or subqueries, that reference other database objects.
- Filter conditions are unsupported for tables with primary or unique keys that are referenced by enabled foreign keys.

Restrictions and Notes on Using Filter Conditions with Online Operations

The following restrictions and notes apply when you specify a filter condition for an online `ALTER TABLE` operation:

- You cannot specify both the *filter_condition* and `ONLINE` clauses if supplemental logging is enabled.
- When you specify both the *filter_condition* and `ONLINE` clauses, DML operations on the table are allowed during the `ALTER TABLE` operation. The filter condition does not have a direct effect on the concurrent DML operations. However, consider this combination carefully, because the filter operation and the DML operations could unintentionally conflict, as follows:

- Inserts into a nonpartitioned table will succeed. Inserts into a partitioned table will succeed if they do not violate the partitioning key criteria.
- Delete operations will apply only to rows that are preserved by the filter condition throughout the `ALTER TABLE` operation.
- Update operations will apply only to rows that are preserved by the filter condition throughout the `ALTER TABLE` operation. These update operations will succeed, regardless of whether the update operation would have disqualified the rows for preservation by the filter condition.
- Rows that do not qualify for preservation by the filter condition at the onset of the `ALTER TABLE` operation will not be preserved, regardless of whether an update operation would qualify the rows for preservation.

allow_disallow_clustering

This clause is valid for tables that use attribute clustering. It lets you allow or disallow attribute clustering for data movement that occurs during the move table operation specified by the *move_table_clause*, and the table partition and subpartition maintenance operations specified by the *coalesce_table_[sub]partition*, *merge_table_[sub]partitions*, *move_table_[sub]partition*, and *split_table_[sub]partition* clauses.

- Specify `ALLOW CLUSTERING` to allow attribute clustering for data movement. This clause overrides a `NO ON DATA MOVEMENT` setting in the DDL that created or altered the table.
- Specify `DISALLOW CLUSTERING` to disallow attribute clustering for data movement. This clause overrides a `YES ON DATA MOVEMENT` setting in the DDL that created or altered the table.

The *allow_disallow_clustering* clause has no effect if you specify it for a table that does not use attribute clustering.

See Also:

clustering_when clause of `CREATE TABLE` for more information on the `NO ON DATA MOVEMENT` and `YES ON DATA MOVEMENT` clauses

{ DEFERRED | IMMEDIATE } INVALIDATION

This clause lets you control when the database invalidates dependent cursors while performing table partition maintenance operations.

- If you specify `DEFERRED INVALIDATION`, then the database avoids or defers invalidating dependent cursors, when possible.
- If you specify `IMMEDIATE INVALIDATION`, then the database immediately invalidates dependent cursors, as it did in Oracle Database 12*c* Release 1 (12.1) and prior releases. This is the default.

If you omit this clause, then the value of the `CURSOR_INVALIDATION` initialization parameter determines when cursors are invalidated.

You can specify this clause only when performing table partition maintenance operations; it is not supported for any other `ALTER TABLE` operations.

See Also:

- *Oracle Database SQL Tuning Guide* for more information on cursor invalidation
- *Oracle Database Reference* for more information in the `CURSOR_INVALIDATION` initialization parameter

move_table_clause

The *`move_table_clause`* lets you relocate data of a nonpartitioned or partitioned table into new segments. Alternatively you can move a partition or subpartition of a partitioned table into a new segment, optionally in a different tablespace, and optionally modify any of its storage attributes.

You can also move any LOB data segments associated with the table or partition using the *`LOB_storage_clause`* and *`varray_col_properties`* clause. LOB items not specified in this clause are not moved.

Moving Partitions and Subpartitions of Heap-Organized Tables

You can move all the partitions and subpartitions of a partitioned heap-organized table with a single `ALTER TABLE MOVE` statement.

Existing partition and subpartition properties that are not modified on table level will be preserved. For example, if you specify `COMPRESS` for the `ALTER TABLE MOVE` command, then all partitions will be compressed, whereas the tablespace location for each partition will be preserved. Conversely, if you specify a target tablespace for the `ALTER TABLE MOVE` , then all partitions will reside in the specified tablespace after the move, but the individual compression attribute for each partition will be preserved.

Restrictions on Moving All Partitions and Subpartions of a Partitioned Table with One Command

- You cannot use this functionality if a domain index is defined on the table.
- You cannot use this functionality if the table has columns of type `VARRAY`.
- You cannot change the attribute clustering properties.
- You can only control table-level segment attributes_clauses, such as tablespace or compression. Any segment attribute that is managed as default on the table-level is not supported.
- You cannot use this functionality for an index-organized table .

ONLINE Clause

Specify `ONLINE` if you want DML operations on the table to be allowed while the table is being moved.

Restrictions on Moving Tables Online

Moving tables online is subject to the following restrictions:

- You cannot combine this clause with any other clause in the same statement.
- You cannot specify this clause for a partitioned index-organized table.
- You cannot specify this clause if a domain index is defined on the table, like spatial, XML, or Text indexes.

- Parallel DML and direct path INSERT operations require an exclusive lock on the table. Therefore, these operations are not supported concurrently with an ongoing online table MOVE, due to conflicting locks.
- You cannot specify this clause for index-organized tables that contain any LOB, VARRAY, Oracle-supplied type, or user-defined object type columns.

index_org_table_clause

For an index-organized table, the *index_org_table_clause* of the *move_table_clause* lets you additionally specify overflow segment attributes. The *move_table_clause* rebuilds the primary key index of the index-organized table. The overflow data segment is not rebuilt unless the OVERFLOW keyword is explicitly stated, with two exceptions:

- If you alter the values of PCTTHRESHOLD or the INCLUDING column as part of this ALTER TABLE statement, then the overflow data segment is rebuilt.
- If you explicitly move any of out-of-line columns (LOBs, varrays, nested table columns) in the index-organized table, then the overflow data segment is also rebuilt.

The index and data segments of LOB columns are not rebuilt unless you specify the LOB columns explicitly as part of this ALTER TABLE statement.

mapping_table_clause

Specify MAPPING TABLE if you want Oracle Database to create a mapping table if one does not already exist. If it does exist, then the database moves the mapping table along with the index-organized table, and marks any bitmapped indexes UNUSABLE. The new mapping table is created in the same tablespace as the parent table.

Specify NOMAPPING to instruct the database to drop an existing mapping table.

Refer to *mapping_table_clauses* (in CREATE TABLE) for more information on this clause.

Restriction on Mapping Tables

You cannot specify NOMAPPING if any bitmapped indexes have been defined on *table*.

prefix_compression

Use the *prefix_compression* clause to enable or disable prefix compression in an index-organized table.

- COMPRESS enables prefix compression, which eliminates repeated occurrence 1of primary key column values in index-organized tables. Use *integer* to specify the prefix length (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.
- NOCOMPRESS disables prefix compression in index-organized tables. This is the default.

TABLESPACE *tablespace*

Specify the tablespace into which the rebuilt index-organized table is to be stored.

LOB_storage_clause

Use this clause to move a LOB segment to a different tablespace. You cannot use this clause to move a LOB segment if the table contains a LONG column. Instead, you must either convert the LONG column to a LOB, or you must export the table, re-create the table specifying the desired tablespace storage for the LOB column, and re-import the table data.

UPDATE INDEXES

This clause is valid only when performing online or offline moves of heap-organized tables. It allows you to update all global indexes on the table.

You can optionally change the tablespace for an index or index partition, as follows:

- Specify the `segment_attributes_clause` to change the tablespace of a nonpartitioned global index. Within this clause, you can specify only the `TABLESPACE` clause.
- Specify the `update_index_partition` clause to change the tablespace for a partition of a partitioned global index. Within this clause, you can specify only the `TABLESPACE` clause of the `segment_attributes_clause`.

Restrictions on Moving Tables

Moving tables is subject to the following restrictions:

- If you specify `MOVE`, then it must be the first clause in the `ALTER TABLE` statement, and the only clauses outside this clause that are allowed are the `physical_attributes_clause`, the `parallel_clause`, and the `LOB_storage_clause`.
- You cannot move a table containing a `LONG` or `LONG RAW` column.
- You cannot `MOVE` an entire partitioned table (either heap- or index-organized). You must move individual partitions or subpartitions.

Note:

For any LOB columns you specify in a `move_table_clause`:

- Oracle Database drops the old LOB data segment and corresponding index segment and creates new segments, even if you do not specify a new tablespace.
- If the LOB index in `table` resided in a different tablespace from the LOB data, then Oracle Database collocates the LOB index in the same tablespace with the LOB data after the move.

See Also:

move_table_partition and *move_table_subpartition*

modify_to_partitioned

Use this clause to partition a nonpartitioned or partitioned table, including indexes, online or offline.

You can change a nonpartitioned or partitioned table into any type of partitioned or composite partitioned table with the following characteristics:

- All data in the original table is preserved.
- The data in the newly created partitions or subpartitions of the modified table is stored in the same tablespace as the original table, unless you specify otherwise in the `table_partitioning_clauses`.

- Local index partitions or subpartitions and lob partitions or subpartitions of the modified table will be co- located with the table partitions or subpartitions unless you specify otherwise in the `table_partitioning_clauses`.
- All triggers, constraints, and VPD policies defined on the original table are preserved.
- If table compression is defined on the original nonpartitioned table, then the partitioned table will use the same type of table compression.
- In case of modifying a partitioned table, the compression setting of the newly created partitions or subpartitions is derived from the default compression setting of the partitioned table prior to the modification unless all partitions or subpartitions shared the same compression method.

Each range, list, or hash partitioning or subpartitioning key column with a character data type, specified in the `modify_to_partitioned` clause must have one of the following declared collations: `BINARY`, `USING_NLS_COMP`, `USING_NLS_SORT`, or `USING_NLS_SORT_CS`.

table_partitioning_clauses

Use this clause to specify the partitioning attributes for the table.

Each range, list, or hash partitioning or subpartitioning key column with a character data type, specified in the `modify_to_partitioned` clause must have one of the following declared collations: `BINARY`, `USING_NLS_COMP`, `USING_NLS_SORT`, or `USING_NLS_SORT_CS`.

This clause has the same semantics here as it has for the `CREATE TABLE` statement. Refer to the `CREATE TABLE` *table_partitioning_clauses* for the full semantics of this clause.

NONPARTITIONED

Specify `NONPARTITIONED` to convert a partitioned table back to a nonpartitioned state.

ONLINE

Specify `ONLINE` to indicate that DML operations on the table will be allowed while changing to a partitioned table.

UPDATE INDEXES

Use this clause to specify how existing indexes on the table are converted into global partitioned indexes or local partitioned indexes.

- For `index`, specify the name of an existing index on the table.
- Specify the `local_partitioned_index` clause to convert `index` into a local partitioned index. This clause has the same semantics here as it has for the `CREATE INDEX` statement. Refer to the clause *local_partitioned_index* in the documentation on `CREATE INDEX` for the full semantics of this clause.
- Specify the `global_partitioned_index` clause to convert `index` into a global partitioned index. This clause has the same semantics here as it has for the `CREATE INDEX` statement. Refer to the clause *global_partitioned_index* in the documentation on `CREATE INDEX` for the full semantics of this clause.
- Specify the `GLOBAL` keyword to allow prefixed partitioned and nonpartitioned global indexes to retain their global shape. This clause prevents such indexes from being converted to local partitioned indexes; it has no effect on nonprefixed global indexes.

If you specify only the `UPDATE INDEXES` keywords, or omit the `UPDATE INDEXES` clause altogether, then existing indexes are converted as follows:

- Nonprefixed indexes retain their original shape: normal indexes are converted to nonpartitioned global indexes, nonpartitioned global indexes remain the same, and partitioned global indexes remain the same and retain their partitioning shape.
- Prefixed indexes are converted to local partitioned indexes. Prefixed indexes include partitioning keys in the index definition, but the index definition is not limited to including only the partitioning keys.
- Bitmap indexes are converted to local partitioned indexes, regardless of whether they are prefixed or not.

Default Index Rules for Conversion from Partitioned to Partitioned Table

The rule set for default index conversion for partitioned to partitioned table is identical to the one for nonpartitioned to partitioned table, with additional handling of existing local indexes on the partitioned table.

- If the index is already local, then the index stays as a local index if the index column is prefixed on both sides of the partitioning dimensions.
- If the partitioning columns are a subset of the key columns, (that is, they are prefixed), then the global index is converted to local. If the global index is not prefixed, then the shape of the global index is retained.

Restrictions on Changing a Nonpartitioned Table to a Partitioned Table

The following restrictions apply to the *`modify_to_partitioned`* clause:

- You cannot specify this clause for an index-organized table.
- You cannot specify this clause if a domain index is defined on the table.
- You cannot specify `ONLINE` when changing a nonpartitioned table to a reference-partitioned child table. This operation is supported only in offline mode.

See Also:

Oracle Database VLDB and Partitioning Guide for more information on converting a nonpartitioned table into a partitioned table

modify_opaque_type

Use the *`modify_opaque_type`* clause to instruct the database to store the specified abstract data type or `XMLType` in an `ANYDATA` column using unpacked storage.

You can specify any abstract data type with this clause. However, it is primarily useful because it allows you to specify the following data types, which cannot be stored in an `ANYDATA` column using conventional storage:

- `XMLType`
- Abstract data types that contain one or more attributes of type `XMLType`, `CLOB`, `BLOB`, or `NCLOB`.

When you use **unpacked storage**, data types are stored in system-generated hidden columns that are associated with the `ANYDATA` column. You can insert and query these data types as you would data types that are stored in an `ANYDATA` column using conventional storage.

anydata_column

Specify the name of a column of type `ANYDATA`. If *`type_name`* is an abstract data type that does not contain an attribute of type `XMLType`, `CLOB`, `BLOB`, or `NCLOB`, then *`anydata_column`* must be empty.

type_name

Specify the name of one or more abstract data types or `XMLType`. The abstract data type can contain an attribute of type `XMLType`, `CLOB`, `BLOB`, or `NCLOB`. The type can be `EDITIONABLE`. When you subsequently insert these data types into *`anydata_column`*, they will use unpacked storage. If you previously specified this clause for the same *`anydata_column`*, then unpacked storage will continue to be used for the previously specified data types as well as the newly specified data types.

See Also:

Oracle Database PL/SQL Packages and Types Reference for information on the `ANYDATA` type and "Unpacked Storage in ANYDATA Columns: Example"

immutable_table_clauses

You can use the `NO DROP` or `NO DELETE` clauses to modify the definition of an immutable table.

Use the `NO DROP` clause to modify the retention period for an immutable table or the retention period for rows within the immutable table. You cannot reduce the retention period.

Example : Modifying the Retention Period for an Immutable Table

The following statement modifies the definition of the immutable table *`imm_tab`* and specifies that it cannot be dropped if the newest row is less than 50 days old.

```
ALTER TABLE imm_tab NO DROP UNTIL 50 DAYS IDLE;
```

Example : Modifying the Retention Period for Immutable Table Rows

The following statement modifies the definition of the immutable table *`imm_tab`* and specifies that rows cannot be deleted until 120 days after they were created.

```
ALTER TABLE imm_tab NO DELETE UNTIL 120 DAYS AFTER
    INSERT;
```

blockchain_table_clauses

You can modify a table created using the keyword `BLOCKCHAIN` in the `ALTER TABLE` statement, and one or more of the *`blockchain_table_clauses`*.

See *`blockchain_table_clauses`* of CREATE TABLE for the full semantics of the clause.

You can add, drop, and rename a column in a V2 blockchain table.

Use the *`blockchain_system_chains_clause`* to configure the number of system chains in a blockchain table. The range of permissible values is 1 to 1024. For `ALTER TABLE`, you can increase or decrease the number of system chains per instance, but you cannot configure a number of system chains per instance that is less than the maximum number of a system chain already in the blockchain table.

You cannot use the *`blockchain_hash_and_data_format_clause`* of the *`blockchain_table_clauses`* in the `ALTER TABLE` statement.

Restrictions on All Versions of Blockchain Tables V1 and V2

You can use all the clauses of `ALTER TABLE` on a blockchain table except the following clauses:

- `DROP (SUB)PARTITION`
- `TRUNCATE (SUB)PARTITION`
- `EXCHANGE (SUB)PARTITION`
- `MODIFY TYPE`
- `RENAME TABLE`

Additional Restrictions on V1 Blockchain Tables

The following `ADD`, `DROP`, and `RENAME COLUMN` restrictions apply to V1 blockchain tables but not V2 blockchain tables:

- `RENAME COLUMN`
- `ADD COLUMN`
- `DROP COLUMN`

duplicated_table_refresh

Use this clause to specify fine-grained refresh rate control for a duplicated table when it is created with `CREATE TABLE`. You can also specify the refresh rate later with `ALTER TABLE`.

enable_disable_clause

The *`enable_disable_clause`* lets you specify whether and how Oracle Database should apply an integrity constraint. The `DROP` and `KEEP` clauses are valid only when you are disabling a unique or primary key constraint.

See Also:

The *enable_disable_clause* (in `CREATE TABLE`) for a complete description of this clause, including notes and restrictions that relate to this statement

TABLE LOCK

Oracle Database permits DDL operations on a table only if the table can be locked during the operation. Such table locks are not required during DML operations.

Note:

Table locks are not acquired on temporary tables.

- Specify `ENABLE TABLE LOCK` to enable table locks, thereby allowing DDL operations on the table. All currently executing transactions must commit or roll back before Oracle Database enables the table lock.

> **Note:**
>
> Oracle Database waits until active DML transactions in the database have completed before locking the table. Sometimes the resulting delay is considerable.

- Specify `DISABLE TABLE LOCK` to disable table locks, thereby preventing DDL operations on the table.

> **Note:**
>
> Parallel DML operations are not performed when the table lock of the target table is disabled.

ALL TRIGGERS

Use the `ALL TRIGGERS` clause to enable or disable all triggers associated with the table.

- Specify `ENABLE ALL TRIGGERS` to enable all triggers associated with the table. Oracle Database fires the triggers whenever their triggering condition is satisfied.

  To enable a single trigger, use the *`enable_clause`* of `ALTER TRIGGER`.

> **See Also:**
>
> CREATE TRIGGER , ALTER TRIGGER , and "Enabling Triggers: Example"

- Specify `DISABLE ALL TRIGGERS` to disable all triggers associated with the table. Oracle Database does not fire a disabled trigger even if the triggering condition is satisfied.

CONTAINER_MAP

Use the `CONTAINER_MAP` clause to enable or disable the table to be queried using a container map.

- Specify `ENABLE CONTAINER_MAP` to enable the table to be queried using a container map.
- Specify `DISABLE CONTAINER_MAP` to disable the table from being queried using a container map.

CONTAINERS_DEFAULT

Use the `CONTAINERS_DEFAULT` clause to enable or disable the table for the `CONTAINERS` clause.

- Specify `ENABLE CONTAINERS_DEFAULT` to enable the table for the `CONTAINERS` clause.
- Specify `DISABLE CONTAINERS_DEFAULT` to disable the table for the `CONTAINERS` clause.

Examples

Adding Constraints to Tables: Example

The following statements create a new table to manipulate data and display the information in the newly created table:

```
CREATE TABLE JOBS_Temp AS SELECT * FROM HR.JOBS;

SELECT * FROM JOBS_Temp WHERE MIN_SALARY < 3000;

JOB_ID       JOB_TITLE                   MIN_SALARY MAX_SALARY
---------- ----------------------------------- ---------- ----------
PU_CLERK   Purchasing Clerk                  2500    5500
ST_CLERK   Stock Clerk                      2008    5000
SH_CLERK   Shipping Clerk                  2500    5500
```

The following statement updates the column values to a higher value:

```
UPDATE JOBS_Temp SET MIN_SALARY = 2300 WHERE MIN_SALARY < 2010;
```

The following statement adds a constraint:

```
ALTER TABLE JOBS_Temp ADD CONSTRAINT chk_sal_min CHECK (MIN_SALARY >=2010);
```

The following statement displays the table information:

```
SELECT * FROM JOBS_Temp WHERE MIN_SALARY < 3000;

JOB_ID       JOB_TITLE                   MIN_SALARY MAX_SALARY
---------- ----------------------------------- ---------- ----------
PU_CLERK   Purchasing Clerk                  2500    5500
ST_CLERK   Stock Clerk                      2300    5000
SH_CLERK   Shipping Clerk                  2500    5500
```

The following statement displays the constraint:

```
SELECT CONSTRAINT_NAME FROM USER_CONSTRAINTS WHERE TABLE_NAME='JOBS_TEMP';

CONSTRAINT_NAME
--------------------------------------------------------------------------------
SYS_C008830
CHK_SAL_MIN
```

Adding and Modifying Precheck State Constraint: Example

The following statement create a product table with constraint state `PRECHECK` set on some columns:

```
CREATE TABLE product(
  id NUMBER NOT NULL PRIMARY KEY,
  name VARCHAR2(50),
  price NUMBER CHECK (mod(price,4) = 0 and 10 <> price) PRECHECK,
  color NUMBER CHECK (color >= 10 and color <=50 and mod(color,2) = 0)
    PRECHECK,
  description VARCHAR2(50) CHECK (length(description) <= 40) PRECHECK,
  constant NUMBER CHECK (constant=10) PRECHECK,
  CONSTRAINT TC1 CHECK (color > 0 AND price > 10) PRECHECK,
  CONSTRAINT TC2 CHECK (CATEGORY IN ('home', 'apparel') AND price > 10)
);
```

Add precheck to a new constraint

```
ALTER TABLE product MODIFY (name VARCHAR2(50) CHECK
  (regexp_like(name, '^Product')) PRECHECK);
```

Modify an existing constraint `TC2`:

```
ALTER TABLE product MODIFY CONSTRAINT TC2 PRECHECK;
```

Remove an exisiting precheck constraint on `TC1`:

```
ALTER TABLE product MODIFY CONSTRAINT TC1 NOPRECHECK;
```

Collection Retrieval: Example

The following statement modifies nested table column `ad_textdocs_ntab` in the sample table `sh.print_media` so that when queried it returns actual values instead of locators:

```
ALTER TABLE print_media MODIFY NESTED TABLE ad_textdocs_ntab
   RETURN AS VALUE;
```

Specifying Parallel Processing: Example

The following statement specifies parallel processing for queries to the sample table `oe.customers`:

```
ALTER TABLE customers
   PARALLEL;
```

Changing the State of a Constraint: Examples

The following statement places in `ENABLE VALIDATE` state an integrity constraint named `emp_manager_fk` in the `employees` table:

```
ALTER TABLE employees
   ENABLE VALIDATE CONSTRAINT emp_manager_fk
   EXCEPTIONS INTO exceptions;
```

Each row of the `employees` table must satisfy the constraint for Oracle Database to enable the constraint. If any row violates the constraint, then the constraint remains disabled. The database lists any exceptions in the table `exceptions`. You can also identify the exceptions in the `employees` table with the following statement:

```
SELECT e.*
   FROM employees e, exceptions ex
   WHERE e.rowid = ex.row_id
      AND ex.table_name = 'EMPLOYEES'
      AND ex.constraint = 'EMP_MANAGER_FK';
```

The following statement tries to place in `ENABLE NOVALIDATE` state two constraints on the `employees` table:

```
ALTER TABLE employees
   ENABLE NOVALIDATE PRIMARY KEY
   ENABLE NOVALIDATE CONSTRAINT emp_last_name_nn;
```

This statement has two `ENABLE` clauses:

- The first places a primary key constraint on the table in `ENABLE NOVALIDATE` state.
- The second places the constraint named `emp_last_name_nn` in `ENABLE NOVALIDATE` state.

In this case, Oracle Database enables the constraints only if both are satisfied by each row in the table. If any row violates either constraint, then the database returns an error and both constraints remain disabled.

Consider the foreign key constraint on the `location_id` column of the `departments` table, which references the primary key of the `locations` table. The following statement disables the primary key of the `locations` table:

```
ALTER TABLE locations
   MODIFY PRIMARY KEY DISABLE CASCADE;
```

The unique key in the `locations` table is referenced by the foreign key in the `departments` table, so you must specify `CASCADE` to disable the primary key. This clause disables the foreign key as well.

Creating an Exceptions Table for Index-Organized Tables: Example

The following example creates the `except_table` table to hold rows from the index-organized table `hr.countries` that violate the primary key constraint:

```
EXECUTE DBMS_IOT.BUILD_EXCEPTIONS_TABLE ('hr', 'countries', 'except_table');

ALTER TABLE countries
   ENABLE PRIMARY KEY
   EXCEPTIONS INTO except_table;
```

To specify an exception table, you must have the privileges necessary to insert rows into the table. To examine the identified exceptions, you must have the privileges necessary to query the exceptions table.

See Also:

INSERT and SELECT for information on the privileges necessary to insert rows into tables

Disabling a CHECK Constraint: Example

The following statement defines and disables a `CHECK` constraint on the `employees` table:

```
ALTER TABLE employees ADD CONSTRAINT check_comp
   CHECK (salary + (commission_pct*salary) <= 5000)
   DISABLE;
```

The constraint `check_comp` ensures that no employee's total compensation exceeds $5000. The constraint is disabled, so you can increase an employee's compensation above this limit.

Enabling Triggers: Example

The following statement enables all triggers associated with the `employees` table:

```
ALTER TABLE employees
   ENABLE ALL TRIGGERS;
```

Deallocating Unused Space: Example

The following statement frees all unused space for reuse in table `employees`, where the high water mark is above `MINEXTENTS`:

```
ALTER TABLE employees
    DEALLOCATE UNUSED;
```

Modifying the Collation of a Column for Fine-Grained Case-Insensitivity: Example

This example shows how to modify a column to be case-insensitive. First, create and populate table `students` as follows:

```
CREATE TABLE students (last_name VARCHAR2(20), id NUMBER);

INSERT INTO students VALUES('Dodd', 364);
INSERT INTO students VALUES('de Niro', 132);
INSERT INTO students VALUES('Vogel', 837);
INSERT INTO students VALUES('van der Kamp', 549);
INSERT INTO students VALUES('van Der Meer', 624);
```

The following statement returns column `last_name` in alphabetical order. Notice that the results are case-sensitive; lowercase letters are ordered after uppercase letters.

```
SELECT last_name, id
  FROM students
  ORDER BY last_name;

LAST_NAME                    ID
-------------------- ----------
Dodd                        364
Vogel                       837
de Niro                     132
van Der Meer                624
van der Kamp                549
```

The following statement changes the data-bound collation of column `last_name` to case-insensitive collation `BINARY_CI`:

```
ALTER TABLE students
  MODIFY (last_name COLLATE BINARY_CI);
```

The following statement again returns column `last_name` in alphabetical order. Notice that the results are now case-insensitive:

```
SELECT last_name, id
  FROM students
  ORDER BY last_name;

LAST_NAME                    ID
-------------------- ----------
de Niro                     132
Dodd                        364
van der Kamp                549
van Der Meer                624
Vogel                       837
```

Renaming a Column: Example

The following example renames the `credit_limit` column of the sample table `oe.customers` to `credit_amount`:

```
ALTER TABLE customers
   RENAME COLUMN credit_limit TO credit_amount;
```

Dropping a Column: Example

This statement illustrates the *`drop_column_clause`* with `CASCADE CONSTRAINTS`. Assume table `t1` is created as follows:

```
CREATE TABLE t1 (
   pk NUMBER PRIMARY KEY,
   fk NUMBER,
   c1 NUMBER,
   c2 NUMBER,
   CONSTRAINT ri FOREIGN KEY (fk) REFERENCES t1,
   CONSTRAINT ck1 CHECK (pk > 0 and c1 > 0),
   CONSTRAINT ck2 CHECK (c2 > 0)
);
```

An error will be returned for the following statements:

```
ALTER TABLE t1 DROP (pk);  -- pk is a parent key
ALTER TABLE t1 DROP (c1);  -- c1 is referenced by multicolumn
                           -- constraint ck1
```

Submitting the following statement drops column `pk`, the primary key constraint, the foreign key constraint, `ri`, and the check constraint, `ck1`:

```
ALTER TABLE t1 DROP (pk) CASCADE CONSTRAINTS;
```

If all columns referenced by the constraints defined on the dropped columns are also dropped, then `CASCADE CONSTRAINTS` is not required. For example, assuming that no other referential constraints from other tables refer to column `pk`, then it is valid to submit the following statement without the `CASCADE CONSTRAINTS` clause:

```
ALTER TABLE t1 DROP (pk, fk, c1);
```

Dropping Unused Columns: Example

The following statements create a new table to manipulate data and display the information in the newly created table:

```
CREATE TABLE JOBS_Temp AS SELECT * FROM HR.JOBS;

SELECT * FROM JOBS_Temp WHERE MAX_SALARY > 20000;

JOB_ID       JOB_TITLE                  MIN_SALARY MAX_SALARY
---------- ----------------------------------- ---------- ----------
AD_PRES    President                    20080      40000
AD_VP       Administration Vice President        15000      30000
SA_MAN       Sales Manager                10000      20080
```

The following statement adds two new columns:

```
ALTER TABLE JOBS_Temp ADD (DUMMY1 NUMBER(2), DUMMY2 NUMBER(2));
```

The following statements inserts values into the newly added columns:

```
INSERT INTO JOBS_Temp(JOB_ID, JOB_TITLE, DUMMY1, DUMMY2) VALUES ('D','DUMMY',10,20);

INSERT INTO JOBS_Temp(JOB_ID, JOB_TITLE, DUMMY1, DUMMY2) VALUES ('D','DUMMY',10,20)
```

The following statement sets the newly added columns to unused:

```
ALTER TABLE JOBS_TEMP SET UNUSED (DUMMY1, DUMMY2);
```

The following statement displays the count of unused columns:

```
SELECT * FROM USER_UNUSED_COL_TABS WHERE TABLE_NAME='JOBS_TEMP';

TABLE_NAM      COUNT
--------- ----------
JOBS_TEMP       2
```

The following statement drops the unused columns:

```
ALTER TABLE JOBS_TEMP DROP UNUSED COLUMNS;
```

The following statement displays the table information:

```
SELECT * FROM JOBS_TEMP;

JOB_ID       JOB_TITLE                   MIN_SALARY MAX_SALARY
---------- ----------------------------------- ---------- ----------
AD_PRES    President                    20080      40000
AD_VP       Administration Vice President        15000      30000
AD_ASST    Administration Assistant              3000     6000
FI_MGR       Finance Manager                  8200      16000
FI_ACCOUNT Accountant                      4200     9000
AC_MGR       Accounting Manager                  8200      16000
AC_ACCOUNT Public Accountant                  4200     9000
SA_MAN       Sales Manager                 10000      20080
SA_REP       Sales Representative               6000      12008
PU_MAN       Purchasing Manager                  8000      15000
PU_CLERK   Purchasing Clerk                  2500     5500
ST_MAN       Stock Manager                  5500     8500
ST_CLERK   Stock Clerk                     2008     5000
SH_CLERK   Shipping Clerk                  2500     5500
IT_PROG    Programmer                      4000      10000
MK_MAN       Marketing Manager                   9000      15000
MK_REP       Marketing Representative              4000     9000
HR_REP       Human Resources Representative          4000     9000
PR_REP       Public Relations Representative          4500      10500
D       DUMMY
D       DUMMY
```

Modifying Index-Organized Tables: Examples

This statement modifies the `INITRANS` parameter for the index segment of index-organized table `countries_demo`, which is based on `hr.countries`:

```
ALTER TABLE countries_demo INITRANS 4;
```

The following statement adds an overflow data segment to index-organized table `countries`:

```
ALTER TABLE countries_demo ADD OVERFLOW;
```

This statement modifies the `INITRANS` parameter for the overflow data segment of index-organized table `countries`:

```
ALTER TABLE countries_demo OVERFLOW INITRANS 4;
```

Splitting Table Partitions: Examples

The following statement splits the old partition `sales_q4_2000` in the sample table `sh.sales`, creating two new partitions, naming one `sales_q4_2000b` and reusing the name of the old partition for the other:

```
ALTER TABLE sales SPLIT PARTITION SALES_Q4_2000
   AT (TO_DATE('15-NOV-2000','DD-MON-YYYY'))
   INTO (PARTITION SALES_Q4_2000, PARTITION SALES_Q4_2000b);
```

The following statement splits the old partition `sales_q1_2002` into three new partitions `sales_jan_2002`, `sales_feb_2002`, and `sales_mar_2002`:

```
ALTER TABLE sales SPLIT PARTITION SALES_Q1_2002 INTO (
 PARTITION SALES_JAN_2002 VALUES LESS THAN (TO_DATE('01-FEB-2002','DD-MON-YYYY')),
 PARTITION SALES_FEB_2002 VALUES LESS THAN (TO_DATE('01-MAR-2002','DD-MON-YYYY')),
 PARTITION SALES_MAR_2002);
```

The following statements create a partitioned version of the pm.print_media table. The `LONG` column in the print_media table has been converted to LOB. The table is stored in tablespaces created in "Creating Oracle Managed Files: Examples". The object types underlying the `ad_textdocs_ntab` and `ad_header` columns are created in the script that creates the `pm` sample schema:

```
CREATE TABLE print_media_part (
    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 textdoc_nt
  PARTITION BY RANGE (product_id)
    (PARTITION p1 VALUES LESS THAN (100),
     PARTITION p2 VALUES LESS THAN (200));
```

The following statement splits partition `p2` of that table into partitions `p2a` and `p2b`:

```
ALTER TABLE print_media_part
   SPLIT PARTITION p2 AT (150) INTO
   (PARTITION p2a TABLESPACE omf_ts1
      LOB (ad_photo, ad_composite) STORE AS (TABLESPACE omf_ts2),
   PARTITION p2b
      LOB (ad_photo, ad_composite) STORE AS (TABLESPACE omf_ts2))
   NESTED TABLE ad_textdocs_ntab INTO (PARTITION nt_p2a, PARTITION nt_p2b);
```

In both partitions `p2a` and `p2b`, Oracle Database creates the LOB segments for columns `ad_photo` and `ad_composite` in tablespace `omf_ts2`. The LOB segments for the remaining columns in partition p2a are stored in tablespace omf_ts1. The LOB segments for the remaining columns in partition p2b remain in the tablespaces in which they resided prior to this `ALTER` statement. However, the database creates new segments for all the LOB data and LOB index segments, even if they are not moved to a new tablespace.

The database also creates new segments for nested table column `ad_textdocs_ntab`. The storage tables is those new segments are `nt_p2a` and `nt_p2b`.

Merging Two Table Partitions: Example

The following statement merges back into one partition the partitions created in "Splitting Table Partitions: Examples":

```
ALTER TABLE sales
   MERGE PARTITIONS sales_q4_2000, sales_q4_2000b
   INTO PARTITION sales_q4_2000;
```

The next statement reverses the example in "Splitting Table Partitions: Examples":

```
ALTER TABLE print_media_part
   MERGE PARTITIONS p2a, p2b INTO PARTITION p2ab TABLESPACE example
   NESTED TABLE ad_textdocs_ntab STORE AS nt_p2ab;
```

Merging Four Adjacent Range Partitions: Example

The following statement merges four adjacent range partitions, `sales_q1_2000`, `sales_q2_2000`, `sales_q3_2000`, and `sales_q4_2000` into one partition `sales_all_2000`:

```
ALTER TABLE sales
  MERGE PARTITIONS sales_q1_2000 TO sales_q4_2000
  INTO PARTITION sales_all_2000;
```

Adding a Table Partition with a LOB and Nested Table Storage: Examples

The following statement adds a partition `p3` to the `print_media_part` table (see preceding example) and specifies storage characteristics for the `BLOB`, `CLOB`, and nested table columns of that table:

```
ALTER TABLE print_media_part ADD PARTITION p3 VALUES LESS THAN (400)
  LOB(ad_photo, ad_composite) STORE AS (TABLESPACE omf_ts1)
  LOB(ad_sourcetext, ad_finaltext) STORE AS (TABLESPACE omf_ts2)
  NESTED TABLE ad_textdocs_ntab STORE AS nt_p3;
```

The LOB data and LOB index segments for columns `ad_photo` and `ad_composite` in partition `p3` will reside in tablespace `omf_ts1`. The remaining attributes for these LOB columns will be inherited first from the table-level defaults, and then from the tablespace defaults.

The LOB data segments for columns `ad_source_text` and `ad_finaltext` will reside in the `omf_ts2` tablespace, and will inherit all other attributes first from the table-level defaults, and then from the tablespace defaults.

The partition for the storage table for nested table storage column `ad_textdocs_ntab` corresponding to partition `p3` of the base table is named `nt_p3` and inherits all other attributes first from the table-level defaults, and then from the tablespace defaults.

Adding Multiple Partitions to a Table: Example

The following statement adds three partitions to the table `print_media_part` created in "Splitting Table Partitions: Examples":

```
ALTER TABLE print_media_part ADD
  PARTITION p3 values less than (300),
  PARTITION p4 values less than (400),
  PARTITION p5 values less than (500);
```

Working with Default List Partitions: Example

The following statements use the list partitioned table created in "List Partitioning Example". The first statement splits the existing default partition into a new `south` partition and a default partition:

```
ALTER TABLE list_customers SPLIT PARTITION rest
   VALUES ('MEXICO', 'COLOMBIA')
   INTO (PARTITION south, PARTITION rest);
```

The next statement merges the resulting default partition with the `asia` partition:

```
ALTER TABLE list_customers
   MERGE PARTITIONS asia, rest INTO PARTITION rest;
```

The next statement re-creates the `asia` partition by splitting the default partition:

```
ALTER TABLE list_customers SPLIT PARTITION rest
   VALUES ('CHINA', 'THAILAND')
   INTO (PARTITION asia, PARTITION rest);
```

Dropping a Table Partition: Example

The following statement drops partition p3 created in "Adding a Table Partition with a LOB and Nested Table Storage: Examples":

```
ALTER TABLE print_media_part DROP PARTITION p3;
```

Exchanging Table Partitions: Example

This example creates the table exchange_table with the same structure as the partitions of the list_customers table created in "List Partitioning Example". It then replaces partition rest of table list_customers with table exchange_table without exchanging local index partitions with corresponding indexes on exchange_table and without verifying that data in exchange_table falls within the bounds of partition rest:

```
CREATE TABLE exchange_table (
   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));

ALTER TABLE list_customers
   EXCHANGE PARTITION rest WITH TABLE exchange_table
   WITHOUT VALIDATION;
```

Modifying Table Partitions: Examples

The following statement marks all the local index partitions corresponding to the asia partition of the list_customers table UNUSABLE:

```
ALTER TABLE list_customers MODIFY PARTITION asia
   UNUSABLE LOCAL INDEXES;
```

The following statement rebuilds all the local index partitions that were marked UNUSABLE:

```
ALTER TABLE list_customers MODIFY PARTITION asia
   REBUILD UNUSABLE LOCAL INDEXES;
```

Moving Table Partitions: Example

The following statement moves partition p2b (from "Splitting Table Partitions: Examples") to tablespace omf_ts1:

```
ALTER TABLE print_media_part
   MOVE PARTITION p2b TABLESPACE omf_ts1;
```

Renaming Table Partitions: Examples

The following statement renames a partition of the sh.sales table:

```
ALTER TABLE sales RENAME PARTITION sales_q4_2003 TO sales_currentq;
```

Truncating Table Partitions: Example

The following statement uses the print_media_demo table created in "Partitioned Table with LOB Columns Example". It deletes all the data in the p1 partition and deallocates the freed space:

```
ALTER TABLE print_media_demo
   TRUNCATE PARTITION p1 DROP STORAGE;
```

Updating Global Indexes: Example

The following statement splits partition `sales_q1_2000` of the sample table `sh.sales` and updates any global indexes defined on it:

```
ALTER TABLE sales SPLIT PARTITION sales_q1_2000
   AT (TO_DATE('16-FEB-2000','DD-MON-YYYY'))
   INTO (PARTITION q1a_2000, PARTITION q1b_2000)
   UPDATE GLOBAL INDEXES;
```

Updating Partitioned Indexes: Example

The following statement splits partition `costs_Q4_2003` of the sample table `sh.costs` and updates the local index defined on it. It uses the tablespaces created in "Creating Basic Tablespaces: Examples".

```
CREATE INDEX cost_ix ON costs(channel_id) LOCAL;

ALTER TABLE costs
  SPLIT PARTITION costs_q4_2003 at
    (TO_DATE('01-Nov-2003','dd-mon-yyyy'))
    INTO (PARTITION c_p1, PARTITION c_p2)
  UPDATE INDEXES (cost_ix (PARTITION c_p1 tablespace tbs_02,
                           PARTITION c_p2 tablespace tbs_03));
```

Specifying Object Identifiers: Example

The following statements create an object type, a corresponding object table with a primary-key-based object identifier, and a table having a user-defined `REF` column:

```
CREATE TYPE emp_t AS OBJECT (empno NUMBER, address CHAR(30));

CREATE TABLE emp OF emp_t (
   empno PRIMARY KEY)
   OBJECT IDENTIFIER IS PRIMARY KEY;

CREATE TABLE dept (dno NUMBER, mgr_ref REF emp_t SCOPE is emp);
```

The next statements add a constraint and a user-defined `REF` column, both of which reference table `emp`

```
ALTER TABLE dept ADD CONSTRAINT mgr_cons FOREIGN KEY (mgr_ref)
   REFERENCES emp;
ALTER TABLE dept ADD sr_mgr REF emp_t REFERENCES emp;
```

Adding a Table Column: Example

The following statement adds to the `countries` table a column named `duty_pct` of data type `NUMBER` and a column named `visa_needed` of data type `VARCHAR2` with a size of 3 and a `CHECK` integrity constraint:

```
ALTER TABLE countries
   ADD (duty_pct     NUMBER(2,2)  CHECK (duty_pct < 10.5),
        visa_needed  VARCHAR2(3));
```

Adding a Virtual Table Column: Example

The following statement adds to a copy of the `hr.employees` table a column named `income`, which is a combination of salary plus commission. Both salary and commission are `NUMBER` columns, so the database creates the virtual column as a `NUMBER` column even though the data type is not specified in the statement:

```
CREATE TABLE emp2 AS SELECT * FROM employees;

ALTER TABLE emp2 ADD (income AS (salary + (salary*commission_pct)));
```

Modifying Table Columns: Examples

The following statement increases the size of the `duty_pct` column:

```
ALTER TABLE countries
   MODIFY (duty_pct NUMBER(3,2));
```

Because the `MODIFY` clause contains only one column definition, the parentheses around the definition are optional.

The following statement changes the values of the `PCTFREE` and `PCTUSED` parameters for the `employees` table to 30 and 60, respectively:

```
ALTER TABLE employees
   PCTFREE 30
   PCTUSED 60;
```

Modifying Storage Attributes for a Table

The following statement creates a table named `JOBS_TEMP` by using the existing `JOBS` table:

```
CREATE TABLE JOBS_TEMP AS SELECT * FROM HR.JOBS;
```

The following statement queries the `USER_TABLES` table for storage parameters:

```
SELECT initial_extent,
       next_extent,
       min_extents,
       max_extents,
       pct_increase,
       blocks,
       sample_size
FROM   user_tables
WHERE  table_name = 'JOBS_TEMP';

INITIAL_EXTENT NEXT_EXTENT MIN_EXTENTS MAX_EXTENTS PCT_INCREASE     BLOCKS
SAMPLE_SIZE
-------------- ----------- ----------- ----------- ------------ ----------
-----------
         65536     1048576           1  2147483645
1           19
```

The following statement alters the `JOBS_TEMP` table with new storage parameters:

```
ALTER TABLE JOBS_TEMP MOVE
      STORAGE ( INITIAL 20K
                NEXT 40K
                MINEXTENTS 2
                MAXEXTENTS 20
                PCTINCREASE 0 )
      TABLESPACE USERS;
```

The following statement queries the `USER_TABLES` table for the new storage parameters:

```
SELECT initial_extent,
       next_extent,
       min_extents,
       max_extents,
       pct_increase,
       blocks,
       sample_size
FROM   user_tables
WHERE  table_name = 'JOBS_TEMP';

INITIAL_EXTENT NEXT_EXTENT MIN_EXTENTS MAX_EXTENTS PCT_INCREASE     BLOCKS 
SAMPLE_SIZE
-------------- ----------- ----------- ----------- ------------ ----------
-----------
         65536       40960           1  2147483645
1           19
```

Adding, Altering, Renaming and Dropping Table Columns: Example

The following statements create a new table to manipulate data and display the information in the newly created table:

```
CREATE TABLE JOBS_Temp AS SELECT * FROM HR.JOBS;

SELECT * FROM JOBS_Temp WHERE MAX_SALARY > 30000;

JOB_ID      JOB_TITLE                  MIN_SALARY MAX_SALARY
---------- ----------------------------------- ---------- ----------
AD_PRES    President                       20080      40000
```

The following statement modifies an existing column definition:

```
ALTER TABLE JOBS_Temp MODIFY(JOB_TITLE VARCHAR2(100));
```

The following statement adds two new columns to the table:

```
ALTER TABLE JOBS_Temp ADD (BONUS NUMBER (7,2), COMM NUMBER (5,2), DUMMY NUMBER(2));
```

The following statement displays the newly added columns:

```
SELECT JOB_ID, BONUS, COMM, DUMMY FROM JOBS_Temp WHERE MAX_SALARY > 20000;

JOB_ID        BONUS        COMM      DUMMY
---------- ---------- ---------- ----------
AD_PRES
AD_VP
SA_MAN
```

The following statements rename an existing column and display the modified column:

```
ALTER TABLE JOBS_Temp RENAME COLUMN COMM TO COMMISSION;

SELECT JOB_ID, COMMISSION FROM JOBS_Temp WHERE MAX_SALARY > 20000;

JOB_ID      COMMISSION
---------- ----------
AD_PRES
```

```
AD_VP
SA_MAN
```

The following statement drops a single column from the table:

```
ALTER TABLE JOBS_Temp DROP COLUMN DUMMY;
```

The following statement drops multiple columns from the table:

```
ALTER TABLE JOBS_Temp DROP (BONUS, COMMISSION);
```

Data Encryption: Examples

The following statement encrypts the salary column of the `hr.employees` table using the encryption algorithm `AES256`. As described in "Semantics" above, you must first enable Transparent Data Encryption:

```
ALTER TABLE employees
   MODIFY (salary ENCRYPT USING 'AES256' 'NOMAC');
```

The following statement adds a new encrypted column `online_acct_pw` to the `oe.customers` table, using the default encryption algorithm `AES192`. Specifying `NO SALT` will allow a B-tree index to be created on the column, if desired.

```
ALTER TABLE customers
   ADD (online_acct_pw VARCHAR2(8) ENCRYPT 'NOMAC' NO SALT);
```

The following example decrypts the customer.online_acct_pw column:

```
ALTER TABLE customers
   MODIFY (online_acct_pw DECRYPT);
```

Allocating Extents: Example

The following statement allocates an extent of 5 kilobytes for the `employees` table and makes it available to instance 4:

```
ALTER TABLE employees
  ALLOCATE EXTENT (SIZE 5K INSTANCE 4);
```

Because this statement omits the `DATAFILE` parameter, Oracle Database allocates the extent in one of the data files belonging to the tablespace containing the table.

Specifying a Default Column Value: Examples

This statement modifies the `min_price` column of the `product_information` table so that it has a default value of 10:

```
ALTER TABLE product_information
  MODIFY (min_price DEFAULT 10);
```

If you subsequently add a new row to the `product_information` table and do not specify a value for the `min_price` column, then the value of the `min_price` column is automatically 10:

```
INSERT INTO product_information (product_id, product_name,
   list_price)
   VALUES (300, 'left-handed mouse', 40.50);

SELECT product_id, product_name, list_price, min_price
    FROM product_information
    WHERE product_id = 300;

PRODUCT_ID PRODUCT_NAME         LIST_PRICE  MIN_PRICE
```

```
---------- -------------------- ---------- ----------
       300 left-handed mouse          40.5         10
```

To discontinue previously specified default values, so that they are no longer automatically inserted into newly added rows, replace the values with `NULL`, as shown in this statement:

```
ALTER TABLE product_information
   MODIFY (min_price DEFAULT NULL);
```

The `MODIFY` clause need only specify the column name and the modified part of the definition, rather than the entire column definition. This statement has no effect on any existing values in existing rows.

The following example adds a column defined with `DEFAULT ON NULL` to a table. The `DEFAULT` column value includes the sequence pseudocolumn `NEXTVAL`.

Create sequence `s1` and table `t1` as follows:

```
CREATE SEQUENCE s1 START WITH 1;

CREATE TABLE t1 (name VARCHAR2(10));
INSERT INTO t1 VALUES('Kevin');
INSERT INTO t1 VALUES('Julia');
INSERT INTO t1 VALUES('Ryan');
```

Add column `id`, which defaults to `s1.NEXTVAL`. The default column value for `id` is assigned to each existing row in the table. The order in which `s1.NEXTVAL` is assigned to each row is nondeterministic.

```
ALTER TABLE t1 ADD (id NUMBER DEFAULT ON NULL s1.NEXTVAL NOT NULL);

SELECT id, name FROM t1 ORDER BY id;

        ID NAME
---------- ----------
         1 Kevin
         2 Julia
         3 Ryan
```

If you subsequently add a new row to the table and specify a NULL value for the `id` column, then the `DEFAULT ON NULL` expression `s1.NEXTVAL` is inserted.

```
INSERT INTO t1(id, name) VALUES(NULL, 'Sean');

SELECT id, name FROM t1 ORDER BY id;

        ID NAME
---------- ----------
         1 Kevin
         2 Julia
         3 Ryan
         4 Sean
```

Adding a Constraint to an XMLType Table: Example

The following example adds a primary key constraint to the `xwarehouses` table, created in "XMLType Examples":

```
ALTER TABLE xwarehouses
   ADD (PRIMARY KEY(XMLDATA."WarehouseID"));
```

Refer to XMLDATA Pseudocolumn for information about this pseudocolumn.

Renaming Constraints: Example

The following statement renames the `cust_fname_nn` constraint on the sample table `oe.customers` to `cust_firstname_nn`:

```
ALTER TABLE customers RENAME CONSTRAINT cust_fname_nn
   TO cust_firstname_nn;
```

Dropping Constraints: Examples

The following statement drops the primary key of the `departments` table:

```
ALTER TABLE departments
    DROP PRIMARY KEY CASCADE;
```

If you know that the name of the `PRIMARY KEY` constraint is `pk_dept`, then you could also drop it with the following statement:

```
ALTER TABLE departments
    DROP CONSTRAINT pk_dept CASCADE;
```

The `CASCADE` clause causes Oracle Database to drop any foreign keys that reference the primary key.

The following statement drops the unique key on the `email` column of the `employees` table:

```
ALTER TABLE employees
    DROP UNIQUE (email);
```

The `DROP` clause in this statement omits the `CASCADE` clause. Because of this omission, Oracle Database does not drop the unique key if any foreign key references it.

LOB Columns: Examples

The following statement adds `CLOB` column `resume` to the `employee` table and specifies LOB storage characteristics for the new column:

```
ALTER TABLE employees ADD (resume CLOB)
  LOB (resume) STORE AS resume_seg (TABLESPACE example);
```

To modify the LOB column `resume` to use caching, enter the following statement:

```
ALTER TABLE employees MODIFY LOB (resume) (CACHE);
```

The following statement adds a SecureFiles `CLOB` column `resume` to the `employee` table and specifies LOB storage characteristics for the new column. SecureFiles LOBs must be stored in tablespaces with automatic segment-space management. Therefore, the LOB data in this example is stored in the `auto_seg_ts` tablespace, which was created in "Specifying Segment Space Management for a Tablespace: Example":

```
ALTER TABLE employees ADD (resume CLOB)
LOB (resume) STORE AS SECUREFILE resume_seg (TABLESPACE auto_seg_ts);
```

To modify the LOB column `resume` so that it does not use caching, enter the following statement:

```
ALTER TABLE employees MODIFY LOB (resume) (NOCACHE);
```

Nested Tables: Examples

The following statement adds the nested table column `skills` to the `employee` table:

```
ALTER TABLE employees ADD (skills skill_table_type)
    NESTED TABLE skills STORE AS nested_skill_table;
```

You can also modify nested table storage characteristics. Use the name of the storage table specified in the *`nested_table_col_properties`* to make the modification. You cannot query or perform DML statements on the storage table. Use the storage table only to modify the nested table column storage characteristics.

The following statement creates table `vet_service` with nested table column `client` and storage table `client_tab`. Nested table `client_tab` is modified to specify constraints:

```
CREATE TYPE pet_t AS OBJECT
   (pet_id NUMBER, pet_name VARCHAR2(10), pet_dob DATE);
/

CREATE TYPE pet AS TABLE OF pet_t;
/

CREATE TABLE vet_service (vet_name VARCHAR2(30),
                          client   pet)
  NESTED TABLE client STORE AS client_tab;

ALTER TABLE client_tab ADD UNIQUE (pet_id);
```

The following statement alters the storage table for a nested table of `REF` values to specify that the `REF` is scoped:

```
CREATE TYPE emp_t AS OBJECT (eno number, ename char(31));
CREATE TYPE emps_t AS TABLE OF REF emp_t;
CREATE TABLE emptab OF emp_t;
CREATE TABLE dept (dno NUMBER, employees emps_t)
   NESTED TABLE employees STORE AS deptemps;
ALTER TABLE deptemps ADD (SCOPE FOR (COLUMN_VALUE) IS emptab);
```

Similarly, to specify storing the `REF` with rowid:

```
ALTER TABLE deptemps ADD (REF(column_value) WITH ROWID);
```

In order to execute these `ALTER TABLE` statements successfully, the storage table `deptemps` must be empty. Also, because the nested table is defined as a table of scalar values (`REF` values), Oracle Database implicitly provides the column name `COLUMN_VALUE` for the storage table.

See Also:

- CREATE TABLE for more information about nested table storage
- *Oracle Database Object-Relational Developer's Guide* for more information about nested tables

REF Columns: Examples

The following statement creates an object type `dept_t` and then creates table `staff`:

```
CREATE TYPE dept_t AS OBJECT
   (deptno NUMBER, dname VARCHAR2(20));
/
```

```
CREATE TABLE staff
   (name   VARCHAR2(100),
    salary NUMBER,
    dept   REF dept_t);
```

An object table `offices` is created as:

```
CREATE TABLE offices OF dept_t;
```

The `dept` column can store references to objects of `dept_t` stored in any table. If you would like to restrict the references to point only to objects stored in the `departments` table, then you could do so by adding a scope constraint on the `dept` column as follows:

```
ALTER TABLE staff
    ADD (SCOPE FOR (dept) IS offices);
```

The preceding `ALTER TABLE` statement will succeed only if the `staff` table is empty.

If you want the `REF` values in the `dept` column of `staff` to also store the rowids, then issue the following statement:

```
ALTER TABLE staff
   ADD (REF(dept) WITH ROWID);
```

Unpacked Storage in ANYDATA Columns: Example

This example creates a table with an `ANYDATA` column, stores opaque data types in the `ANYDATA` column using unpacked storage, and then queries the data types. This example assumes that you are connected to the database as user `hr`.

Create table `t1`, which contains a `NUMBER` column `n` and an `ANYDATA` column `x`:

```
CREATE TABLE t1 (n NUMBER, x ANYDATA);
```

Create an object type `clob_typ`, which contains a `CLOB` attribute:

```
CREATE OR REPLACE TYPE clob_typ AS OBJECT (c clob);
/
```

Enable unpacked storage of the opaque data types `XMLType` and `clob_typ` in `ANYDATA` column `x` of table `t1`:

```
ALTER TABLE t1 MODIFY OPAQUE TYPE x STORE (XMLType, clob_typ) UNPACKED;
```

Insert `XMLType` and `clob_typ` objects into table `t1`. These types will use unpacked storage:

```
INSERT INTO t1
  VALUES(1, anydata.convertobject(XMLType('<Test>This is test XML</Test>')));

INSERT INTO t1
  VALUES(2, anydata.convertobject(clob_typ(TO_CLOB('This is a test CLOB'))));
```

Query table `t1` to view the names of the types stored in `ANYDATA` column `x`:

```
SELECT t1.*, anydata.getTypeName(t1.x) typename FROM t1;

    N X()                  TYPENAME
----- -------------------- --------------------
    1 ANYDATA()            SYS.XMLTYPE
    2 ANYDATA()            HR.CLOB_TYP
```

Create functions that allow you to query the values stored in the `XMLType` and `clob_typ` data types:

```
CREATE FUNCTION get_xmltype (ad IN ANYDATA) RETURN VARCHAR2 AS
      rtn_val PLS_INTEGER;
      my_xmltype XMLType;
      string_val VARCHAR2(30);
   BEGIN
      rtn_val := ad.getObject(my_xmltype);
      string_val := my_xmltype.getstringval();
      return (string_val);
   END;
/

CREATE FUNCTION get_clob_typ (ad IN ANYDATA) RETURN VARCHAR2 AS
      rtn_val PLS_INTEGER;
      my_clob_typ clob_typ;
      string_val VARCHAR2(30);
   BEGIN
      rtn_val := ad.getObject(my_clob_typ);
      string_val := (my_clob_typ.c);
      return (string_val);
   END;
/
```

Query table `t1` to view the values stored in each data type in `ANYDATA` column `x`:

```
SELECT t1.*, anydata.getTypeName(t1.x) typename,
  CASE
    WHEN anydata.gettypename(t1.x) = 'SYS.XMLTYPE' THEN get_xmltype(t1.x)
    WHEN anydata.gettypename(t1.x) = 'HR.CLOB_TYP' THEN get_clob_typ(t1.x)
  END string_value
FROM t1;

    N X()                  TYPENAME             STRING_VALUE
----- -------------------- -------------------- -------------------------------
    1 ANYDATA()            SYS.XMLTYPE          <Test>This is test XML</Test>
    2 ANYDATA()            HR.CLOB_TYP          This is a test CLOB
```

Additional Examples

For examples of defining integrity constraints with the `ALTER TABLE` statement, see the *constraint*.

For examples of changing the storage parameters of a table, see the storage_clause.

Add and Drop Annotations at the Table Level

The following examples use `table1` :

```
CREATE TABLE table1 (T NUMBER) ANNOTATIONS(Operations 'Sort', Hidden);
```

The following example drops all annotations from `table1`:

```
ALTER TABLE table1 ANNOTATIONS(DROP Operations, DROP Hidden);
```

The following example adds a new annotation `Operations` with a JSON value:

```
ALTER TABLE table1 ANNOTATIONS(ADD Operations '["Sort", "Group"]');
```

Add and Drop Annotations at the Column Level

The following example adds a new `Identity` annotation for column `T` of `table1`:

```
ALTER TABLE table1 MODIFY T ANNOTATIONS(Identity 'ID');
```

The following example adds `Hidden`, and drops `Identity`:

```
ALTER TABLE table1 MODIFY T ANNOTATIONS(ADD Hidden, DROP Identity);
```

Operations on Directory-Based Partitioned Table

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
   );
```

The following two examples use the table `departments` above for operations add and split.

Add Partitions to a Table Partitioned by Directory

```
ALTER TABLE departments ADD
    PARTITION p_3 TABLESPACE tbs3,
    PARTITION p_4 TABLESPACE tbs4;
```

Split Partitions of a Table Partitioned by Directory

```
ALTER TABLE departments
  SPLIT PARTITION p_1 INTO
   (PARTITION p_1 TABLESPACE tbs1,
    PARTITION p_3 TABLESPACE tbs3)
    UPDATE INDEXES;
```

ALTER TABLESPACE

Purpose

Use the `ALTER TABLESPACE` statement to alter an existing tablespace or one or more of its data files or temp files.

You cannot use this statement to convert a dictionary-managed tablespace to a locally managed tablespace. For that purpose, use the `DBMS_SPACE_ADMIN` package, which is documented in *Oracle Database PL/SQL Packages and Types Reference*.

See Also:

Oracle Database Administrator's Guide and CREATE TABLESPACE for information on creating a tablespace

Prerequisites

To alter the SYSAUX tablespace, you must have the SYSDBA system privilege.

If you have the ALTER TABLESPACE system privilege, then you can perform any ALTER TABLESPACE operation. If you have the MANAGE TABLESPACE system privilege, then you can only perform the following operations:

- Take a tablespace online or offline
- Begin or end a backup
- Make a tablespace read only or read write
- Change the state of a tablespace to PERMANENT or TEMPORARY
- Set the default logging mode of a tablespace to LOGGING or NOLOGGING
- Put a tablespace in force logging mode or take it out of force logging mode
- Rename a tablespace or a tablespace data file
- Specify RETENTION GUARANTEE or RETENTION NOGUARANTEE for an undo tablespace
- Resize a data file for a tablespace
- Enable or disable autoextension of a data file for a tablespace
- Shrink the amount of space a temporary tablespace or a temp file is taking

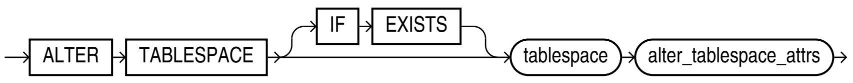
Before you can make a tablespace read only, the following conditions must be met:

- The tablespace must be online.
- The tablespace must not contain any active rollback segments. For this reason, the SYSTEM tablespace can never be made read only, because it contains the SYSTEM rollback segment. Additionally, because the rollback segments of a read-only tablespace are not accessible, Oracle recommends that you drop the rollback segments before you make a tablespace read only.
- The tablespace must not be involved in an open backup, because the end of a backup updates the header file of all data files in the tablespace.

Performing this function in restricted mode may help you meet these restrictions, because only users with RESTRICTED SESSION system privilege can be logged on.

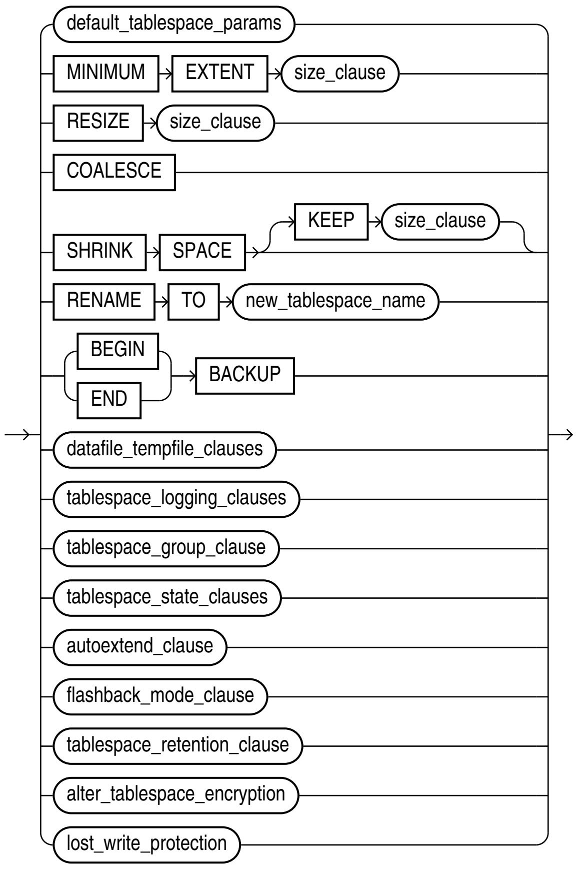
Syntax

***alter_tablespace*::=**



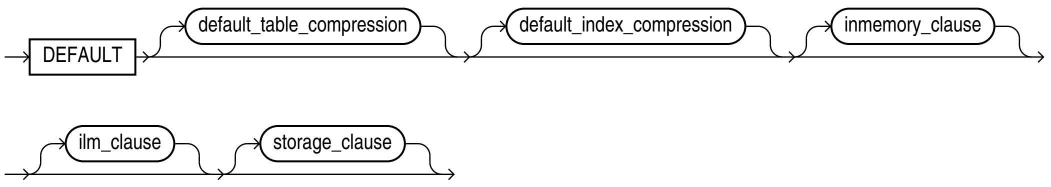
(*alter_tablespace_attrs::=*)

***alter_tablespace_attrs*::=**



(*default_tablespace_params*::=, *size_clause*::=, *datafile_tempfile_clauses*::=, *tablespace_logging_clauses*::=, *tablespace_group_clause*::=, *tablespace_state_clauses*::=, *autoextend_clause*::=, *flashback_mode_clause*::=, *tablespace_retention_clause*::=, *alter_tablespace_encryption*::=, *lost_write_protection*::=)

***default_tablespace_params*::=**

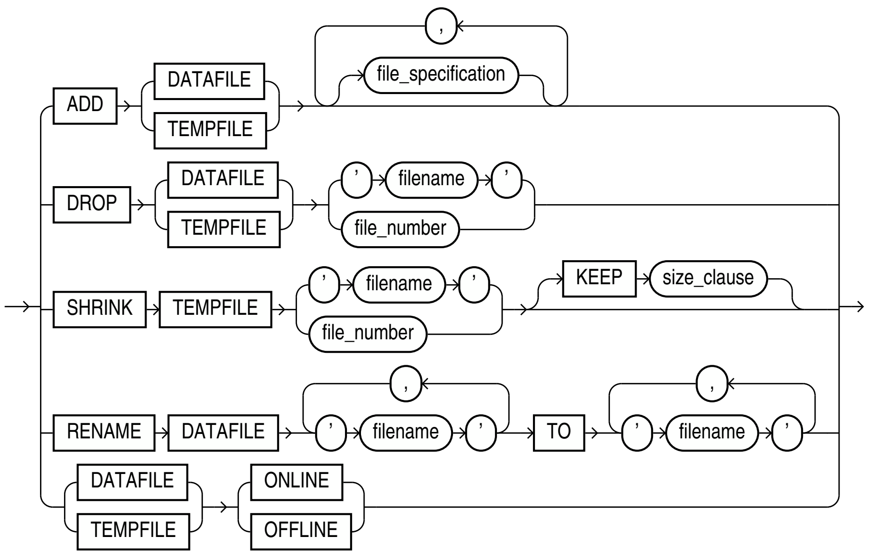


(*default_table_compression*::=—part of `CREATE TABLESPACE`, *default_index_compression*::=—part of `CREATE TABLESPACE`, *inmemory_clause*::=—part of `CREATE TABLESPACE`, *ilm_clause*::=—part of `ALTER TABLE`, *storage_clause*::=)

Note:

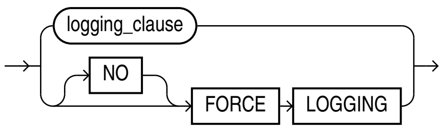
If you specify the `DEFAULT` clause, then you must specify at least one of the clauses *`default_table_compression`*, *`default_index_compression`*, *`inmemory_clause`*, *`ilm_clause`*, or *`storage_clause`*.

***datafile_tempfile_clauses*::=**



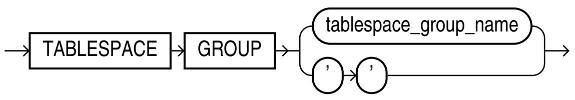
(*file_specification*::=).

***tablespace_logging_clauses*::=**

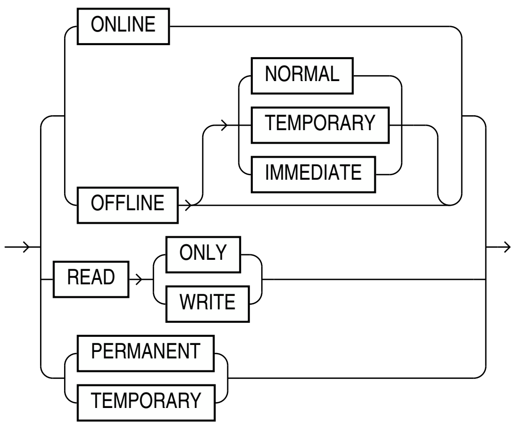


(*logging_clause*::=)

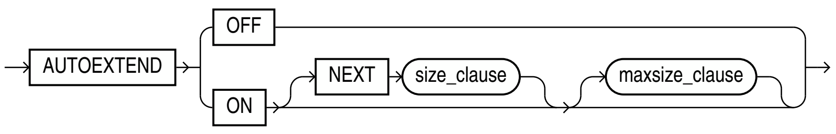
***tablespace_group_clause*::=**



tablespace_state_clauses::=

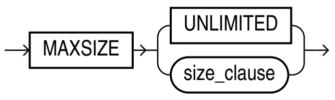


autoextend_clause::=



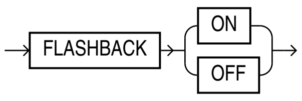
(*size_clause::=*)

maxsize_clause::=

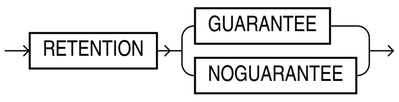


(*size_clause::=*)

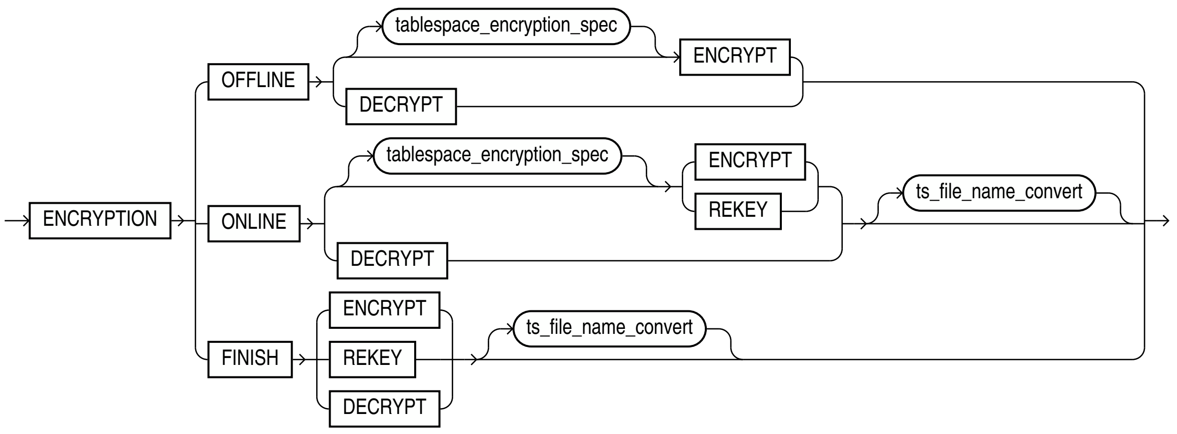
flashback_mode_clause::=



tablespace_retention_clause::=

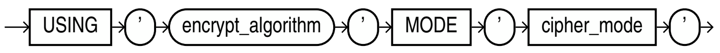


***alter_tablespace_encryption*::=**

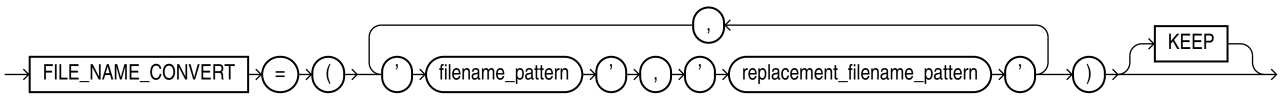


(*tablespace_encryption_spec*::=, *ts_file_name_convert*::=)

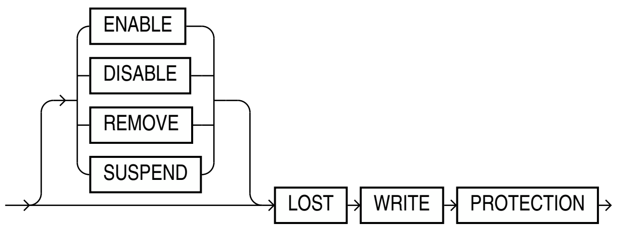
***tablespace_encryption_spec*::=**



***ts_file_name_convert*::=**



***lost_write_protection*::=**



Semantics

IF NOT EXISTS

Specify `IF EXISTS` to alter an existing tablespace.

Specifying `IF NOT EXISTS` with `ALTER` results in error: Incorrect `IF EXISTS` clause for `ALTER/DROP` statement.

tablespace

Specify the name of the tablespace to be altered.

Restrictions on Altering Tablespaces

Altering tablespaces is subject to the following restrictions:

- If *`tablespace`* is an undo tablespace, then the only other clauses you can specify in this statement are `ADD DATAFILE`, `RENAME DATAFILE`, `RENAME TO` (renaming the tablespace), `DATAFILE` ... `ONLINE`, `DATAFILE` ... `OFFLINE`, `BEGIN BACKUP`, and `END BACKUP`.
- You cannot make the `SYSTEM` tablespace read only or temporary and you cannot take it offline.
- For locally managed temporary tablespaces, the only clause you can specify in this statement is the `ADD` clause.

See Also:

Oracle Database Administrator's Guide for information on automatic undo management and undo tablespaces

alter_tablespace_attrs

Use the *`alter_tablespace_attrs`* clauses to change the attributes of the tablespace.

default_tablespace_params

This clause lets you specify new default parameters for the tablespace. The new default parameters apply to objects subsequently created in the tablespace.

The clauses *`default_table_compression`*, *`default_index_compression`*, *`inmemory_clause`*, *`ilm_clause`*, and *`storage_clause`* have the same semantics in `CREATE TABLESPACE` and `ALTER TABLESPACE`. For complete information on these clauses, refer to the *default_tablespace_params* clause in the documentation on `CREATE TABLESPACE`.

MINIMUM EXTENT

This clause is valid only for permanent dictionary-managed tablespaces. The `MINIMUM EXTENT` clause lets you control free space fragmentation in the tablespace by ensuring that 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`*.

Restriction on MINIMUM EXTENT

You cannot specify this clause for a locally managed tablespace or for a dictionary-managed temporary tablespace.

See Also:

size_clause for information about that clause, *Oracle Database Administrator's Guide* for more information about using `MINIMUM EXTENT` to control space fragmentation

RESIZE Clause

This clause is valid only for bigfile tablespaces, including shadow tablespaces which store lost write protection tracking data. It lets you increase or decrease the size of the single data file to an absolute size. Use `K`, `M`, `G`, or `T` to specify the size in kilobytes, megabytes, gigabytes, or terabytes, respectively.

To change the size of a newly added data file or temp file in smallfile tablespaces, use the `ALTER DATABASE` ... *`autoextend_clause`* (see *database_file_clauses*).

See Also:

BIGFILE | SMALLFILE for information on bigfile tablespaces

COALESCE

For each data file in the tablespace, this clause combines all contiguous free extents into larger contiguous extents.

SHRINK SPACE Clause

This clause is valid only for temporary tablespaces. It lets you reduce the amount of space the tablespace is taking. In the optional `KEEP` clause, the *`size_clause`* defines the lower bound that a tablespace can be shrunk to. It is the opposite of `MAXSIZE` for an autoextensible tablespace. If you omit the `KEEP` clause, then the database will attempt to shrink the tablespace as much as possible as long as other tablespace storage attributes are satisfied.

RENAME Clause

Use this clause to rename *`tablespace`*. This clause is valid only if *`tablespace`* and all its data files are online and the `COMPATIBLE` parameter is set to 10.0.0 or greater. You can rename both permanent and temporary tablespaces.

If *`tablespace`* is read only, then Oracle Database does not update the data file headers to reflect the new name. The alert log will indicate that the data file headers have not been updated.

Note:

If you re-create the control file, and if the data files that Oracle Database uses for this purpose are restored backups whose headers reflect the old tablespace name, then the re-created control file will also reflect the old tablespace name. However, after the database is fully recovered, the control file will reflect the new name.

If *`tablespace`* has been designated as the undo tablespace for any instance in an Oracle Real Application Clusters (Oracle RAC) environment, and if a server parameter file was used to start up the database, then Oracle Database changes the value of the `UNDO_TABLESPACE` parameter for that instance in the server parameter file (`SPFILE`) to reflect the new tablespace name. If a single-instance database is using a parameter file (pfile) instead of an spfile, then the database puts a message in the alert log advising the database administrator to change the value manually in the pfile.

Note:

The `RENAME` clause does not change the value of the `UNDO_TABLESPACE` parameter in the running instance. Although this does not affect the functioning of the undo tablespace, Oracle recommends that you issue the following statement to manually change the value of `UNDO_TABLESPACE` to the new tablespace name for the duration of the instance:

```
ALTER SYSTEM SET UNDO_TABLESPACE = new_tablespace_name SCOPE = MEMORY;
```

You only need to issue this statement once. If the `UNDO_TABLESPACE` parameter is set to the new tablespace name in the pfile or spfile, then the parameter will be set correctly when the instance is next restarted.

Restriction on Renaming Tablespaces

You cannot rename the `SYSTEM` or `SYSAUX` tablespaces.

BACKUP Clauses

Use these clauses to move all data files in a tablespace into or out of online (sometimes called hot) backup mode.

See Also:

- *Oracle Database Administrator's Guide* for information on restarting the database without media recovery
- `ALTER DATABASE` "BACKUP Clauses" for information on moving all data files in the database into and out of online backup mode
- `ALTER DATABASE` *alter_datafile_clause* for information on taking individual data files out of online backup mode

BEGIN BACKUP

Specify `BEGIN BACKUP` to indicate that an open backup is to be performed on the data files that make up this tablespace. This clause does not prevent users from accessing the tablespace. You must use this clause before beginning an open backup.

Restrictions on Beginning Tablespace Backup

Beginning tablespace backup is subject to the following restrictions:

- You cannot specify this clause for a read-only tablespace or for a temporary locally managed tablespace.
- While the backup is in progress, you cannot take the tablespace offline normally, shut down the instance, or begin another backup of the tablespace.

See Also:

"Backing Up Tablespaces: Examples"

END BACKUP

Specify `END BACKUP` to indicate that an online backup of the tablespace is complete. Use this clause as soon as possible after completing an online backup. Otherwise, if an instance failure or `SHUTDOWN ABORT` occurs, then Oracle Database assumes that media recovery (possibly requiring archived redo log) is necessary at the next instance startup.

Restriction on Ending Tablespace Backup

You cannot use this clause on a read-only tablespace.

datafile_tempfile_clauses

The tablespace file clauses let you add or modify a data file or temp file.

ADD Clause

Specify `ADD` to add to the tablespace a data file or temp file specified by *`file_specification`*. Use the *`datafile_tempfile_spec`* form of *`file_specification`* (see *file_specification*) to list regular data files and temp files in an operating system file system or to list Oracle Automatic Storage Management disk group files.

For locally managed temporary tablespaces, this is the only clause you can specify at any time.

If you omit *`file_specification`*, then Oracle Database creates an Oracle Managed File of 100M with `AUTOEXTEND` enabled.

You can add a data file or temp file to a locally managed tablespace that is online or to a dictionary managed tablespace that is online or offline. Ensure the file is not in use by another database.

Restriction on Adding Data Files and Temp Files

You cannot specify this clause for a bigfile (single-file) tablespace, as such a tablespace has only one data file or temp file.

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 , "Adding and Dropping Data Files and Temp Files: Examples", and "Adding an Oracle-managed Data File: Example"

DROP Clause

Specify `DROP` to drop from the tablespace an empty data file or temp file specified by *`filename`* or *`file_number`*. This clause causes the data file or temp file to be removed from the data dictionary and deleted from the operating system. The database must be open at the time this clause is specified.

The `ALTER TABLESPACE` ... `DROP TEMPFILE` statement is equivalent to specifying the `ALTER DATABASE TEMPFILE` ... `DROP INCLUDING DATAFILES`.

Restrictions on Dropping Files

To drop a data file or temp file, the data file or temp file:

- Must be empty.
- Cannot be the first data file that was created in the tablespace. In such cases, drop the tablespace instead.
- Cannot be in a read-only tablespace that was migrated from dictionary managed to locally managed. Dropping a data file from all other read-only tablespaces is supported.
- Cannot be offline.

See Also:

- `ALTER DATABASE` *alter_tempfile_clause* for additional information on dropping temp files
- *Oracle Database Administrator's Guide* for information on data file numbers and for guidelines on managing data files
- "Adding and Dropping Data Files and Temp Files: Examples"

SHRINK TEMPFILE Clause

This clause is valid only when altering a temporary tablespace. It lets you reduce the amount of space the specified temp file is taking. In the optional `KEEP` clause, the *`size_clause`* defines the lower bound that the temp file can be shrunk to. It is the opposite of `MAXSIZE` for an autoextensible tablespace. If you omit the `KEEP` clause, then the database will attempt to shrink the temp file as much as possible as long as other storage attributes are satisfied.

RENAME DATAFILE Clause

Specify `RENAME DATAFILE` to rename one or more of the tablespace data files. The database must be open, and you must take the tablespace offline before renaming it. Each *`filename`* must fully specify a data file using the conventions for filenames on your operating system.

This clause merely associates the tablespace with the new file rather than the old one. This clause does not actually change the name of the operating system file. You must change the name of the file through your operating system.

See Also:

"Moving and Renaming Tablespaces: Example"

ONLINE | OFFLINE Clauses

Use these clauses to take all data files or temp files in the tablespace offline or put them online. These clauses have no effect on the `ONLINE` or `OFFLINE` status of the tablespace itself.

The database must be mounted. If *`tablespace`* is `SYSTEM`, or an undo tablespace, or the default temporary tablespace, then the database must not be open.

tablespace_logging_clauses

Use these clauses to set or change the logging characteristics of the tablespace.

logging_clause

Specify `LOGGING` if you want logging of all tables, indexes, and partitions within the tablespace. The tablespace-level logging attribute can be overridden by logging specifications at the table, index, and partition levels.

When an existing tablespace logging attribute is changed by an `ALTER TABLESPACE` statement, all tables, indexes, and partitions created *after* the statement will have the new default logging attribute (which you can still subsequently override). The logging attribute of existing objects is not changed.

If the tablespace is in `FORCE LOGGING` mode, then you can specify `NOLOGGING` in this statement to set the default logging mode of the tablespace to `NOLOGGING`, but this will not take the tablespace out of `FORCE LOGGING` mode.

[NO] FORCE LOGGING

Use this clause to put the tablespace in force logging mode or take it out of force logging mode. The database must be open and in `READ WRITE` mode. Neither of these settings changes the default `LOGGING` or `NOLOGGING` mode of the tablespace.

Restriction on Force Logging Mode

You cannot specify `FORCE LOGGING` for an undo or a temporary tablespace.

See Also:

Oracle Database Administrator's Guide for information on when to use `FORCE LOGGING` mode and "Changing Tablespace Logging Attributes: Example"

tablespace_group_clause

This clause is valid only for locally managed temporary tablespaces. Use this clause to add *`tablespace`* to or remove it from the *`tablespace_group_name`* tablespace group.

- Specify a group name to indicate that `tablespace` is a member of this tablespace group. If `tablespace_group_name` does not already exist, then Oracle Database implicitly creates it when you alter tablespace to be a member of it.
- Specify an empty string (' ') to remove `tablespace` from the `tablespace_group_name` tablespace group.

Restriction on Tablespace Groups

You cannot specify a tablespace group for a permanent tablespace or for a dictionary-managed temporary tablespace.

See Also:

Oracle Database Administrator's Guide for more information on tablespace groups and "Assigning a Tablespace Group: Example"

tablespace_state_clauses

Use these clauses to set or change the state of the tablespace.

ONLINE | OFFLINE

Specify `ONLINE` to bring the tablespace online. Specify `OFFLINE` to take the tablespace offline and prevent further access to its segments. When you take a tablespace offline, all of its data files are also offline.

Note:

Before taking a tablespace offline for a long time, consider changing the tablespace allocation of any users who have been assigned the tablespace as either a default or temporary tablespace. While the tablespace is offline, such users cannot allocate space for objects or sort areas in the tablespace. See ALTER USER for more information on allocating tablespace quota to users.

Restriction on Taking Tablespaces Offline

You cannot take a temporary tablespace offline.

OFFLINE NORMAL

Specify `NORMAL` to flush all blocks in all data files in the tablespace out of the system global area (SGA). You need not perform media recovery on this tablespace before bringing it back online. This is the default.

OFFLINE TEMPORARY

If you specify `TEMPORARY`, then Oracle Database performs a checkpoint for all online data files in the tablespace but does not ensure that all files can be written. Files that are offline when you issue this statement may require media recovery before you bring the tablespace back online.

OFFLINE IMMEDIATE

If you specify `IMMEDIATE`, then Oracle Database does not ensure that tablespace files are available and does not perform a checkpoint. You must perform media recovery on the tablespace before bringing it back online.

Note:

The `FOR RECOVER` setting for `ALTER TABLESPACE` ... `OFFLINE` has been deprecated. The syntax is supported for backward compatibility. However, Oracle recommends that you use the transportable tablespaces feature for tablespace recovery.

See Also:

Oracle Database Backup and Recovery User's Guide for information on using transportable tablespaces to perform media recovery

READ ONLY | READ WRITE

Specify `READ ONLY` to place the tablespace in **transition read-only mode**. In this state, existing transactions can complete (commit or roll back), but no further DML operations are allowed to the tablespace except for rollback of existing transactions that previously modified blocks in the tablespace. You cannot make the `SYSAUX`, `SYSTEM`, or temporary tablespaces `READ ONLY`.

When a tablespace is read only, you can copy its files to read-only media. You must then rename the data files in the control file to point to the new location by using the SQL statement `ALTER DATABASE` ... `RENAME`.

See Also:

- *Oracle Database Concepts* for more information on read-only tablespaces
- ALTER DATABASE

Specify `READ WRITE` to indicate that write operations are allowed on a previously read-only tablespace.

PERMANENT | TEMPORARY

Specify `PERMANENT` to indicate that the tablespace is to be converted from a temporary to a permanent tablespace. A permanent tablespace is one in which permanent database objects can be stored. This is the default when a tablespace is created.

Specify `TEMPORARY` to indicate that the tablespace is to be converted from a permanent to a temporary tablespace. A temporary tablespace is one in which no permanent database objects can be stored. Objects in a temporary tablespace persist only for the duration of the session.

Restrictions on Temporary Tablespaces

Temporary tablespaces are subject to the following restrictions:

- You cannot specify `TEMPORARY` for the `SYSAUX` tablespace.

- If `tablespace` was not created with a standard block size, then you cannot change it from permanent to temporary.
- You cannot specify `TEMPORARY` for a tablespace in `FORCE LOGGING` mode.

autoextend_clause

This clause is valid only for bigfile (single-file) tablespaces. Use this clause to enable or disable autoextension of the single data file in the tablespace. To enable or disable autoextension of a newly added data file or temp file in smallfile tablespaces, use the `autoextend_clause` of the *database_file_clauses* in the `ALTER DATABASE` statement.

See Also:

- *Oracle Database Administrator's Guide* for information about bigfile (single-file) tablespaces
- *file_specification* for more information about the `autoextend_clause`

flashback_mode_clause

Use this clause to specify whether this tablespace should participate in any subsequent `FLASHBACK DATABASE` operation.

- For you to turn `FLASHBACK` mode on, the database must be mounted and closed.
- For you to turn `FLASHBACK` mode off, the database must be mounted, either open `READ WRITE` or closed.

This clause is not valid for temporary tablespaces.

Refer to CREATE TABLESPACE for more complete information on this clause.

See Also:

Oracle Database Backup and Recovery User's Guide for more information about Flashback Database

tablespace_retention_clause

This clause has the same semantics in `CREATE TABLESPACE` and `ALTER TABLESPACE` statements. Refer to tablespace_retention_clause in the documentation on `CREATE TABLESPACE`.

alter_tablespace_encryption

These clauses let you encrypt, decrypt, or rekey the tablespace.

`ONLINE` is the default for `ALTER TABLESPACE ENCRYPTION`.

ONLINE

- Specify `ENCRYPT` to encrypt the tablespace. The tablespace must be unencrpyted.

- Specify `REKEY` to encrypt an encrypted the tablespace using a different encryption algorithm. The tablespace must have been encrypted when it was created or encrypted with online conversion (`ONLINE ENCRYPT`).
- Specify `DECRYPT` to decrypt the tablespace. The tablespace must have been encrypted when it was created or encrypted with online conversion (`ONLINE ENCRYPT`).

FINISH

If an online conversion operation is interrupted, use the `FINISH` clause to finish the operation. The `ENCRYPT`, `DECRYPT`, `REKEY`, and *`ts_file_name_convert`* clauses have the same semantics here as they have for the `ONLINE` clause.

You can use `FINISH` to encrypt, decrypt, or rekey the tablespace with online conversion. The tablespace must be online. The online conversion method creates a new datafile for each datafile in the tablespace. Therefore, before using this clause, ensure that the amount of free disk space is greater than or equal to the amount of disk space currently used by the tablespace.

OFFLINE

This clause lets you encrypt or decrypt the tablespace with offline conversion. The tablespace must be offline or the database must be mounted, but not open. The offline conversion method does not use auxiliary disk space or files; it operates directly on the existing datafiles. Therefore, you should perform a full backup of the tablespace before converting it offline.

- Specify `ENCRYPT` to encrypt the tablespace. You can encrypt the tablespace using `AES128`, `AES192`, or `AES256` algorithms. The tablespace must be unencrpyted.
- Specify `DECRYPT` to decrypt the tablespace. The tablespace must have been previously encrypted with offline conversion (`OFFLINE ENCRYPT`).

If an offline conversion operation is interrupted, then you can reissue the offline conversion command to finish the operation.

tablespace_encryption_spec

Use this clause to specify the encryption algorithm to use when encrypting or rekeying the tablespace. If you omit this clause, then the datafiles will be encrypted using the `AES128` algorithm. Refer to *tablespace_encryption_spec* in the documentation on `CREATE TABLESPACE` for the full semantics of this clause.

ts_file_name_convert

Use this clause to determine how the database generates the names of the new datafiles that are created during online conversion.

If `FILE_NAME_CONVERT` is omitted, Oracle will internally select a name for the auxiliary file, and later rename it back to the original name.

- For *`filename_pattern`*, specify a string found in an existing datafile name.
- For *`replacement_filename_pattern`*, specify a replacement string. Oracle Database will replace *`filename_pattern`* with *`replacement_filename_pattern`* when naming the new datafile.
- Specify `KEEP` to retain the original files after the tablespace conversion is finished. If you omit this clause, then the original files are deleted when the conversion is finished.

Restriction on the *alter_tablespace_encryption* Clause

You cannot perform offline or online conversions on temporary tablespaces.

lost_write_protection

Before you can enable lost write protection on individual tablespaces, you must first enable the database for shadow lost write protection with `ALTER DATABASE`. Then you must create at least one shadow tablespace in that database using the `CREATE TABLESPACE` command.

After these steps you can use `ALTER TABLESPACE` to enable, remove, and suspend lost write protection on the shadow tablespace.

Example: Enable Lost Write Protection for a Tablespace

The following command enables lost write protection for the `tbsu1` tablespace.

```
 ALTER TABLESPACE tbsu1 ENABLE LOST WRITE PROTECTION
```

Example: Remove Lost Write Protection for a Shadow Tablespace

The following command removes lost write protection for the `tbsu1` tablespace.

```
 ALTER TABLESPACE tbsu1 REMOVE LOST WRITE PROTECTION
```

Example: Suspend Lost Write Protection for a Shadow Tablespace

The following command suspends lost write protection for the `tbsu1` tablespace.

```
 ALTER TABLESPACE tbsu1 SUSPEND LOST WRITE PROTECTION
```

See Also:

Managing Lost Write Protection with Shadow Tablespaces

Examples

Backing Up Tablespaces: Examples

The following statement signals to the database that a backup is about to begin:

```
ALTER TABLESPACE tbs_01
    BEGIN BACKUP;
```

The following statement signals to the database that the backup is finished:

```
ALTER TABLESPACE tbs_01
   END BACKUP;
```

Moving and Renaming Tablespaces: Example

This example moves and renames a data file associated with the `tbs_02` tablespace, created in "Enabling Autoextend for a Tablespace: Example", from `diskb:tbs_f5.dbf` to `diska:tbs_f5.dbf`:

1. Take the tablespace offline using an `ALTER TABLESPACE` statement with the `OFFLINE` clause:

   ```
   ALTER TABLESPACE tbs_02 OFFLINE NORMAL;
   ```

2. Copy the file from `diskb:tbs_f5.dbf` to `diska:tbs_f5.dbf` using your operating system commands.

3. Rename the data file using an `ALTER TABLESPACE` statement with the `RENAME DATAFILE` clause:

```
ALTER TABLESPACE tbs_02
  RENAME DATAFILE 'diskb:tbs_f5.dbf'
  TO              'diska:tbs_f5.dbf';
```

4. Bring the tablespace back online using an `ALTER TABLESPACE` statement with the `ONLINE` clause:

```
ALTER TABLESPACE tbs_02 ONLINE;
```

Adding and Dropping Data Files and Temp Files: Examples

The following statement adds a data file to the tablespace. When more space is needed, new 10-kilobytes extents will be added up to a maximum of 100 kilobytes:

```
ALTER TABLESPACE tbs_03
    ADD DATAFILE 'tbs_f04.dbf'
    SIZE 100K
    AUTOEXTEND ON
    NEXT 10K
    MAXSIZE 100K;
```

The following statement drops the empty data file:

```
ALTER TABLESPACE tbs_03
    DROP DATAFILE 'tbs_f04.dbf';
```

The following statements add a temp file to the temporary tablespace created in "Creating a Temporary Tablespace: Example" and then drops the temp file:

```
ALTER TABLESPACE temp_demo ADD TEMPFILE 'temp05.dbf' SIZE 5 AUTOEXTEND ON;

ALTER TABLESPACE temp_demo DROP TEMPFILE 'temp05.dbf';
```

Managing Space in a Temporary Tablespace: Example

The following statement manages the space in the temporary tablespace created in "Creating a Temporary Tablespace: Example" using the `SHRINK SPACE` clause. The `KEEP` clause is omitted, so the database will attempt to shrink the tablespace as much as possible as long as other tablespace storage attributes are satisfied.

```
ALTER TABLESPACE temp_demo SHRINK SPACE;
```

Adding an Oracle-managed Data File: Example

The following example adds an Oracle-managed data file to the `omf_ts1` tablespace (see "Creating Oracle Managed Files: Examples" for the creation of this tablespace). The new data file is 100M and is autoextensible with unlimited maximum size:

```
ALTER TABLESPACE omf_ts1 ADD DATAFILE;
```

Changing Tablespace Logging Attributes: Example

The following example changes the default logging attribute of a tablespace to `NOLOGGING`:

```
ALTER TABLESPACE tbs_03 NOLOGGING;
```

Altering a tablespace logging attribute has no affect on the logging attributes of the existing schema objects within the tablespace. The tablespace-level logging attribute can be overridden by logging specifications at the table, index, and partition levels.

Changing Undo Data Retention: Examples

The following statement changes the undo data retention for tablespace `undots1` to normal undo data behavior:

```
ALTER TABLESPACE undots1
  RETENTION NOGUARANTEE;
```

The following statement changes the undo data retention for tablespace `undots1` to behavior that preserves unexpired undo data:

```
ALTER TABLESPACE undots1
  RETENTION GUARANTEE;
```

ALTER 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 `ALTER TABLESPACE SET` statement to change an attribute of an existing tablespace set. The attribute change is applied to all tablespaces in the tablespace set.

See Also:

CREATE TABLESPACE SET and DROP TABLESPACE SET

Prerequisites

You must be connected to a shard catalog database as an SDB user.

If you have the `ALTER TABLESPACE` system privilege, then you can perform any `ALTER TABLESPACE SET` operation. If you have the `MANAGE TABLESPACE` system privilege, then you can only perform the following operations:

- Take all tablespaces in a tablespace set online or offline
- Begin or end a backup
- Make all tablespaces in a tablespace set read only or read write
- Set the default logging mode of all tablespaces in a tablespace set to `LOGGING` or `NOLOGGING`
- Put all tablespaces in a tablespace set in force logging mode or take them out of force logging mode
- Resize all data files for a tablespace set
- Enable or disable autoextension of all data files for a tablespace set

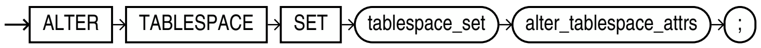
Before you can make a tablespace set read only, the following conditions must be met:

- The tablespaces in the tablespace set must be online.

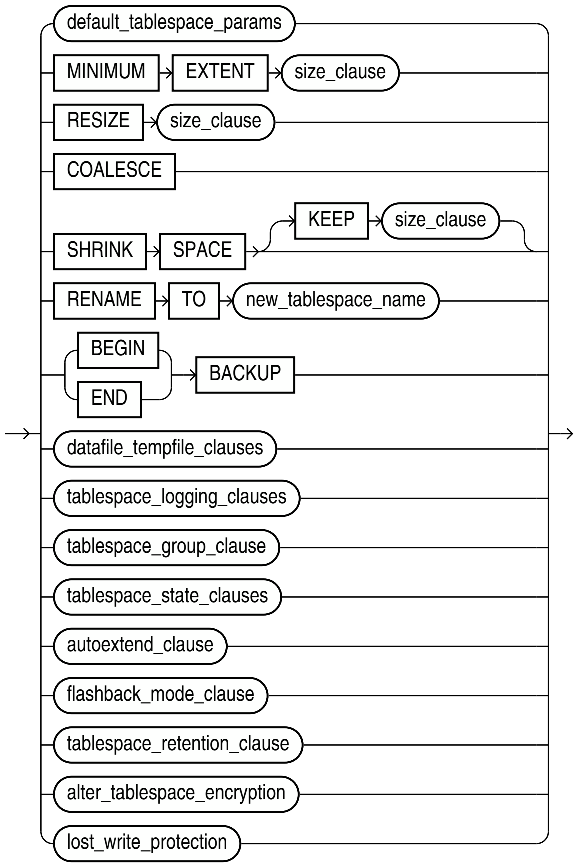
- The tablespace set must not contain any active rollback segments. Additionally, because the rollback segments of a read-only tablespace set are not accessible, Oracle recommends that you drop the rollback segments before you make a tablespace set read only.
- The tablespace set must not be involved in an open backup, because the end of a backup updates the header file of all data files in the tablespace set.

Syntax

alter_tablespace_set::=



alter_tablespace_attrs::=



(See the following clauses of `ALTER TABLESPACE`: *default_tablespace_params*::=, *size_clause*::=, *datafile_tempfile_clauses*::=, *tablespace_logging_clauses*::=, *tablespace_state_clauses*::=, *autoextend_clause*::=, *alter_tablespace_encryption*::=)

Semantics

tablespace_set

Specify the name of the tablespace set to be altered.

alter_tablespace_attrs

Use this clause to change an attribute for all tablespaces in the tablespace set.

The subclauses of `alter_tablespace_attrs` have the same semantics here as for the `ALTER TABLESPACE` statement, with the following exceptions:

- You cannot specify the following subclauses for tablespace sets:
  - `MINIMUM EXTENT size_clause`
  - `SHRINK SPACE [ KEEP size_clause ]`
  - `tablespace_group_clause`
  - `flashback_mode_clause`
  - `tablespace_retention_clause`
- For the `datafile_tempfile_clauses`, only the following subclauses are supported for tablespace sets:
  - `RENAME DATAFILE`
  - `DATAFILE { ONLINE | OFFLINE }`
- For the `tablespace_state_clauses`, the `PERMANENT` and `TEMPORARY` subclauses are not supported for tablespace sets.

See Also:

alter_tablespace_attrs in the documentation on `ALTER TABLESPACE` for the full semantics of this clause

Examples

Altering a Tablespace Set: Example

The following statement puts all tablespaces in tablespace set `ts1` in force logging mode:

```
ALTER TABLESPACE SET ts1
  FORCE LOGGING;
```

ALTER 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 `ALTER TRIGGER` statement to enable, disable, or compile a database trigger.

Note:

This statement does not change the declaration or definition of an existing trigger. To redeclare or redefine a trigger, use the `CREATE TRIGGER` statement with the `OR REPLACE` keywords.

See Also:

- CREATE TRIGGER for information on creating a trigger
- DROP TRIGGER for information on dropping a trigger
- *Oracle Database Concepts* for general information on triggers

Prerequisites

The trigger must be in your own schema or you must have `ALTER ANY TRIGGER` system privilege.

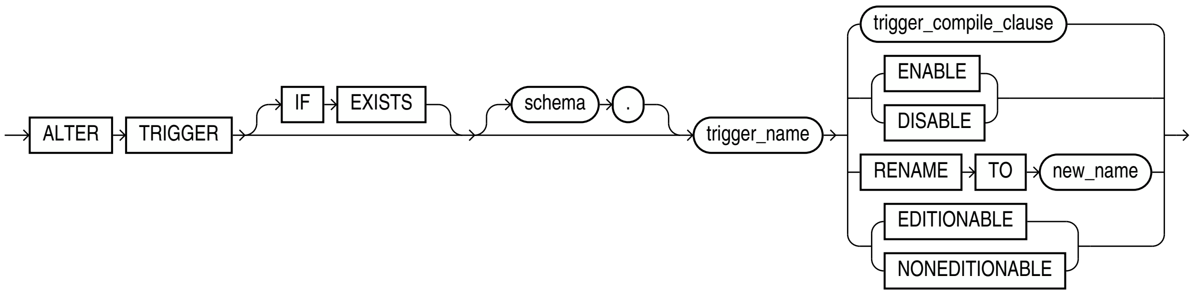
In addition, to alter a trigger on `DATABASE`, you must have the `ADMINISTER DATABASE TRIGGER` privilege.

See Also:

CREATE TRIGGER for more information on triggers based on `DATABASE` triggers

Syntax

***alter_trigger*::=**



(*`trigger_compile_clause`*: See *Oracle Database PL/SQL Language Reference* for the syntax of this clause.)

Semantics

IF EXISTS

Specify `IF EXISTS` to alter an existing table.

Specifying `IF NOT EXISTS` with `ALTER VIEW` results in `ORA-11544: Incorrect IF EXISTS clause for ALTER/DROP statement`.

schema

Specify the schema containing the trigger. If you omit *`schema`*, then Oracle Database assumes the trigger is in your own schema.

trigger_name

Specify the name of the trigger to be altered.

trigger_compile_clause

See *Oracle Database PL/SQL Language Reference* for the syntax and semantics of this clause and for complete information on creating and compiling triggers.

ENABLE | DISABLE

Specify `ENABLE` to enable the trigger. You can also use the `ENABLE ALL TRIGGERS` clause of `ALTER TABLE` to enable all triggers associated with a table. See ALTER TABLE.

Specify `DISABLE` to disable the trigger. You can also use the `DISABLE ALL TRIGGERS` clause of `ALTER TABLE` to disable all triggers associated with a table.

RENAME Clause

Specify `RENAME TO` *`new_name`* to rename the trigger. Oracle Database renames the trigger and leaves it in the same state it was in before being renamed.

When you rename a trigger, the database rebuilds the remembered source of the trigger in the `USER_SOURCE`, `ALL_SOURCE`, and `DBA_SOURCE` data dictionary views. As a result, comments and formatting may change in the `TEXT` column of those views even though the trigger source did not change.

EDITIONABLE | NONEDITIONABLE

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

Restriction on NONEDITIONABLE

You cannot specify `NONEDITIONABLE` for a crossedition trigger.

ALTER 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.

Use the `ALTER TYPE` statement to add or drop member attributes or methods. You can change the existing properties (`FINAL` or `INSTANTIABLE`) of an object type, and you can modify the scalar attributes of the type.

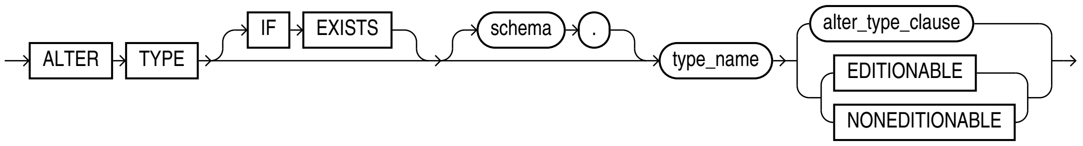
You can also use this statement to recompile the specification or body of the type or to change the specification of an object type by adding new object member subprogram specifications.

Prerequisites

The object type must be in your own schema and you must have `CREATE TYPE` or `CREATE ANY TYPE` system privilege, or you must have `ALTER ANY TYPE` system privileges.

Syntax

alter_type::=



(*`alter_type_clause`*: See *Oracle Database PL/SQL Language Reference* for the syntax of this clause.)

Semantics

IF EXISTS

Specify `IF EXISTS` to alter an existing table.

Specifying `IF NOT EXISTS` with `ALTER VIEW` results in `ORA-11544: Incorrect IF EXISTS clause for ALTER/DROP statement`.

schema

Specify the schema that contains the type. If you omit *`schema`*, then Oracle Database assumes the type is in your current schema.

type_name

Specify the name of an object type, a nested table type, or a varray type.

Restriction on *type_name*

You cannot evolve an editioned object type. The `ALTER TYPE` statement fails with ORA-22348 if either of the following is true:

- The type is an editioned object type and the `ALTER TYPE` statement has no *`type_compile_clause`*. You can use the `ALTER TYPE` statement to recompile an editioned object type, but not for any other purpose.
- The type has a dependent that is an editioned object type and the `ALTER TYPE` statement has a `CASCADE` clause.

Refer to *Oracle Database PL/SQL Language Reference* for more information on the *`type_compile_clause`* and the `CASCADE` clause.

alter_type_clause

See *Oracle Database PL/SQL Language Reference* for the syntax and semantics of this clause and for complete information on creating and compiling object types.

EDITIONABLE | NONEDITIONABLE

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

ALTER USER

Purpose

Use the `ALTER USER` statement:

- To change the authentication or database resource characteristics of a database user
- To permit a proxy server to connect as a client without authentication
- In an Oracle Automatic Storage Management (Oracle ASM) cluster, to change the password of a user in the password file that is local to the Oracle ASM instance of the current node

See Also:

Oracle Database Security Guide for detailed information about user authentication methods

Prerequisites

In general, you must have the `ALTER USER` system privilege. However, the current user can change his or her own password without this privilege.

To change the `SYS` password, password file must exist, and an account granted alter user privilege must have the `SYSDBA` administrative role in order to have the ability to change `SYS` password.

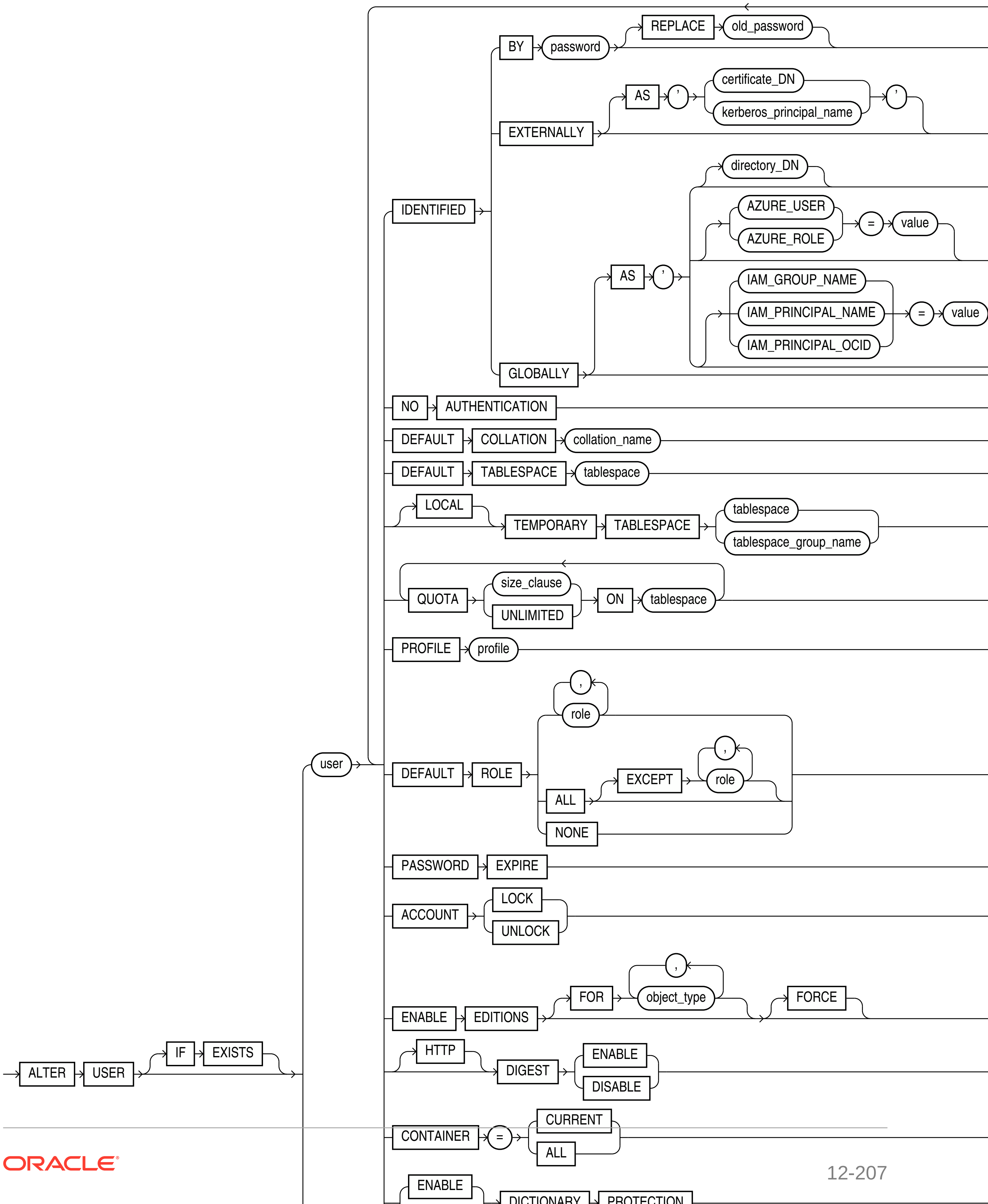
You must be authenticated `AS SYSASM` to change the password of a user other than yourself in an Oracle ASM instance password file.

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

To set and modify `CONTAINER_DATA` attributes using the *`container_data_clause`*, you must be connected to a CDB and the current container must be the root.

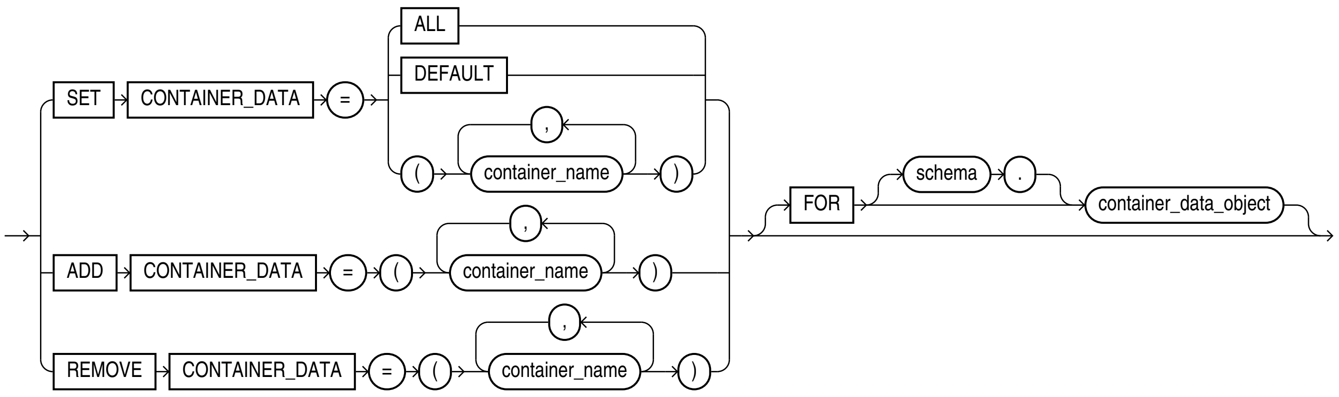
Syntax

***alter_user*::=**

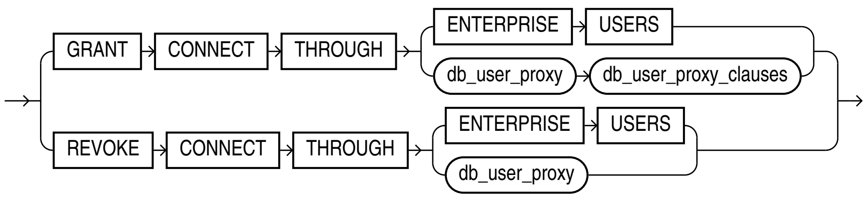


(*size_clause::=*)

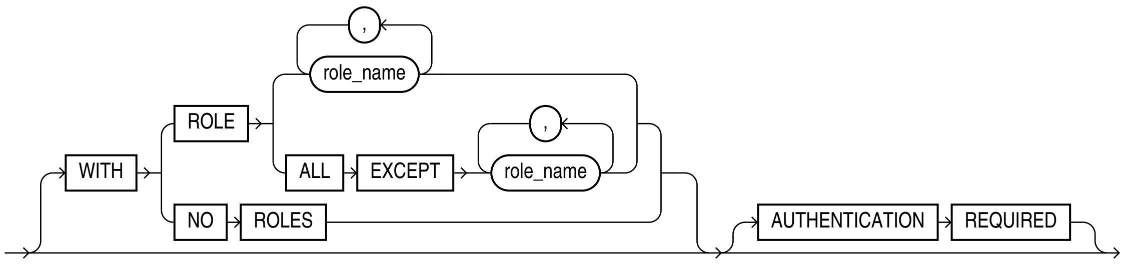
***container_data_clause*::=**



***proxy_clause*::=**



***db_user_proxy_clauses*::=**



Semantics

The keywords, parameters, and clauses described in this section are unique to `ALTER USER` or have different semantics than they have in `CREATE USER`. Keywords, parameters, and clauses that do not appear here have the same meaning as in the `CREATE USER` statement.

Note:

Oracle recommends that user names and passwords be encoded in ASCII or EBCDIC characters only, depending on your platform.

See Also:

CREATE USER for information on the keywords and parameters and CREATE PROFILE for information on assigning limits on database resources to a user

IF EXISTS

Specify `IF EXISTS` to alter an existing table.

Specifying `IF NOT EXISTS` with `ALTER VIEW` results in `ORA-11544: Incorrect IF EXISTS clause for ALTER/DROP statement`.

IDENTIFIED Clause

BY *password*

Specify `BY` *`password`* to specify a new password for the user. Passwords are case sensitive. 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 `ALTER USER` statement. Passwords can contain single-byte, or multibyte characters, or both from your database character set.

Note:

Oracle Database expects a different timestamp for each resetting of a particular password. If you reset one password multiple times within one second (for example, by cycling through a set of passwords using a script), then the database may return an error message that the password cannot be reused. For this reason, Oracle recommends that you avoid using scripts to reset passwords.

You can omit the `REPLACE` clause if you are setting your own password or you have the `ALTER USER` system privilege and you are changing another user's password. However, unless you have the `ALTER USER` system privilege, you must always specify the `REPLACE` clause if a password complexity verification function has been enabled, either by running the `UTLPWDMG.SQL` script or by specifying such a function in the `PASSWORD_VERIFY_FUNCTION` parameter of a profile that has been assigned to the user.

In an Oracle ASM cluster, you can use this clause to change the password of a user in the password file that is local to an Oracle ASM instance of the current node. You must be authenticated `AS SYSASM` to specify `IDENTIFIED BY` *`password`* without the `REPLACE` *`old_password`* clause. If you are not authenticated `AS SYSASM`, then you can only change your own password by specifying `REPLACE` *`old_password`*.

Oracle Database does not check the old password, even if you provide it in the `REPLACE` clause, unless you are changing your own existing password.

Changing a Password to Begin the Gradual Database Password Rollover Period

Prerequisite

Enable gradual database password rollover period by setting a non-zero value to the `PASSWORD_ROLLOVER_TIME` user profile parameter using `CREATE PROFILE` or `ALTER PROFILE` .

After you set the `PASSWORD_ROLLOVER_TIME` to specify the duration of the gradual password rollover period in the profile of the user, you can use the `ALTER USER` statement to change the user's password, which will allow clients to login using both the old password and the new password until the password rollover period expires.

During the password rollover period, you must propagate the new password to all clients (before the `PASSWORD_ROLLOVER_TIME` ends). If you successfully propagated the new password to all clients early (before the end of the password rollover period), then you can use the `EXPIRE PASSWORD ROLLOVER PERIOD` clause to end the password rollover (finalizing the password change, so that only the new password can be used).

Changing a Password During the Gradual Database Password Rollover Period

You can change the password during the password rollover period (before the rollover period expires) using `ALTER USER` with or without the `REPLACE` clause.

For example, say user `u1` has an original password `p1`, and `p2` is the new password that started the rollover process. Now you want to switch to `p3` instead of `p2`. You can use any one of the statements to change the password to `p3`:

```
ALTER USER u1 IDENTIFIED BY p3;

ALTER USER u1 IDENTIFIED BY p3 REPLACE p1;

ALTER USER u1 IDENTIFIED BY p3 REPLACE p2;
```

After you change the password to `p3`, the user can log in using either `p1` or `p3`. Logging in with `p2` returns error `Invalid credential or not authorized; logon denied` and is recorded as a failed login attempt.

The rollover start time remains set to the password change timestamp, this is the time the password of the user was changed. The rollover start time and password change time are not affected by any further password change made during the password rollover period. The old password can be used for at most `PASSWORD_ROLLOVER_TIME` days.

See Also:

- *Oracle Database Security Guide* for guidelines on creating passwords
- Configuring Authentication

GLOBALLY

Refer to CREATE USER for more information on this clause.

You can change a user's access verification method *from* `IDENTIFIED GLOBALLY` to either `IDENTIFIED BY` *`password`* or `IDENTIFIED EXTERNALLY`. You can change a user's access verification method *to* `IDENTIFIED GLOBALLY` from one of the other methods only if all external roles granted explicitly to the user are revoked.

EXTERNALLY

Refer to CREATE USER for more information on this clause.

See Also:

Oracle Database Enterprise User Security Administrator's Guide for more information on globally and externally identified users, "Changing User Identification: Example", and "Changing User Authentication: Examples"

NO AUTHENTICATION Clause

Use this clause to change an existing user account with authentication to a schema account without authentication to prevent logins to the account.

DEFAULT COLLATION Clause

Use this clause to change the default collation for the schema owned by the user. The new default collation is assigned to tables, views, and materialized views that are subsequently created in the schema. It does not influence default collations for existing tables views, and materialized views. Refer to the DEFAULT COLLATION Clause clause of `CREATE USER` for the full semantics of this clause.

DEFAULT TABLESPACE Clause

Use this clause to assign or reassign a tablespace for the user's permanent segments. This clause overrides any default tablespace that has been specified for the database.

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.

[LOCAL] TEMPORARY TABLESPACE Clause

Use this clause to assign or reassign a temporary tablespace or tablespace group for the user's temporary segments.

- 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.

Restriction on User Temporary Tablespace

Any individual tablespace you assign or reassign as the user's temporary tablespace must be a temporary tablespace and must have a standard block size.

See Also:

"Assigning a Tablespace Group: Example"

DEFAULT ROLE Clause

Specify the roles enabled by default for the user at logon.This clause can contain only roles that have been granted directly to the user with a `GRANT` statement, or roles created by the user with the `CREATE ROLE` privilege. You cannot use the `DEFAULT ROLE` clause to specify:

- Roles not granted to the user
- Roles granted through other roles
- Roles managed by an external service (such as the operating system), or by the Oracle Internet Directory
- Roles that are enabled by the `SET ROLE` statement, such as password-authenticated roles and secure application roles

See Also:

CREATE ROLE

Assigning Default Roles to Common Users in a CDB

You can modify the default role assigned to a common user both in the current container and across all containers in a CDB.

While assigning a default role to a common user across all containers, `role` must be a common role that was commonly granted to the common user.

While assigning a default role to a common user in the current container, `role` must be one of the following:

- A local role that was granted to the common user in the current container
- A common role that was granted to the common user, either commonly or locally in the current container

EXPIRE PASSWORD ROLLOVER PERIOD Clause

You can manually expire the password rollover period with `EXPIRE PASSWORD ROLLOVER PERIOD`.

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 non-editions-enabled schemas cannot be editioned.

Use the `FOR` clause to specify one or more object types for which the user can create editionable objects. For a list of valid values for *`object_type`*, query the `V$EDITIONABLE_TYPES` dynamic performance view.

If you omit the `FOR` clause, then the types that become editionable in the schema are `VIEW`, `SYNONYM`, `PROCEDURE`, `FUNCTION`, `PACKAGE`, `PACKAGE BODY`, `TRIGGER`, `TYPE`, `TYPE BODY`, and `LIBRARY`.

To enable edition for other object types that are not enabled by default, you must explicitly specify the object type in the `FOR` clause.

Example: Enable Edition for Object Type not Enabled by Default

```
ALTER USER username ENABLE EDITIONS FOR SQL TRANSLATION PROFILE;
```

See Also:

- For more on the semantics of the ENABLE EDITIONS clause see the corresponding section in *CREATE USER*
- *Enabling Editions for a User*
- *Oracle Database Reference* for more information about the `V$EDITIONABLE_TYPES` dynamic performance view

If the schema to be editions-enabled contains any objects that are not editionable and that depend on editionable type objects in the schema, then you must specify `FORCE` to enable editions for this schema. In this case, all the objects that are not editionable and that depend on the editionable type objects in the schema being editions-enabled become invalid.

[HTTP] DIGEST Clause

This clause lets you enable or disable HTTP Digest Access Authentication for the user.

- Specify `ENABLE` to enable HTTP Digest Access Authentication. After specifing this clause, you must change the user's password. This causes the database to generate an HTTP Digest verifier for the new password. Only then will HTTP Digest Access Authentication take effect. One way to ensure that the user's password is changed after you issue this clause is to specify the `PASSWORD EXPIRE` clause in the same statement with the `HTTP DIGEST ENABLE` clause, as follows:

  ```
  ALTER USER user PASSWORD EXPIRE HTTP DIGEST ENABLE;
  ```

  This causes the database to prompt the user for a new password on his or her next attempt to log in to the database. After that, HTTP Digest Access Authentication will take effect for the user.
- Specify `DISABLE` to disable HTTP Digest Access Authentication for the user. You do not need to change the user's password in order for this clause to take effect. Specifying the `DISABLE` clause removes the HTTP Digest from dictionary tables.

  ```
  ALTER USER user PASSWORD EXPIRE HTTP DIGEST DISABLE;
  ```

Refer to [HTTP] DIGEST Clause in the documentation on `CREATE USER` for more information on this clause.

CONTAINER Clause

If the current container is a PDB, then you can specify `CONTAINER = CURRENT` to change the attributes of a local user, or the container-specific attributes (such as the default tablespace) of a common user, in the current container. If the current container is the root, then you can specify `CONTAINER = ALL` to change the attributes of a common user across the entire CDB. If you omit this clause and the current container is a PDB, then `CONTAINER = CURRENT` is the default. If you omit this clause and the current container is the root, then `CONTAINER = ALL` is the default.

Restriction on Modifying Common Users in a CDB

Certain attributes of a common user must be modified for all the containers in a CDB and not for only some containers. Therefore, when you use any of the following clauses to modify a common user, ensure that you modify all of the containers by connecting to the root and specifying `CONTAINER=ALL`:

- `IDENTIFIED` clause
- `PASSWORD` clause
- `[HTTP] DIGEST` clause

ENABLE or DISABLE DICTIONARY PROTECTION

Use this clause to enable or disable dictionary protection on the created user. When a schema is dictionary protected, other users cannot use system privileges (including `ANY` privileges) on the schema, even if they have been granted the system privilege on the schema. Only the `SELECT ANY DICTIONARY` and `ANALYZE ANY DICTIONARY` system privileges can be used on a dictionary-protected schema. Users can still use object privileges on the schema, assuming that the user has been granted the object privilege on the schema. A user without the object privileges on the object but with corresponding system privileges will be denied access to the object with an insufficient privileges error.

By default, Oracle-maintained schemas have dictionary protection, but this protection can be temporarily removed if necessary. You cannot enable dictionary protection on a customer (Non-Oracle maintained) schema. You also cannot create a custom schema with dictionary protection enabled.

You must be logged in as user `SYS` with the `SYSDBA` administrative privilege in order to manage dictionary protection for Oracle-maintained schemas.

See Also:

- *Configuring Privilege and Role Authorization* of the *Oracle Database Security Guide*.

READ ONLY | READ WRITE

Specify `READ ONLY` 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 revoke `READ ONLY` access on a local user.

You must have the `ALTER USER` privilege to execute this statement.

You can view the state of a local user in the `*_USERS` view.

container_data_clause

The *`container_data_clause`* allows you the set and modify `CONTAINER_DATA` attributes for a common user. Use the `FOR` clause to indicate whether to set or modify the default

CONTAINER_DATA attribute or an object-specific CONTAINER_DATA attribute. These attributes determine the set of containers (which can never exclude the root) whose data will be visible via CONTAINER_DATA objects to the specified common user when the current session is the root.

To specify the *container_data_clause*, the current session must be the root and you must specify CONTAINER = CURRENT.

SET CONTAINER_DATA

Use this clause to set the default CONTAINER_DATA attribute or an object-specific CONTAINER_DATA attribute for a common user. When you specify this clause, you replace the existing value, if any, of the CONTAINER_DATA attribute.

Use *container_name* to specify one or more containers that will be accessible to the user.

Use ALL to specify that all current and future containers in the CDB will be accessible to the user.

Use DEFAULT to specify the default behavior, which is as follows:

- For a default CONTAINER_DATA attribute, the current container, that is, the root, and the CDB as a whole will be accessible to the user.
- For an object-specific CONTAINER_DATA attribute, the database will use the user's default CONTAINER_DATA attribute.

Note:

CONTAINER_DATA attributes that are set to DEFAULT are not visible in the DBA_CONTAINER_DATA view.

ADD CONTAINER_DATA

Use this clause to add containers to the default CONTAINER_DATA attribute or an object-specific CONTAINER_DATA attribute for a common user. Use *container_name* to specify one or more containers to add.

You cannot use this clause if the default CONTAINER_DATA attribute is set to ALL. If you use this clause when the default CONTAINER_DATA attribute is set to DEFAULT, then CDB$ROOT will automatically be added to the set of containers, unless the set already contains CDB$ROOT.

You cannot use this clause if the object-specific CONTAINER_DATA attribute is set to ALL or DEFAULT.

REMOVE CONTAINER_DATA

Use this clause to remove containers from the default CONTAINER_DATA attribute or an object-specific CONTAINER_DATA attribute for a common user. Use *container_name* to specify one or more containers to remove.

You cannot use this clause if the default CONTAINER_DATA attribute or object-specific CONTAINER_DATA attribute is set to ALL or DEFAULT.

FOR *container_data_object*

If you specify the FOR clause, then you can set and modify the object-specific CONTAINER_DATA attribute for *container_data_object* for a common user. *container_data_object* must be a

`CONTAINER_DATA` table or view. If you omit *`schema`*, then Oracle Database assumes that *`container_data_object`* is in your own schema.

If you omit the `FOR` clause, then you can set and modify the default `CONTAINER_DATA` attribute for a common user.

> **See Also:**
>
> *Oracle Database Security Guide* for more information about enabling common users to view information about PDB objects

proxy_clause

The *`proxy_clause`* lets you control the ability of an enterprise user (a user outside the database) or a database proxy (another database user) to connect as the database user being altered.

GRANT CONNECT THROUGH

Specify `GRANT CONNECT THROUGH` to allow the connection.

REVOKE CONNECT THROUGH

Specify `REVOKE CONNECT THROUGH` to prohibit the connection.

ENTERPRISE USER

This clause lets you expose *`user`* to proxy use by enterprise users. The administrator working in Oracle Internet Directory must then grant privileges for appropriate enterprise users to act on behalf of *`user`*.

db_user_proxy

This clause lets you expose *`user`* to proxy use by database user *`db_user_proxy`* (the proxy).

- The proxy will have all privileges that were directly granted to user.
- The proxy will have all roles associated with user, unless you specify the `WITH` clauses of *`db_user_proxy_clauses`* to limit the proxy to some or none of the roles of user. For each role associated with the proxy, if the role is enabled by default for *`user`* at login, then that role will also be enabled by default for the proxy at login.

db_user_proxy_clauses

You can enable password-protected roles in a proxy session. Both secure application role and password-protected roles provide a secure method for enabling a role in a session. Oracle recommends using secure password roles instead of password protected roles in instances where the password has to be maintained and transmitted over insecure channels, or if more than one person needs to know the password. Password-protected roles in a proxy session are suitable for situations where automation is used to set the role.

Proxy users can access password-protected roles. Specify the `WITH` clauses to limit the proxy to some or none of the roles associated with *`user`*, and the `AUTHENTICATION REQUIRED` clause to specify whether authentication is required.

WITH ROLE

WITH ROLE *role_name* permits the proxy to connect as the specified user and to activate only the roles that are specified by *role_name*. This clause can contain only roles that are associated with *user*. Password protected roles and secure application roles also need to be listed in the WITH ROLE clause if the Proxy user will need to use these secure roles. These secure roles will be included with the WITH ROLE ALL clause (the default if WITH ROLE is not specified). If WITH ROLE doesn't specify the secure roles, then those cannot be enabled even with right password.

WITH ROLE ALL EXCEPT

WITH ROLE ALL EXCEPT *role_name* permits the proxy to connect as the specified user and to activate all roles associated with that user except those specified for *role_name*. This clause can contain only roles that are associated with *user*.

WITH NO ROLES

WITH NO ROLES permits the proxy to connect as the specified user, but prohibits the proxy from activating any of that user's roles after connecting, even the secure roles like password protected roles and secure application roles.

AUTHENTICATION REQUIRED

Oracle Database does not expect the proxy to authenticate the user unless you specify the AUTHENTICATION REQUIRED clause. This clause ensures that authentication credentials for the user must be presented when the user is authenticated through the specified proxy. The credential is a password.

AUTHENTICATED USING

The AUTHENTICATED USING clauses, which appeared in the syntax of earlier releases, have been deprecated and are no longer needed. If you specify the AUTHENTICATED USING PASSWORD clause, then Oracle Database converts it to the AUTHENTICATION REQUIRED clause. Specifying the AUTHENTICATED USING CERTIFICATE clause or the AUTHENTICATED USING DISTINGUISHED NAME clause is equivalent to omitting the AUTHENTICATION REQUIRED clause.

See Also:

- *Oracle Security Overview* for an overview of database security and for information on middle-tier systems and proxy authentication
- *Oracle Database Security Guide* for more information on proxies and their use of the database and "Proxy Users: Examples"

Examples

Changing User Identification: Example

The following statement changes the password of the user sidney (created in "Creating a Database User: Example") second_2nd_pwd and default tablespace to the tablespace example:

```
ALTER USER sidney
    IDENTIFIED BY second_2nd_pwd
    DEFAULT TABLESPACE example;
```

The following statement assigns the new_profile profile (created in "Creating a Profile: Example") to the sample user sh:

```
ALTER USER sh
    PROFILE new_profile;
```

In subsequent sessions, `sh` is restricted by limits in the `new_profile` profile.

The following statement makes all roles granted directly to `sh` default roles, except the `dw_manager` role:

```
ALTER USER sh
    DEFAULT ROLE ALL EXCEPT dw_manager;
```

At the beginning of `sh`'s next session, Oracle Database enables all roles granted directly to `sh` except the `dw_manager` role.

Changing User Authentication: Examples

The following statement changes the authentication mechanism of user `app_user1` (created in "Creating a Database User: Example"):

```
ALTER USER app_user1 IDENTIFIED GLOBALLY AS 'CN=tom,O=oracle,C=US';
```

The following statement causes user `sidney`'s password to expire:

```
ALTER USER sidney PASSWORD EXPIRE;
```

If you cause a database user's password to expire with `PASSWORD EXPIRE`, then the user (or the DBA) must change the password before attempting to log in to the database following the expiration. However, tools such as SQL*Plus allow the user to change the password on the first attempted login following the expiration.

Assigning a Tablespace Group: Example

The following statement assigns `tbs_grp_01` (created in "Adding a Temporary Tablespace to a Tablespace Group: Example") as the tablespace group for user `sh`:

```
ALTER USER sh
  TEMPORARY TABLESPACE tbs_grp_01;
```

Proxy Users: Examples

The following statement alters the user `app_user1`. The example permits the `app_user1` to connect through the proxy user `sh`. The example also allows `app_user1` to enable its `warehouse_user` role (created in "Creating a Role: Example") when connected through the proxy `sh`:

```
ALTER USER app_user1
   GRANT CONNECT THROUGH sh
   WITH ROLE warehouse_user;
```

To show basic syntax, this example uses the sample database Sales History user (`sh`) as the proxy. Normally a proxy user would be an application server or middle-tier entity. For information on creating the interface between an application user and a database by way of an application server, refer to *Oracle Call Interface Programmer's Guide.*

See Also:

- "Creating External Database Users: Examples" to see how to create the app_user user
- "Creating a Role: Example" to see how to create the dw_user role

The following statement takes away the right of user app_user1 to connect through the proxy user sh:

```
ALTER USER app_user1 REVOKE CONNECT THROUGH sh;
```

The following hypothetical examples shows another method of proxy authentication:

```
ALTER USER sully GRANT CONNECT THROUGH OAS1
   AUTHENTICATED USING PASSWORD;
```

The following example exposes the user app_user1 to proxy use by enterprise users. The enterprise users cannot act on behalf of app_user1 until the Oracle Internet Directory administrator has granted them appropriate privileges:

```
ALTER USER app_user1
   GRANT CONNECT THROUGH ENTERPRISE USERS;
```

ALTER VIEW

Purpose

Use the ALTER VIEW statement to explicitly recompile a view that is invalid or to modify view constraints. Explicit recompilation lets you locate recompilation errors before run time. You may want to recompile a view explicitly after altering one of its base tables to ensure that the alteration does not affect the view or other objects that depend on it.

You can also use ALTER VIEW to define, modify, or drop view constraints.

You cannot use this statement to change the definition of an existing view. Further, if DDL changes to the view's base tables invalidate the view, then you cannot use this statement to compile the invalid view. In these cases, you must redefine the view using CREATE VIEW with the OR REPLACE keywords.

When you issue an ALTER VIEW statement, Oracle Database recompiles the view regardless of whether it is valid or invalid. The database also invalidates any local objects that depend on the view.

If you alter a view that is referenced by one or more materialized views, then those materialized views are invalidated. Invalid materialized views cannot be used by query rewrite and cannot be refreshed.

See Also:

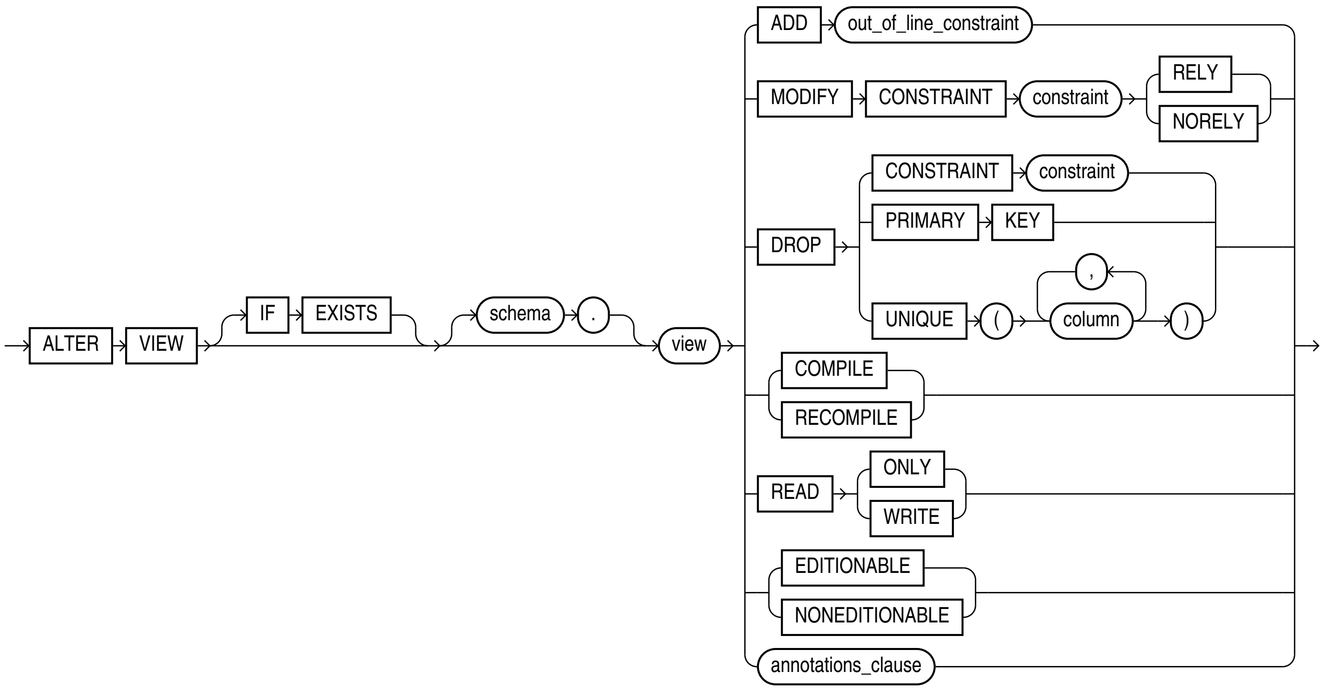
- CREATE VIEW for information on redefining a view and ALTER MATERIALIZED VIEW for information on revalidating an invalid materialized view
- *Oracle Database Data Warehousing Guide* for general information on data warehouses
- *Oracle Database Concepts* for more about dependencies among schema objects

Prerequisites

The view must be in your own schema or you must have `ALTER ANY TABLE` system privilege.

Syntax

alter_view::=



(*out_of_line_constraint*::=—part of *constraint*::= syntax), *annotations_clause*

Semantics

IF EXISTS

Specify `IF EXISTS` to alter an existing table.

Specifying `IF NOT EXISTS` with `ALTER VIEW` results in `ORA-11544: Incorrect IF EXISTS clause for ALTER/DROP statement`.

schema

Specify the schema containing the view. If you omit *`schema`*, then Oracle Database assumes the view is in your own schema.

view

Specify the name of the view to be recompiled.

MODIFY CONSTRAINT Clause

Use the `MODIFY CONSTRAINT` clause to change the `RELY` or `NORELY` setting of an existing view constraint. Refer to "Notes on View Constraints" for general information on view constraints.

Restriction on Modifying Constraints

You cannot change the setting of a unique or primary key constraint if it is part of a referential integrity constraint without dropping the foreign key or changing its setting to match that of *`view`*.

ADD Clause

Use the `ADD` clause to add a constraint to *`view`*. Refer to *constraint* for information on view constraints and their restrictions.

DROP Clause

Use the `DROP` clause to drop an existing view constraint.

Restriction on Dropping Constraints

You cannot drop a unique or primary key constraint if it is part of a referential integrity constraint on a view.

COMPILE | RECOMPILE

`RECOMPILE` and `COMPILE` keywords direct Oracle Database to recompile the view.

Use `RECOMPILE` to explicitly recompile a view that is invalid or to modify view constraints. Explicit recompilation allows users to locate recompilation errors, before run time.

{ READ ONLY | READ WRITE }

These clauses are valid only for editioning views.

- Specify `READ ONLY` to indicate that the editioning view cannot be updated.
- Specify `READ WRITE` to return a read-only editioning view to read/write status.

When you specify these clauses, the database does not invalidate dependent objects, but it may invalidate cursors.

EDITIONABLE | NONEDITIONABLE

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

See Also:

CREATE VIEW for information about editioning views

annotations_clause

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

You can only change annotations at the view level with the `ALTER` statement. To drop column-level annotations, you must drop and recreate the view.

Examples

Altering a View: Example

To recompile the view `customer_ro` (created in "Creating a Read-Only View: Example"), issue the following statement:

```
ALTER VIEW customer_ro
    COMPILE;
```

If Oracle Database encounters no compilation errors while recompiling `customer_ro`, then `customer_ro` becomes valid. If recompiling results in compilation errors, then the database returns an error and `customer_ro` remains invalid.

Oracle Database also invalidates all dependent objects. These objects include any procedures, functions, package bodies, and views that reference `customer_ro`. If you subsequently reference one of these objects without first explicitly recompiling it, then the database recompiles it implicitly at run time.

Add and Drop Annotations: Example

The following example drops annotation `Title` from view `MView1` and adds annotation `Identity`:

```
ALTER VIEW HighWageEmp ANNOTATIONS(DROP Title, ADD Identity);
```

ANALYZE

Purpose

Use the `ANALYZE` statement to collect statistics, for example, to:

- Collect or delete statistics about an index or index partition, table or table partition, index-organized table, cluster, or scalar object attribute.
- Validate the structure of an index or index partition, table or table partition, index-organized table, cluster, or object reference (`REF`).
- Identify migrated and chained rows of a table or cluster.

Note:

The use of `ANALYZE` for the collection of optimizer statistics is obsolete.

If you want to collect optimizer statistics, use the `DBMS_STATS` package, which lets you collect statistics in parallel, global statistics for partitioned objects, and helps you fine tune your statistics collection in other ways. See *Oracle Database PL/SQL Packages and Types Reference* for more information on the `DBMS_STATS` package.

Use the `ANALYZE` statement only for the following cases:

- To use the `VALIDATE` or `LIST CHAINED ROWS` clauses
- To collect information on freelist blocks

Prerequisites

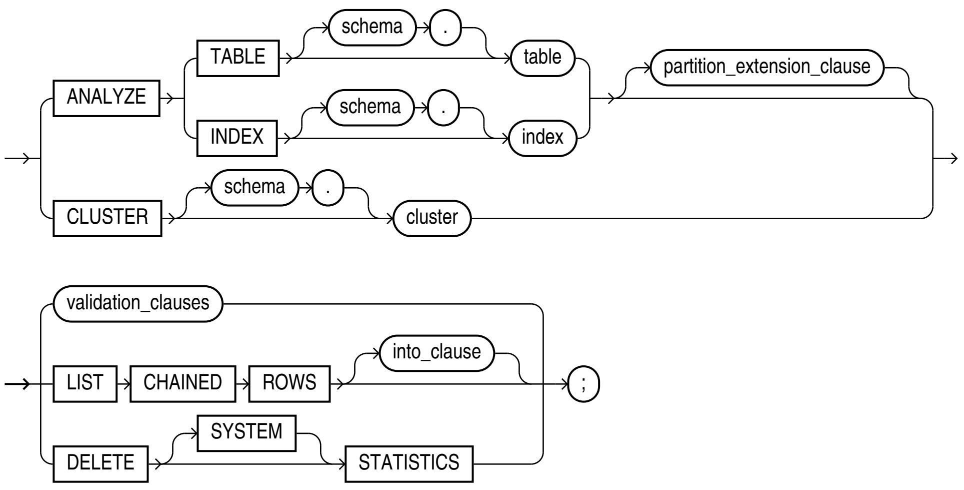
The schema object to be analyzed must be local, and it must be in your own schema or you must have the `ANALYZE ANY` system privilege.

If you want to list chained rows of a table or cluster into a list table, then the list table must be in your own schema, or you must have `INSERT` privilege on the list table, or you must have `INSERT ANY TABLE` system privilege.

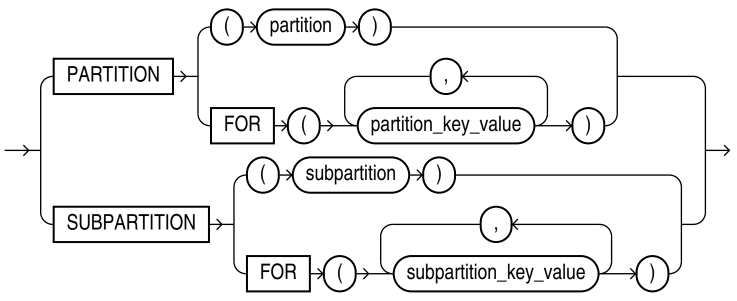
If you want to validate a partitioned table, then you must have the `INSERT` object privilege on the table into which you list analyzed rowids, or you must have the `INSERT ANY TABLE` system privilege.

Syntax

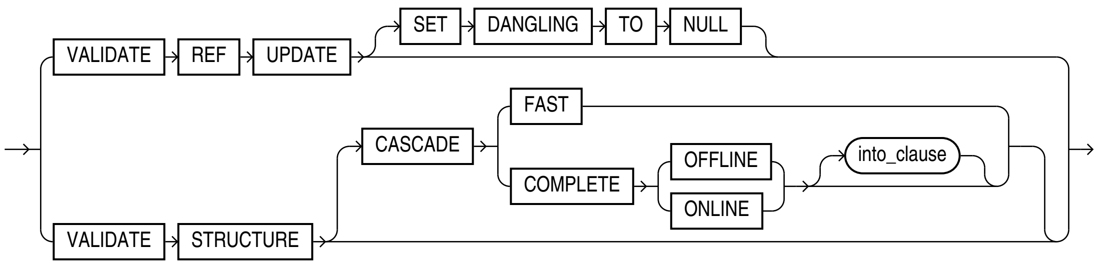
***analyze*::=**



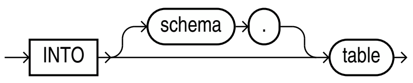
***partition_extension_clause*::=**



***validation_clauses*::=**



***into_clause*::=**



Semantics

schema

Specify the schema containing the table, index, or cluster. If you omit `schema`, then Oracle Database assumes the table, index, or cluster is in your own schema.

TABLE *table*

Specify a table to be analyzed. When you analyze a table, the database collects statistics about expressions occurring in any function-based indexes as well. Therefore, be sure to create function-based indexes on the table before analyzing the table. Refer to CREATE INDEX for more information about function-based indexes.

When analyzing a table, the database skips all domain indexes marked `LOADING` or `FAILED`.

For an index-organized table, the database also analyzes any mapping table and calculates its `PCT_ACCESSS_DIRECT` statistics. These statistics estimate the accuracy of guess data block addresses stored as part of the local rowids in the mapping table.

Oracle Database collects the following statistics for a table. Statistics marked with an asterisk are always computed exactly. Table statistics, including the status of domain indexes, appear in the data dictionary views `USER_TABLES`, `ALL_TABLES`, and `DBA_TABLES` in the columns shown in parentheses.

- Number of rows (`NUM_ROWS`)
- * Number of data blocks below the high water mark—the number of data blocks that have been formatted to receive data, regardless whether they currently contain data or are empty (`BLOCKS`)
- * Number of data blocks allocated to the table that have never been used (`EMPTY_BLOCKS`)
- Average available free space in each data block in bytes (`AVG_SPACE`)
- Number of chained rows (`CHAIN_COUNT`)
- Average row length, including the row overhead, in bytes (`AVG_ROW_LEN`)

Restrictions on Analyzing Tables

Analyzing tables is subject to the following restrictions:

- You cannot use `ANALYZE` to collect statistics on data dictionary tables.
- You cannot use `ANALYZE` to collect statistics on an external table. Instead, you must use the `DBMS_STATS` package.
- You cannot use `ANALYZE` to collect default statistics on a temporary table. However, if you have already created an association between one or more columns of a temporary table and a user-defined statistics type, then you can use `ANALYZE` to collect the user-defined statistics on the temporary table.
- You cannot compute or estimate statistics for the following column types: `REF` column types, varrays, nested tables, LOB column types (LOB column types are not analyzed, they are skipped), `LONG` column types, or object types. However, if a statistics type is associated with such a column, then Oracle Database collects user-defined statistics.

See Also:

- ASSOCIATE STATISTICS
- *Oracle Database Reference* for information on the data dictionary views

partition_extension_clause

partition_extension_clause

Specify the partition or subpartition, or the partition or subpartition value, on which you want statistics to be gathered. You cannot use this clause when analyzing clusters.

If you specify `PARTITION` and *`table`* is composite-partitioned, then Oracle Database analyzes all the subpartitions within the specified partition.

INDEX *index*

Specify an index to be analyzed.

Oracle Database collects the following statistics for an index. Statistics marked with an asterisk are always computed exactly. For conventional indexes, when you compute or estimate statistics, the statistics appear in the data dictionary views `USER_INDEXES`, `ALL_INDEXES`, and `DBA_INDEXES` in the columns shown in parentheses.

- * Depth of the index from its root block to its leaf blocks (`BLEVEL`)
- Number of leaf blocks (`LEAF_BLOCKS`)
- Number of distinct index values (`DISTINCT_KEYS`)
- Average number of leaf blocks for each index value (`AVG_LEAF_BLOCKS_PER_KEY`)
- Average number of data blocks for each index value (for an index on a table) (`AVG_DATA_BLOCKS_PER_KEY`)
- Clustering factor (how well ordered the rows are about the indexed values) (`CLUSTERING_FACTOR`)

For domain indexes, this statement invokes the user-defined statistics collection function specified in the statistics type associated with the index (see ASSOCIATE STATISTICS). If no statistics type is associated with the domain index, then the statistics type associated with its indextype is used. If no statistics type exists for either the index or its indextype, then no user-defined statistics are collected. User-defined index statistics appear in the `STATISTICS` column of the data dictionary views `USER_USTATS`, `ALL_USTATS`, and `DBA_USTATS`.

Note:

- When you analyze an index from which a substantial number of rows has been deleted, Oracle Database sometimes executes a `COMPUTE` statistics operation (which can entail a full table scan) even if you request an `ESTIMATE` statistics operation. Such an operation can be quite time consuming.
- In some cases, analyzing an index with the `ANALYZE` statement takes an inordinate amount of time to complete. In these cases, you can use a SQL query to validate the index. If the query determines that there is an inconsistency between a table and the index, then you can use the `ANALYZE` statement for a thorough analysis of the index. Refer to *Oracle Database Administrator's Guide* for more information.

Restriction on Analyzing Indexes

You cannot analyze a domain index that is marked `IN_PROGRESS` or `FAILED`.

See Also:

- CREATE INDEX for more information on domain indexes
- *Oracle Database Reference* for information on the data dictionary views
- "Analyzing an Index: Example"

CLUSTER *cluster*

Specify a cluster to be analyzed. When you collect statistics for a cluster, Oracle Database also automatically collects the statistics for all the tables in the cluster and all their indexes, including the cluster index.

For both indexed and hash clusters, the database collects the average number of data blocks taken up by a single cluster key (`AVG_BLOCKS_PER_KEY`). These statistics appear in the data dictionary views `ALL_CLUSTERS`, `USER_CLUSTERS`, and `DBA_CLUSTERS`.

See Also:

Oracle Database Reference for information on the data dictionary views and "Analyzing a Cluster: Example"

validation_clauses

The validation clauses let you validate `REF` values and the structure of the analyzed object.

See Also:

Oracle Database Administrator's Guide for more information about validating tables, indexes, clusters, and materialized views

VALIDATE REF UPDATE Clause

Specify `VALIDATE REF UPDATE` to validate the `REF` values in the specified table, check the rowid portion in each `REF`, compare it with the true rowid, and correct it, if necessary. You can use this clause only when analyzing a table.

If the owner of the table does not have the `READ` or `SELECT` object privilege on the referenced objects, then Oracle Database will consider them invalid and set them to null. Subsequently these `REF` values will not be available in a query, even if it is issued by a user with appropriate privileges on the objects.

SET DANGLING TO NULL

`SET DANGLING TO NULL` sets to null any `REF` values (whether or not scoped) in the specified table that are found to point to an invalid or nonexistent object.

VALIDATE STRUCTURE

Specify `VALIDATE STRUCTURE` to validate the structure of the analyzed object. The statistics collected by this clause are not used by the Oracle Database optimizer.

See Also:

"Validating a Table: Example"

- For a table, Oracle Database verifies the integrity of each of the data blocks and rows. For an index-organized table, the database also generates compression statistics (optimal prefix compression count) for the primary key index on the table.
- For a cluster, Oracle Database automatically validates the structure of the cluster tables.

- For a partitioned table, Oracle Database also verifies that each row belongs to the correct partition. If a row does not collate correctly, then its rowid is inserted into the `INVALID_ROWS` table.
- For a temporary table, Oracle Database validates the structure of the table and its indexes during the current session.
- For an index, Oracle Database verifies the integrity of each data block in the index and checks for block corruption. This clause does not confirm that each row in the table has an index entry or that each index entry points to a row in the table. You can perform these operations by validating the structure of the table with the CASCADE clause.

  Oracle Database also computes compression statistics (optimal prefix compression count) for all normal indexes.

  Oracle Database stores statistics about the index in the data dictionary views `INDEX_STATS` and `INDEX_HISTOGRAM`.

  See Also:

  Oracle Database Reference for information on these views

If Oracle Database encounters corruption in the structure of the object, then an error message is returned. In this case, drop and re-create the object.

CASCADE

Specify `CASCADE` if you want Oracle Database to validate the structure of the indexes associated with the table or cluster. If you use this clause when validating a table, then the database also validates the indexes defined on the table. If you use this clause when validating a cluster, then the database also validates all the cluster tables indexes, including the cluster index.

By default, `CASCADE` performs a `COMPLETE` validation, which can be resource intensive. Specify `FAST` if you want the database to check for the existence of corruptions without reporting details about the corruption. If the `FAST` check finds a corruption, you can then use the `CASCADE` option without the `FAST` clause to locate and learn details about it.

If you use this clause to validate an enabled (but previously disabled) function-based index, then validation errors may result. In this case, you must rebuild the index.

ONLINE | OFFLINE

Specify `ONLINE` to enable Oracle Database to run the validation while DML operations are ongoing within the object. The database reduces the amount of validation performed to allow for concurrency.

Note:

When you validate the structure of an object `ONLINE`, Oracle Database does not collect any statistics, as it does when you validate the structure of the object `OFFLINE`.

Specify `OFFLINE`, to maximize the amount of validation performed. This setting prevents `INSERT`, `UPDATE`, and `DELETE` statements from concurrently accessing the object during validation but allows queries. This is the default.

Restriction on ONLINE

You cannot specify ONLINE when analyzing a cluster.

INTO

The INTO clause of VALIDATE STRUCTURE is valid only for partitioned tables. Specify a table into which Oracle Database lists the rowids of the partitions whose rows do not collate correctly. If you omit *schema*, then the database assumes the list is in your own schema. If you omit this clause altogether, then the database assumes that the table is named INVALID_ROWS. The SQL script used to create this table is UTLVALID.SQL.

LIST CHAINED ROWS

LIST CHAINED ROWS lets you identify migrated and chained rows of the analyzed table or cluster. You cannot use this clause when analyzing an index.

In the INTO clause, specify a table into which Oracle Database lists the migrated and chained rows. If you omit *schema*, then the database assumes the chained-rows table is in your own schema. If you omit this clause altogether, then the database assumes that the table is named CHAINED_ROWS. The chained-rows table must be on your local database.

You can create the CHAINED_ROWS table using one of these scripts:

- UTLCHAIN.SQL uses physical rowids. Therefore it can accommodate rows from conventional tables but not from index-organized tables. (See the Note that follows.)
- UTLCHN1.SQL uses universal rowids, so it can accommodate rows from both conventional and index-organized tables.

If you create your own chained-rows table, then it must follow the format prescribed by one of these two scripts.

If you are analyzing index-organized tables based on primary keys (rather than universal rowids), then you must create a separate chained-rows table for each index-organized table to accommodate its primary-key storage. Use the SQL scripts DBMSIOTC.SQL and PRVTIOTC.PLB to define the BUILD_CHAIN_ROWS_TABLE procedure, and then execute this procedure to create an IOT_CHAINED_ROWS table for each such index-organized table.

See Also:

- The DBMS_IOT package in *Oracle Database PL/SQL Packages and Types Reference* for information on the packaged SQL scripts
- "Listing Chained Rows: Example"

DELETE STATISTICS

Specify DELETE STATISTICS to delete any statistics about the analyzed object that are currently stored in the data dictionary. Use this statement when you no longer want Oracle Database to use the statistics.

When you use this clause on a table, the database also automatically removes statistics for all the indexes defined on the table. When you use this clause on a cluster, the database also automatically removes statistics for all the cluster tables and all their indexes, including the cluster index.

Specify `SYSTEM` if you want Oracle Database to delete only system (not user-defined) statistics. If you omit `SYSTEM`, and if user-defined column or index statistics were collected for an object, then the database also removes the user-defined statistics by invoking the statistics deletion function specified in the statistics type that was used to collect the statistics.

See Also:

"Deleting Statistics: Example"

Examples

Deleting Statistics: Example

The following statement deletes statistics about the sample table `oe.orders` and all its indexes from the data dictionary:

```
ANALYZE TABLE orders DELETE STATISTICS;
```

Analyzing an Index: Example

The following statement validates the structure of the sample index `oe.inv_product_ix`:

```
ANALYZE INDEX inv_product_ix VALIDATE STRUCTURE;
```

Validating a Table: Example

The following statement analyzes the sample table `hr.employees` and all of its indexes:

```
ANALYZE TABLE employees VALIDATE STRUCTURE CASCADE;
```

For a table, the `VALIDATE REF UPDATE` clause verifies the `REF` values in the specified table, checks the rowid portion of each `REF`, and then compares it with the true rowid. If the result is an incorrect rowid, then the `REF` is updated so that the rowid portion is correct.

The following statement validates the `REF` values in the sample table `oe.customers`:

```
ANALYZE TABLE customers VALIDATE REF UPDATE;
```

The following statement validates the structure of the sample table `oe.customers` while allowing simultaneous DML:

```
ANALYZE TABLE customers VALIDATE STRUCTURE ONLINE;
```

Analyzing a Cluster: Example

The following statement analyzes the `personnel` cluster (created in "Creating a Cluster: Example"), all of its tables, and all of their indexes, including the cluster index:

```
ANALYZE CLUSTER personnel
    VALIDATE STRUCTURE CASCADE;
```

Listing Chained Rows: Example

The following statement collects information about all the chained rows in the table `orders`:

```
ANALYZE TABLE orders
   LIST CHAINED ROWS INTO chained_rows;
```

The preceding statement places the information into the table `chained_rows`. You can then examine the rows with this query (no rows will be returned if the table contains no chained rows):

```
SELECT owner_name, table_name, head_rowid, analyze_timestamp
    FROM chained_rows
    ORDER BY owner_name, table_name, head_rowid, analyze_timestamp;

OWNER_NAME  TABLE_NAME  HEAD_ROWID         ANALYZE_TIMESTAMP
----------  ----------  ------------------ -----------------
OE          ORDERS      AAAAZzAABAAABrXAAA 25-SEP-2000
```

ASSOCIATE STATISTICS

Purpose

Use the `ASSOCIATE STATISTICS` statement to associate a statistics type (or default statistics) containing functions relevant to statistics collection, selectivity, or cost with one or more columns, standalone functions, packages, types, domain indexes, or indextypes.

For a listing of all current statistics type associations, query the `USER_ASSOCIATIONS` data dictionary view. If you analyze the object with which you are associating statistics, then you can also query the associations in the `USER_USTATS` view.

See Also:

ANALYZE for information on the order of precedence with which `ANALYZE` uses associations

Prerequisites

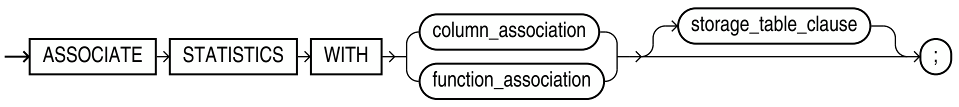
To issue this statement, you must have the appropriate privileges to alter the base object (table, function, package, type, domain index, or indextype). In addition, unless you are associating only default statistics, you must have execute privilege on the statistics type. The statistics type must already have been defined.

See Also:

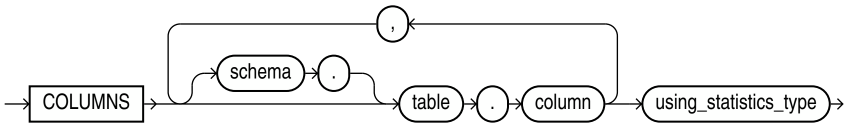
CREATE TYPE for information on defining types

Syntax

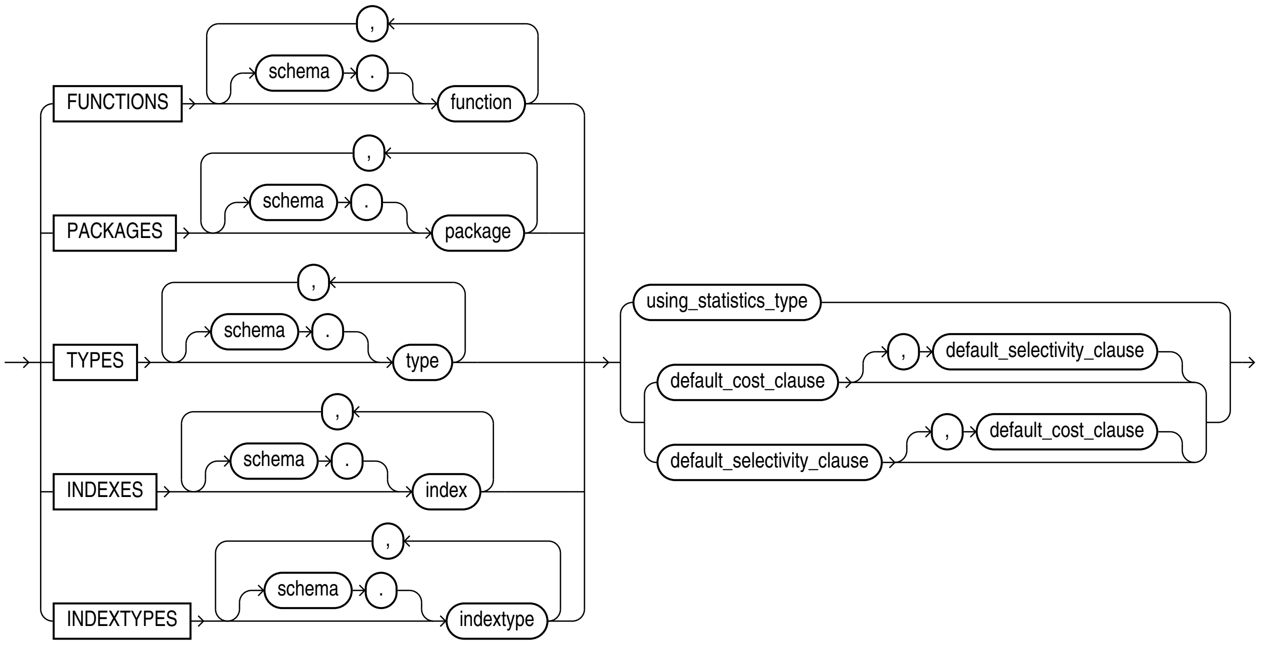
***associate_statistics*::=**



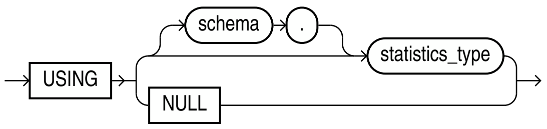
***column_association*::=**



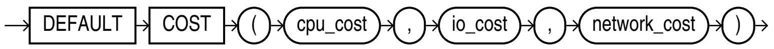
***function_association*::=**



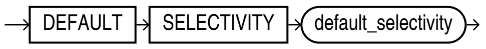
***using_statistics_type*::=**



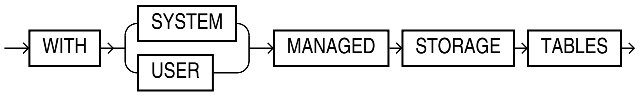
***default_cost_clause*::=**



***default_selectivity_clause*::=**



***storage_table_clause*::=**



Semantics

column_association

Specify one or more table columns. If you do not specify `schema`, then Oracle Database assumes the table is in your own schema.

function_association

Specify one or more standalone functions, packages, user-defined data types, domain indexes, or indextypes. If you do not specify `schema`, then Oracle Database assumes the object is in your own schema.

- `FUNCTIONS` refers only to standalone functions, not to method types or to built-in functions.
- `TYPES` refers only to user-defined types, not to built-in SQL data types.

Restriction on *function_association*

You cannot specify an object for which you have already defined an association. You must first disassociate the statistics from this object.

See Also:

DISASSOCIATE STATISTICS "Associating Statistics: Example"

using_statistics_type

Specify the statistics type (or a synonym for the type) being associated with column, function, package, type, domain index, or indextype. The `statistics_type` must already have been created.

The `NULL` keyword is valid only when you are associating statistics with a column or an index. When you associate a statistics type with an object type, columns of that object type inherit the statistics type. Likewise, when you associate a statistics type with an indextype, index instances of the indextype inherit the statistics type.You can override this inheritance by associating a different statistics type for the column or index. Alternatively, if you do not want to associate any statistics type for the column or index, then you can specify `NULL` in the `using_statistics_type` clause.

Restriction on Specifying Statistics Type

You cannot specify `NULL` for functions, packages, types, or indextypes.

See Also:

Oracle Database Data Cartridge Developer's Guide for information on creating statistics collection functions

default_cost_clause

Specify default costs for standalone functions, packages, types, domain indexes, or indextypes. If you specify this clause, then you must include one number each for CPU cost, I/O cost, and network cost, in that order. Each cost is for a single execution of the function or method or for a single domain index access. Accepted values are integers of zero or greater.

default_selectivity_clause

Specify as a percent the default selectivity for predicates with standalone functions, types, packages, or user-defined operators. The `default_selectivity_clause` must be a number between 0 and 100. Values outside this range are ignored.

Restriction on the *default_selectivity_clause*

You cannot specify `DEFAULT SELECTIVITY` for domain indexes or indextypes.

See Also:

"Specifying Default Cost: Example"

storage_table_clause

This clause is relevant only for statistics on `INDEXTYPE`.

- Specify `WITH SYSTEM MANAGED STORAGE TABLES` to indicate that the storage of statistics data is to be managed by the system. The type you specify in `statistics_type` should be storing the statistics related information in tables that are maintained by the system. Also, the indextype you specify must already have been created or altered to support the `WITH SYSTEM MANAGED STORAGE TABLES` clause.
- Specify `WITH USER MANAGED STORAGE TABLES` to indicate that the tables that store the user-defined statistics will be managed by the user. This is the default behavior.

Examples

Associating Statistics: Example

This statement creates an association for the standalone package `emp_mgmt`. See *Oracle Database PL/SQL Language Reference* for the example that creates this package.

```
ASSOCIATE STATISTICS WITH PACKAGES emp_mgmt DEFAULT SELECTIVITY 10;
```

Specifying Default Cost: Example

This statement specifies that using the domain index `salary_index`, created in "Using Extensible Indexing ", to implement a given predicate always has a CPU cost of 100, I/O cost of 5, and network cost of 0.

```
ASSOCIATE STATISTICS WITH INDEXES salary_index DEFAULT COST (100,5,0);
```

The optimizer will use these default costs instead of calling a cost function.

AUDIT (Traditional Auditing)

Note:

Traditional Auditing is desupported in Oracle Database Release 23. Oracle recommends that you use unified auditing instead.

AUDIT (Unified Auditing)

This section describes the `AUDIT` 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 `AUDIT` statement to:

- Enable a unified audit policy for all users or for specified users
- Specify whether an audit record is created if the audited event fails, succeeds, or both
- Specify application context attributes, whose values will be recorded in audit records

Changes made to the audit policy become effective immediately in the current session and in all active sessions without re-login.

See Also:

- NOAUDIT (Unified Auditing)
- CREATE AUDIT POLICY (Unified Auditing)
- ALTER AUDIT POLICY (Unified Auditing)
- DROP AUDIT POLICY (Unified Auditing)

Prerequisites

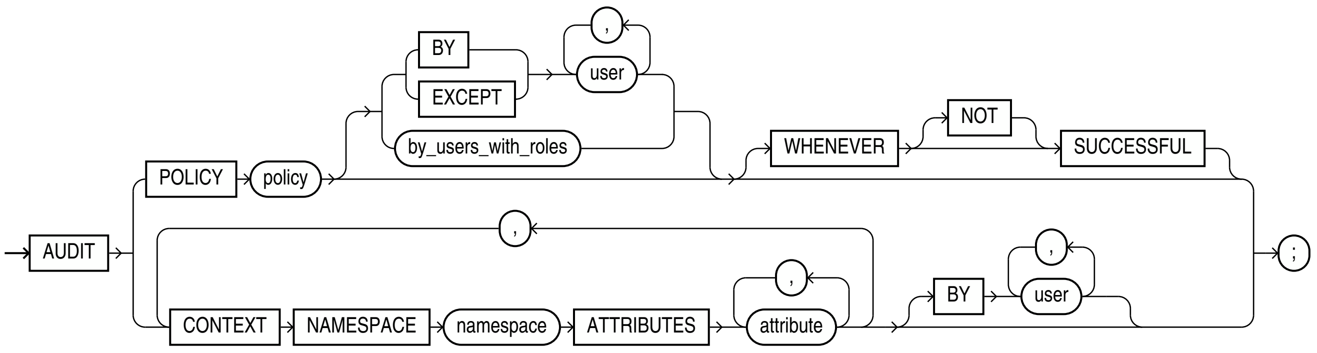
You must have the `AUDIT SYSTEM` system privilege or the `AUDIT_ADMIN` role.

If you are connected to a multitenant container database (CDB), then to enable 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 enable 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.

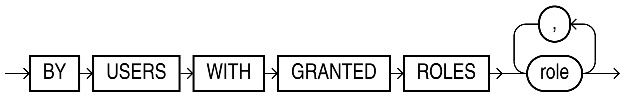
To specify the `AUDIT CONTEXT` ... statement when connected to a CDB, you must have the commonly granted `AUDIT SYSTEM` privilege or the `AUDIT_ADMIN` common role, or you must have the locally granted `AUDIT SYSTEM` privilege or the `AUDIT_ADMIN` local role in the current session's container.

Syntax

***unified_audit*::=**



***by_users_with_roles*::=**



Semantics

policy

With Oracle Database Release 21c unified audit policies are enforced on the current user who executes the SQL statement.

Specify the name of the unified audit policy to be enabled. (The policy must be created previously by the `CREATE AUDIT POLICY` statement.) The policy becomes active immediately for the current session and active ongoing sessions as soon as the `AUDIT POLICY` statement is executed.

You can find descriptions of all unified audit policies by querying the `AUDIT_UNIFIED_POLICIES` view and descriptions of all *enabled* unified audit policies by querying the `AUDIT_UNIFIED_ENABLED_POLICIES` view.

When you enable a unified audit policy, all SQL statements and operations that satisfy either a system privilege or action or role audit option specified in the enabled policy will be audited—that is, a unified audit record will be created in the `UNIFIED_AUDIT_TRAIL` view. If a single SQL statement or operation satisfies multiple enabled policies, then only one unified audit record will be created and all satisfied audit policy names will appear in a comma-separated list in the `UNIFIED_AUDIT_POLICIES` column of the `UNIFIED_AUDIT_TRAIL` view.

See Also:

- CREATE AUDIT POLICY (Unified Auditing)
- *Oracle Database Reference* for more information on the `AUDIT_UNIFIED_POLICIES`, `AUDIT_UNIFIED_ENABLED_POLICIES`, and `UNIFIED_AUDIT_TRAIL` views

BY | EXCEPT

Specify the `BY` clause to enable *`policy`* for only the specified users.

Specify the `EXCEPT` clause to enable *`policy`* for all users except the specified users.

If you omit the `BY` and `EXCEPT` clauses and the *`by_users_with_roles`* clause, then Oracle Database enables *`policy`* for all users.

If *`policy`* is a common unified audit policy, then *`user`* must be a common user. If *`policy`* is a local unified audit policy, then *`user`* must be a common user or a local user in the container to which you are connected.

Notes on the BY and EXCEPT Clauses

The following notes apply to the `BY` and `EXCEPT` clauses:

- If multiple `AUDIT` ... `BY` ... statements are specified for the same unified audit policy, then the policy is enabled for the union of the users specified in each statement.
- If multiple `AUDIT` ... `EXCEPT` ... statements are specified for the same unified audit policy, then only the most recently specified statement takes effect. That is, the policy is enabled for all users except the users specified in the most recent `AUDIT` ... `EXCEPT` ... statement.
- If a policy is enabled using the `BY` clause and you would like to instead enable it using the `EXCEPT` clause, then you must first use the `NOAUDIT` ... `BY` ... statement to disable the policy for all users for whom the policy is currently enabled, and then enable the policy with the `AUDIT` ... `EXCEPT` ... statement.
- If a policy is enabled using the `EXCEPT` clause and you would like to instead enable it using the `BY` clause, then you must first use the `NOAUDIT` statement to disable the audit policy. Note that you cannot specify the `EXCEPT` clause with the `NOAUDIT` statement. You can then enable the policy with the `AUDIT` ... `BY` ... statement.

Restriction on the BY and EXCEPT Clauses

You cannot specify an `AUDIT` ... `BY` ... statement and an `AUDIT` ... `EXCEPT` ... statement for the same unified audit policy. If you attempt to do so, then an error occurs.

by_users_with_roles

Specify this clause to enable *`policy`* for users who have been directly or indirectly granted the specified roles. If you subsequently grant one of the roles to an additional user or to a role which is directly or indirectly granted to a user, then the policy automatically applies to that user. If you subsequently revoke one of the roles from a user or from a role which was directly or indirectly granted to a role or a user, then the policy no longer applies to that user.

When you are connected to a CDB, if *`policy`* is a common unified audit policy, then *`role`* must be a common role. If *`policy`* is a local unified audit policy, then *`role`* must be a common role or a local role in the container to which you are connected.

Enabling a Local Audit Policy on Roles

Local audit policy can be enabled on local roles as well as on common roles. When a local audit policy is enabled on a common role, it generates audit records when a common role is granted to user locally or commonly in the container.

Enabling a Common Audit Policy on Roles

Common audit policy can only be enabled on common roles. When a common audit policy is enabled on a common role, it generates audit records when a common role is granted to an user commonly or locally in the ROOT container.

WHENEVER [NOT] SUCCESSFUL

Specify `WHENEVER SUCCESSFUL` to audit only SQL statements and operations that succeed.

Specify `WHENEVER NOT SUCCESSFUL` to audit only SQL statements and operations that fail or result in errors.

If you omit this clause, then Oracle Database performs the audit regardless of success or failure.

CONTEXT Clause

Specify the `CONTEXT` clause to include the values of context attributes in audit records.

- For *`namespace`*, specify the context namespace.
- For *`attribute`*, specify one or more context attributes whose values you want to include in audit records.
- Use the optional `BY` *`user`* clause to include the values of the context attributes only in audit records for events executed by the specified users. If you omit the `BY` clause, then the values of the context attributes are included in all audit records.

If you specify the `CONTEXT` clause when the current container is the root of a CDB, then the values of context attributes will be included in audit records only for events executed in the root. If you specify the optional `BY` clause, then *`user`* must be a common user.

If you specify the `CONTEXT` clause when the current container is a pluggable database (PDB), then the values of context attributes will be included in audit records only for events executed in that PDB. If you specify the optional `BY` clause, then *`user`* must be a common user or a local user in that PDB.

You can find the application context attributes that are configured to be captured in the audit trail by querying the `AUDIT_UNIFIED_CONTEXTS` view.

See Also:

Oracle Database Reference for more information on the `AUDIT_UNIFIED_CONTEXTS` view

Examples

The following examples enable unified audit policies that were created in the `CREATE AUDIT POLICY` "Examples".

Enabling a Unified Audit Policy for All Users: Example

The following statement enables unified audit policy `table_pol` for all users:

```
AUDIT POLICY table_pol;
```

The following statement verifies that `table_pol` is enabled for all users:

```
SELECT policy_name, enabled_option, entity_name
  FROM audit_unified_enabled_policies
  WHERE policy_name = 'TABLE_POL';

POLICY_NAME  ENABLED_OPTION  ENTITY_NAME
-----------  -----------  ---------
TABLE_POL    BY           ALL USERS
```

Enabling a Unified Audit Policy for Specific Users: Examples

The following statement enables unified audit policy `dml_pol` for only users `hr` and `sh`:

```
AUDIT POLICY dml_pol BY hr, sh;
```

The following statement verifies that `dml_pol` is enabled for only users `hr` and `sh`:

```
SELECT policy_name, enabled_option, entity_name
  FROM audit_unified_enabled_policies
  WHERE policy_name = 'DML_POL'
  ORDER BY entity_name;

POLICY_NAME  ENABLED_OPTION  ENTITY_NAME
-----------  -----------  ---------
DML_POL      BY           HR
DML_POL      BY           SH
```

The following statement enables unified audit policy `read_dir_pol` for all users except `hr`:

```
AUDIT POLICY read_dir_pol EXCEPT hr;
```

The following statement verifies that `read_dir_pol` is enabled for all users except `hr`:

```
SELECT policy_name, enabled_option, entity_name
  FROM audit_unified_enabled_policies
  WHERE policy_name = 'READ_DIR_POL';

POLICY_NAME   ENABLED_OPTION  ENTITY_NAME
------------  -----------  ---------
READ_DIR_POL  EXCEPT       HR
```

The following statement enables unified audit policy `security_pol` for user `hr` and audits only the SQL statements and operations that fail:

```
AUDIT POLICY security_pol BY hr WHENEVER NOT SUCCESSFUL;
```

The following statement verifies that `security_pol` is enabled for only user `hr` and that only the SQL statements and operations that fail will be audited:

```
SELECT policy_name, enabled_option, entity_name, success, failure
  FROM audit_unified_enabled_policies
  WHERE policy_name = 'SECURITY_POL';

POLICY_NAME   ENABLED_OPTION          ENTITY_NAME   SUCCESS  FAILURE
------------  --------------------  ----------  -------  -------
SECURITY_POL  BY                    HR          NO       YES
```

Including Values of Context Attributes in Audit Records: Example

The following statement instructs the database to include the values of namespace `USERENV` attributes `CURRENT_USER` and `DB_NAME` in all audit records for user `hr`:

```
AUDIT CONTEXT NAMESPACE userenv
  ATTRIBUTES current_user, db_name
  BY hr;
```

CALL

Purpose

Use the `CALL` statement to execute a **routine** (a standalone procedure or function, or a procedure or function defined within a type or package) from within SQL.

Note:

The restrictions on user-defined function expressions specified in "Function Expressions " apply to the `CALL` statement as well.

See Also:

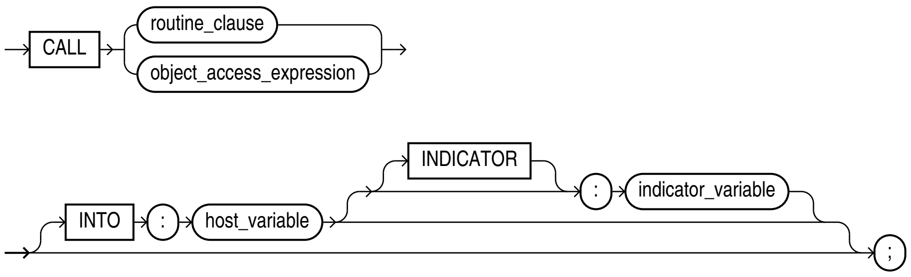
Oracle Database PL/SQL Language Reference for information on creating such routine

Prerequisites

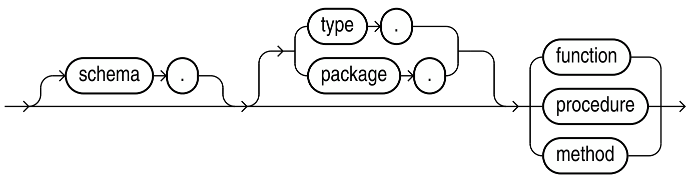
You must have `EXECUTE` privilege on the standalone routine or on the type or package in which the routine is defined.

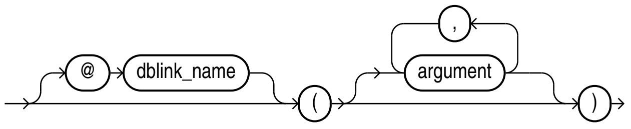
Syntax

call::=

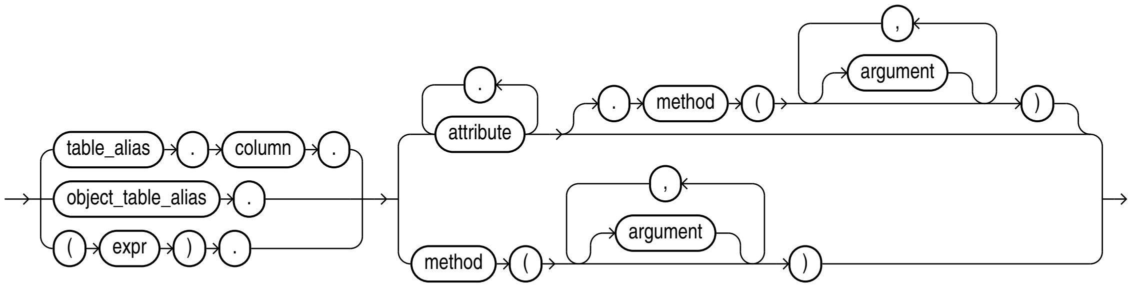


***routine_clause*::=**





***object_access_expression*::=**



Semantics

You can execute a routine in two ways. You can issue a call to the routine itself by name, by using the `routine_clause`, or you can invoke a routine inside the type of an expression, by using an `object_access_expression`.

routine_clause

Specify the name of the function or procedure being called, or a synonym that resolves to a function or procedure.

When you call a member function or procedure of a type, if the first argument (`SELF`) is a null `IN OUT` argument, then Oracle Database returns an error. If `SELF` is a null `IN` argument, then the database returns null. In both cases, the function or procedure is not invoked.

Restriction on Functions

If the routine is a function, then the `INTO` clause is required.

schema

Specify the schema in which the standalone routine, or the package or type containing the routine, resides. If you do not specify `schema`, then Oracle Database assumes the routine is in your own schema.

type or *package*

Specify the type or package in which the routine is defined.

@dblink

In a distributed database system, specify the name of the database containing the standalone routine, or the package or function containing the routine. If you omit *`dblink`*, then Oracle Database looks in your local database.

> **See Also:**
>
> "Calling a Procedure: Example" for an example of calling a routine directly

object_access_expression

If you have an expression of an object type, such as a type constructor or a bind variable, then you can use this form of expression to call a routine defined within the type. In this context, the *`object_access_expression`* is limited to method invocations.

> **See Also:**
>
> "Object Access Expressions " for syntax and semantics of this form of expression, and "Calling a Procedure Using an Expression of an Object Type: Example" for an example of calling a routine using an expression of an object type

argument

Specify one or more arguments to the routine, if the routine takes arguments. You can use positional, named, or mixed notation for *`argument`*. For example, all of the following notations are correct:

```
CALL my_procedure(arg1 => 3, arg2 => 4)

CALL my_procedure(3, 4)

CALL my_procedure(3, arg2 => 4)
```

Restrictions on Applying Arguments to Routines

The *`argument`* is subject to the following restrictions:

- The data types of the parameters passed by the `CALL` statement must be SQL data types. They cannot be PL/SQL-only data types such as `BOOLEAN`.
- An *`argument`* cannot be a pseudocolumn or either of the object reference functions `VALUE` or `REF`.
- Any *`argument`* that is an `IN OUT` or `OUT` argument of the routine must correspond to a host variable expression.
- The number of arguments, including any return argument, is limited to 1000.
- You cannot bind arguments of character and raw data types (`CHAR`, `VARCHAR2`, `NCHAR`, `NVARCHAR2`, `RAW`, `LONG RAW`) that are larger than 4K.

INTO :*host_variable*

The `INTO` clause applies only to calls to functions. Specify which host variable will store the return value of the function.

:*indicator_variable*

Specify the value or condition of the host variable.

See Also:

*Pro*C/C++ Programmer's Guide* for more information on host variables and indicator variables

Examples

Calling a Procedure: Example

The following statement removes the Entertainment department (created in "Inserting Sequence Values: Example") using uses the `remove_dept` procedure. See *Oracle Database PL/SQL Language Reference* for the example that creates this procedure.

```
CALL emp_mgmt.remove_dept(162);
```

Calling a Procedure Using an Expression of an Object Type: Example

The following examples show how call a procedure by using an expression of an object type in the `CALL` statement. The example uses the `warehouse_typ` object type in the order entry sample schema `OE`:

```
ALTER TYPE warehouse_typ
      ADD MEMBER FUNCTION ret_name
      RETURN VARCHAR2
      CASCADE;

CREATE OR REPLACE TYPE BODY warehouse_typ
      AS MEMBER FUNCTION ret_name
      RETURN VARCHAR2
      IS
         BEGIN
            RETURN self.warehouse_name;
         END;
      END;
/
VARIABLE x VARCHAR2(25);

CALL warehouse_typ(456, 'Warehouse 456', 2236).ret_name()
   INTO :x;

PRINT x;
X
--------------------------------
Warehouse 456
```

The next example shows how to use an external function to achieve the same thing:

```
CREATE OR REPLACE FUNCTION ret_warehouse_typ(x warehouse_typ)
  RETURN warehouse_typ
  IS
    BEGIN
      RETURN x;
    END;
/
CALL ret_warehouse_typ(warehouse_typ(234, 'Warehouse 234',
   2235)).ret_name()
   INTO :x;

PRINT x;

X
--------------------------------
Warehouse 234
```

COMMENT

Purpose

Use the `COMMENT` statement to add to the data dictionary a comment about a table or table column, unified audit policy, edition, indextype, materialized view, mining model, operator, or view.

To drop a comment from the database, set it to the empty string ' '.

See Also:

- "Comments " for more information on associating comments with SQL statements and schema objects
- *Oracle Database Reference* for information on the data dictionary views that display comments

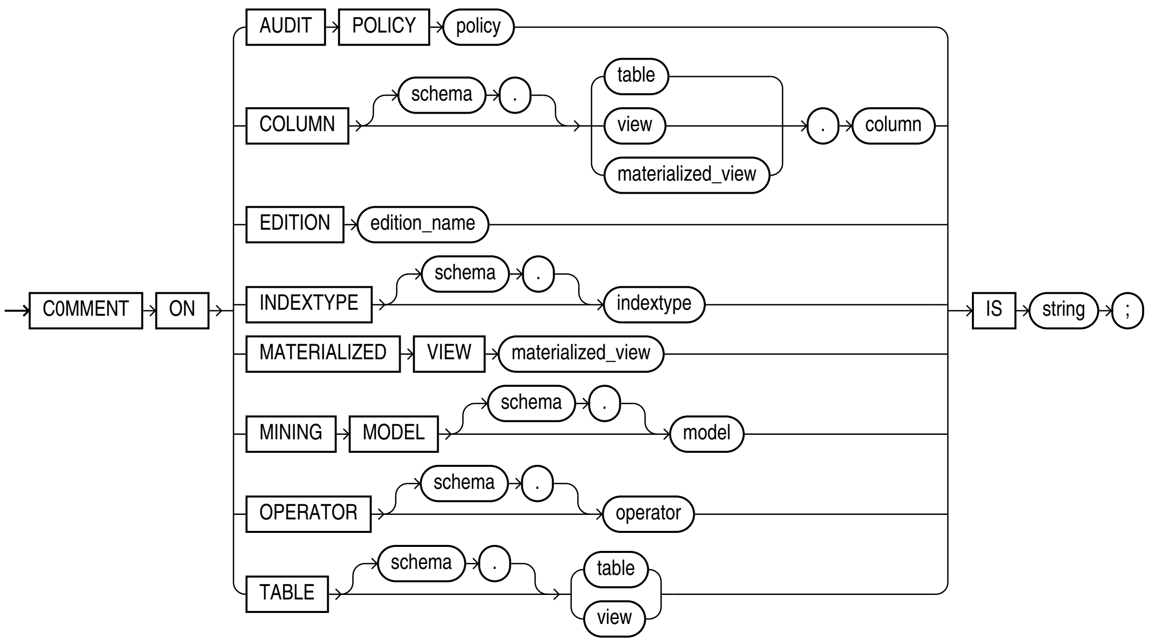
Prerequisites

The object about which you are adding a comment must be in your own schema or:

- To add a comment to a table, view, or materialized view, you must have `COMMENT ANY TABLE` system privilege.
- To add a comment to a unified audit policy, you must have the `AUDIT SYSTEM` system privilege or the `AUDIT_ADMIN` role.
- To add a comment to an edition, you must have the `CREATE ANY EDITION` system privilege, granted either directly or through a role.
- To add a comment to an indextype, you must have the `CREATE ANY INDEXTYPE` system privilege.
- To add a comment to a mining model, you must have the `COMMENT ANY MINING MODEL` system privilege.
- To add a comment to an operator, you must have the `CREATE ANY OPERATOR` system privilege.

Syntax

***comment*::=**



Semantics

AUDIT POLICY Clause

Specify the name of the unified audit policy to be commented.

You can view the comments on a particular unified audit policy by querying the `AUDIT_UNIFIED_POLICY_COMMENTS` data dictionary view.

COLUMN Clause

Specify the name of the column of a table, view, or materialized view to be commented. If you omit *`schema`*, then Oracle Database assumes the table, view, or materialized view is in your own schema.

You can view the comments on a particular table or column by querying the data dictionary views `USER_TAB_COMMENTS`, `DBA_TAB_COMMENTS`, or `ALL_TAB_COMMENTS` or `USER_COL_COMMENTS`, `DBA_COL_COMMENTS`, or `ALL_COL_COMMENTS`.

EDITION Clause

Specify the name of an existing edition to be commented.

You can query the data dictionary view `ALL_EDITION_COMMENTS` to view comments associated with editions that are accessible to the current user. You can query `DBA_EDITION_COMMENTS` to view comments associated with all editions in the database.

TABLE Clause

Specify the schema and name of the table or materialized view to be commented. If you omit *`schema`*, then Oracle Database assumes the table or materialized view is in your own schema.

Note:

In earlier releases, you could use this clause to create a comment on a materialized view. You should now use the `COMMENT ON MATERIALIZED VIEW` clause for materialized views.

INDEXTYPE Clause

Specify the name of the indextype to be commented. If you omit *`schema`*, then Oracle Database assumes the indextype is in your own schema.

You can view the comments on a particular indextype by querying the data dictionary views `USER_INDEXTYPE_COMMENTS`, `DBA_INDEXTYPE_COMMENTS`, or `ALL_INDEXTYPE_COMMENTS`.

MATERIALIZED VIEW Clause

Specify the name of the materialized view to be commented. If you omit *`schema`*, then Oracle Database assumes the materialized view is in your own schema.

You can view the comments on a particular materialized view by querying the data dictionary views `USER_MVIEW_COMMENTS`, `DBA_MVIEW_COMMENTS`, or `ALL_MVIEW_COMMENTS`.

MINING MODEL

Specify the name of the mining model to be commented.

You can view the comments on a particular mining model by querying the `COMMENTS` column of the data dictionary views `USER_MINING_MODELS`, `DBA_MINING_MODELS`, or `ALL_MINING_MODELS`.

OPERATOR Clause

Specify the name of the operator to be commented. If you omit *`schema`*, then Oracle Database assumes the operator is in your own schema.

You can view the comments on a particular operator by querying the data dictionary views `USER_OPERATOR_COMMENTS`, `DBA_OPERATOR_COMMENTS`, or `ALL_OPERATOR_COMMENTS`.

IS '*string*'

Specify the text of the comment. Refer to "Text Literals " for a syntax description of *`'string'`*.

Examples

Creating Comments: Example

To insert an explanatory remark on the `job_id` column of the `employees` table, you might issue the following statement:

```
COMMENT ON COLUMN employees.job_id
   IS 'abbreviated job title';
```

To drop this comment from the database, issue the following statement:

```
COMMENT ON COLUMN employees.job_id IS '';
```