8
Common SQL DDL Clauses

This chapter describes some SQL data definition clauses that appear in multiple SQL statements.

This chapter contains these sections:

allocate_extent_clause

Purpose

Use the *allocate_extent_clause* clause to explicitly allocate a new extent for a database object.

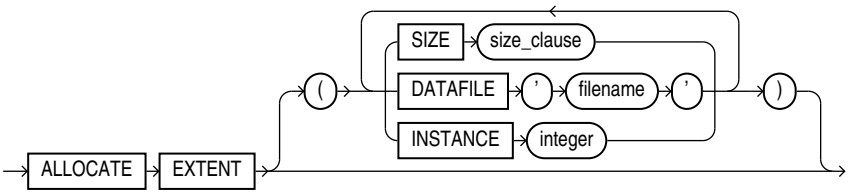
Explicitly allocating an extent with this clause does not change the values of the NEXT and PCTINCREASE storage parameters, so does not affect the size of the next extent to be allocated implicitly by Oracle Database. Refer to *storage_clause* for information about the NEXT and PCTINCREASE storage parameters.

You can allocate an extent in the following SQL statements:

- ALTER CLUSTER (see ALTER CLUSTER)
- ALTER INDEX: to allocate an extent to the index, an index partition, or an index subpartition (see ALTER INDEX)
- ALTER MATERIALIZED VIEW: to allocate an extent to the materialized view, one of its partitions or subpartitions, or the overflow segment of an index-organized materialized view (see ALTER MATERIALIZED VIEW)
- ALTER MATERIALIZED VIEW LOG (see ALTER MATERIALIZED VIEW LOG)
- ALTER TABLE: to allocate an extent to the table, a table partition, a table subpartition, the mapping table of an index-organized table, the overflow segment of an index-organized table, or a LOB storage segment (see ALTER TABLE)

Syntax

***allocate_extent_clause*::=**



(*size_clause*::=)

Semantics

This section describes the parameters of the *allocate_extent_clause*. For additional information, refer to the SQL statement in which you set or reset these parameters for a particular database object.

You cannot specify the *allocate_extent_clause* and the *deallocate_unused_clause* in the same statement.

SIZE

Specify the size of the extent in bytes. The value of *integer* can be 0 through 2147483647. To specify a larger extent size, use an integer within this range with `K`, `M`, `G`, or `T` to specify the extent size in kilobytes, megabytes, gigabytes, or terabytes.

For a table, index, materialized view, or materialized view log, if you omit `SIZE`, then Oracle Database determines the size based on the values of the storage parameters of the object. However, for a cluster, Oracle does not evaluate the cluster's storage parameters, so you must specify `SIZE` if you do not want Oracle to use a default value.

DATAFILE '*filename*'

Specify one of the data files in the tablespace of the table, cluster, index, materialized view, or materialized view log to contain the new extent. If you omit `DATAFILE`, then Oracle chooses the data file.

INSTANCE *integer*

Use this parameter only if you are using Oracle Real Application Clusters.

Specifying `INSTANCE` *integer* makes the new extent available to the freelist group associated with the specified instance. If the instance number exceeds the maximum number of freelist groups, then Oracle divides the specified number by the maximum number and uses the remainder to identify the freelist group to be used. An instance is identified by the value of its initialization parameter `INSTANCE_NUMBER`.

If you omit this parameter, then the space is allocated to the table, cluster, index, materialized view, or materialized view log but is not drawn from any particular freelist group. Instead, Oracle uses the master freelist and allocates space as needed.

> **Note:**
>
> If you are using automatic segment-space management, then the `INSTANCE` parameter of the *`allocate_extent_clause`* may not reserve the newly allocated space for the specified instance, because automatic segment-space management does not maintain rigid affinity between extents and instances.

constraint

Purpose

Use a *`constraint`* to define an **integrity constraint—**a rule that restricts the values in a database. Oracle Database lets you create six types of constraints and lets you declare them in two ways.

The six types of integrity constraint are described briefly here and more fully in "Semantics":

- A **`NOT NULL` constraint** prohibits a database value from being null.
- A **unique constraint** prohibits multiple rows from having the same value in the same column or combination of columns but allows some values to be null.
- A **primary key constraint** combines a `NOT NULL` constraint and a unique constraint in a single declaration. It prohibits multiple rows from having the same value in the same column or combination of columns and prohibits values from being null.
- A **foreign key constraint** requires values in one table to match values in another table.
- A **check constraint** requires a value in the database to comply with a specified condition.
- A `REF` column by definition references an object in another object type or in a relational table. A **REF constraint** lets you further describe the relationship between the `REF` column and the object it references.

You can define constraints syntactically in two ways:

- As part of the definition of an individual column or attribute. This is called **inline** specification.
- As part of the table definition. This is called **out-of-line** specification.

`NOT NULL` constraints must be declared inline. All other constraints can be declared either inline or out of line.

Constraint clauses can appear in the following statements:

- `CREATE TABLE` (see CREATE TABLE)
- `ALTER TABLE` (see ALTER TABLE)
- `CREATE VIEW` (see CREATE VIEW)
- `ALTER VIEW` (see ALTER VIEW)

View Constraints

Oracle Database does not enforce view constraints. However, you can enforce constraints on views through constraints on base tables.

You can specify only unique, primary key, and foreign key constraints on views, and they are supported only in `DISABLE NOVALIDATE` mode. You cannot define view constraints on attributes of an object column.

> **See Also:**
>
> View Constraints for additional information on view constraints and "DISABLE Clause" for information on `DISABLE NOVALIDATE` mode

External Table Constraints

You can specify only `NOT NULL`, unique, primary key, and foreign key constraints on external tables. Unique, primary key, and foreign key constraints are supported only in `RELY DISABLE` mode.

> **See Also:**
>
> DISABLE Clause for information on `RELY` and `DISABLE`.

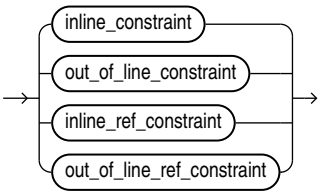
Prerequisites

You must have the privileges necessary to issue the statement in which you are defining the constraint.

To create a foreign key constraint, in addition, the parent table or view must be in your own schema or you must have the `REFERENCES` privilege on the columns of the referenced key in the parent table or view.

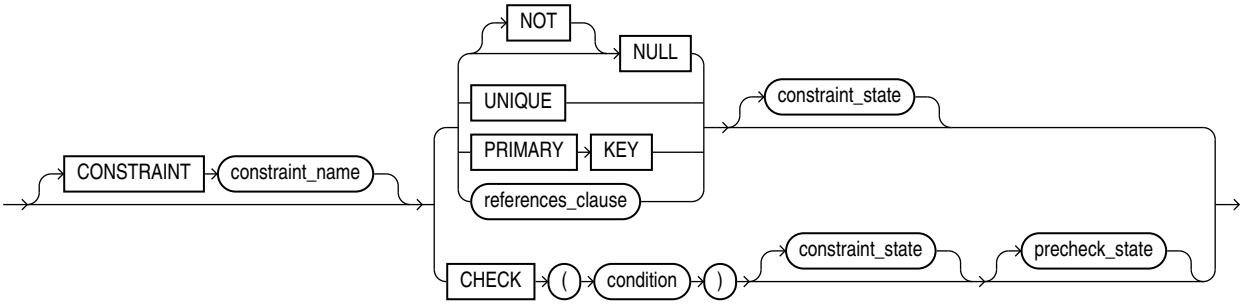
Syntax

***constraint*::=**



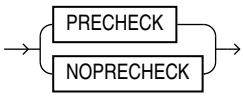
(*inline_constraint*::=, *out_of_line_constraint*::=, *inline_ref_constraint*::=, *out_of_line_ref_constraint*::=)

***inline_constraint*::=**

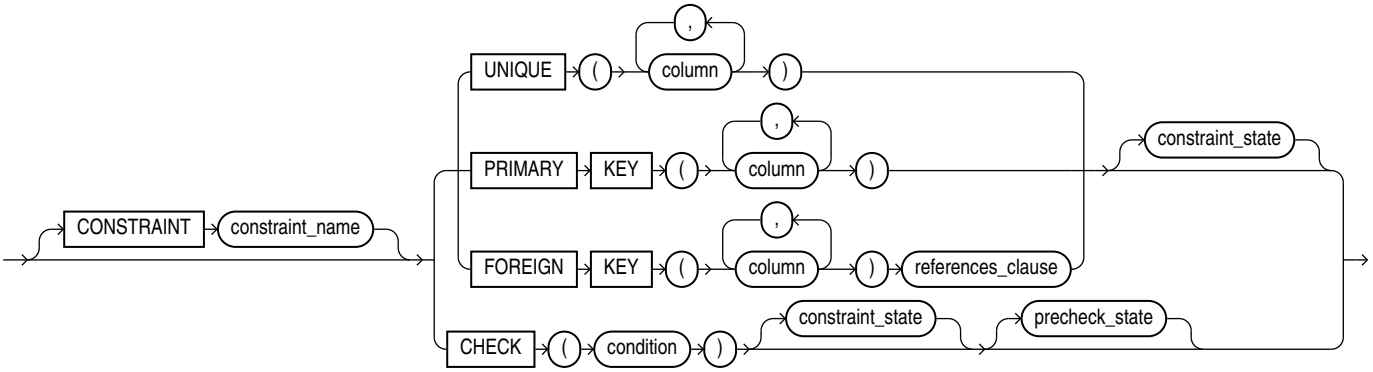


(*references_clause*::=)

***precheck_state*::=**

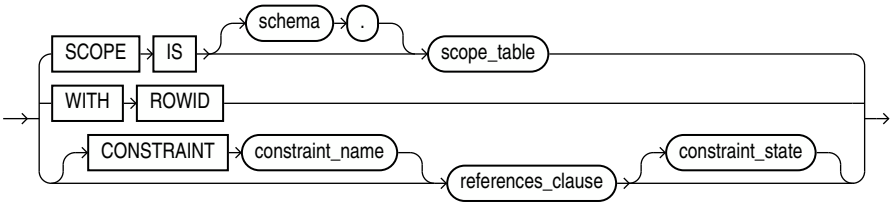


***out_of_line_constraint*::=**



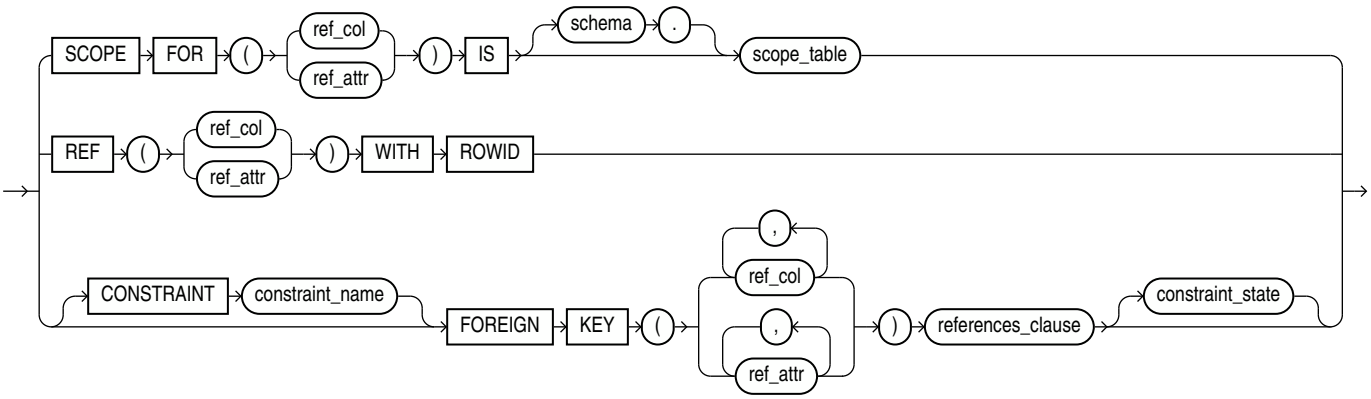
(*references_clause*::=, *constraint_state*::=)

***inline_ref_constraint*::=**



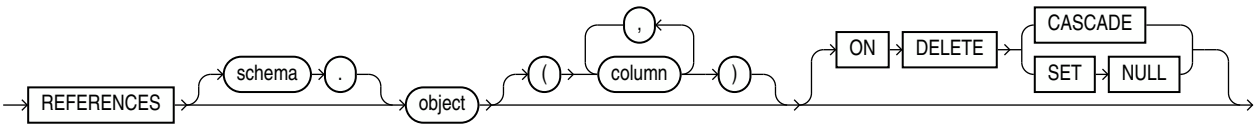
(*references_clause*::=, *constraint_state*::=)

***out_of_line_ref_constraint*::=**

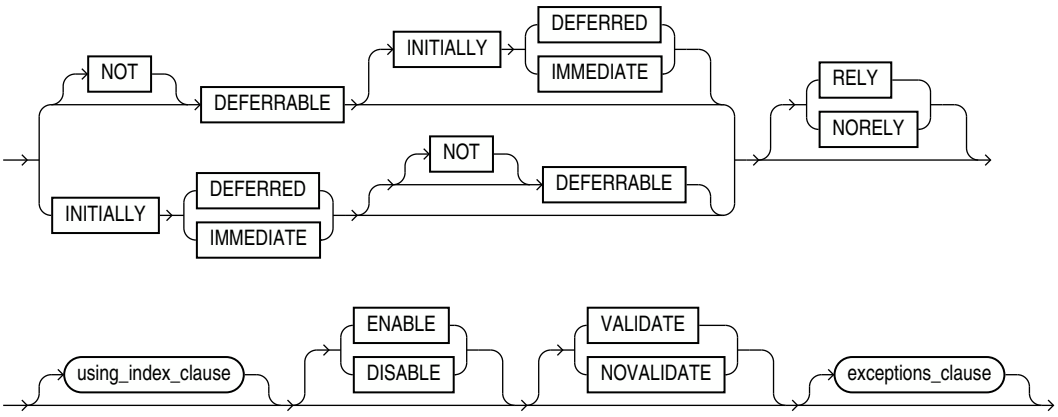


(*references_clause*::=, *constraint_state*::=)

***references_clause*::=**

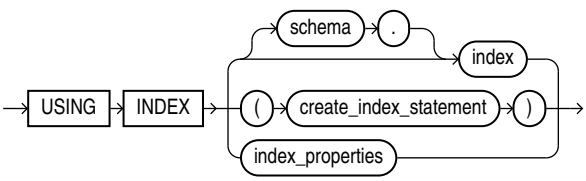


***constraint_state*::=**



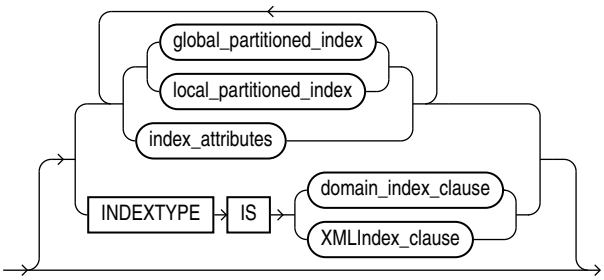
(*using_index_clause*::=, *exceptions_clause*::=)

***using_index_clause*::=**



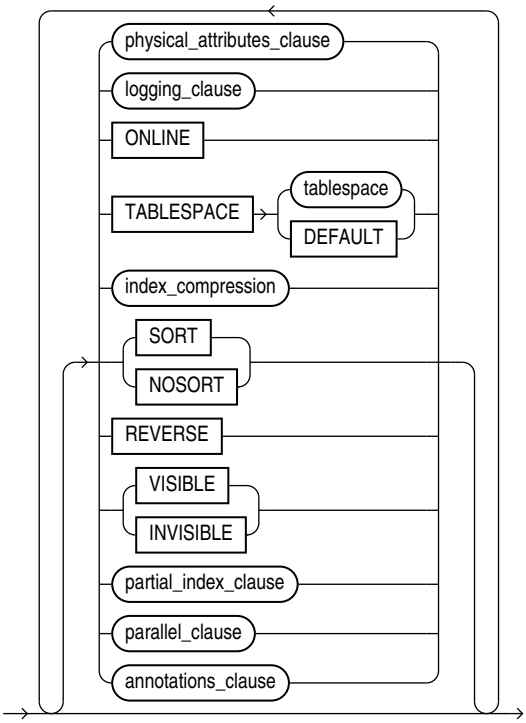
(*create_index*::=, *index_properties*::=)

***index_properties*::=**



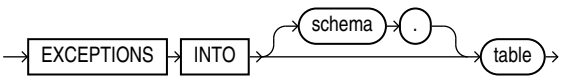
(*global_partitioned_index*::=, *local_partitioned_index*::=--part of `CREATE INDEX`, *index_attributes*::=. The `INDEXTYPE IS` ... clause is not valid when defining a constraint.)

***index_attributes*::=**



(*physical_attributes_clause*::=, *logging_clause*::=, *index_compression*::=, *partial_index_clause*::=--all part of `CREATE INDEX`, *`parallel_clause`*: not supported in *`using_index_clause`*)

***exceptions_clause*::=**



Semantics

This section describes the semantics of *constraint*. For additional information, refer to the SQL statement in which you define or redefine a constraint for a table or view.

Oracle Database does not support constraints on columns or attributes whose type is a user-defined object, nested table, VARRAY, REF, or LOB, with two exceptions:

- NOT NULL constraints are supported for a column or attribute whose type is user-defined object, VARRAY, REF, or LOB.
- NOT NULL, foreign key, and REF constraints are supported on a column of type REF.

CONSTRAINT *constraint_name*

Specify a name for the constraint. The name must satisfy the requirements listed in "Database Object Naming Rules ". If you omit this identifier, then Oracle Database generates a name with the form SYS_C*n*. Oracle stores the name and the definition of the integrity constraint in the USER_, ALL_, and DBA_CONSTRAINTS data dictionary views (in the CONSTRAINT_NAME and SEARCH_CONDITION columns, respectively).

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

NOT NULL Constraints

A NOT NULL constraint prohibits a column from containing nulls. The NULL keyword by itself does not actually define an integrity constraint, but you can specify it to explicitly permit a column to contain nulls. You must define NOT NULL and NULL using inline specification. If you specify neither NOT NULL nor NULL, then the default is NULL.

NOT NULL constraints are the only constraints you can specify inline on XMLType and VARRAY columns.

To satisfy a NOT NULL constraint, every row in the table must contain a value for the column.

> **Note:**
>
> Oracle Database does not index table rows in which all key columns are null except in the case of bitmap indexes. Therefore, if you want an index on all rows of a table, then you must either specify NOT NULL constraints for at least one of the index key columns or create a bitmap index.

Restrictions on NOT NULL Constraints

NOT NULL constraints are subject to the following restrictions:

- You cannot specify NULL or NOT NULL in a view constraint.
- You cannot specify NULL or NOT NULL for an attribute of an object. Instead, use a CHECK constraint with the IS [NOT] NULL condition.

> **See Also:**
>
> "Attribute-Level Constraints Example" and "NOT NULL Example"

Unique Constraints

A **unique** constraint designates a column as a unique key. A **composite unique key** designates a combination of columns as the unique key. When you define a unique constraint inline, you need only the `UNIQUE` keyword. When you define a unique constraint out of line, you must also specify one or more columns. You must define a composite unique key out of line.

To satisfy a unique constraint, no two rows in the table can have the same value for the unique key. However, the unique key made up of a single column can contain nulls. To satisfy a composite unique key, no two rows in the table or view can have the same combination of values in the key columns. Any row that contains nulls in all key columns automatically satisfies the constraint. However, two rows that contain nulls for one or more key columns and the same combination of values for the other key columns violate the constraint.

Unique constraints are sensitive to declared collations of their key columns. See Collation Sensitivity of Constraints for more details.

When you specify a unique constraint on one or more columns, Oracle implicitly creates an index on the unique key. If you are defining uniqueness for purposes of query performance, then Oracle recommends that you instead create the unique index explicitly using a `CREATE UNIQUE INDEX` statement. You can also use the `CREATE UNIQUE INDEX` statement to create a unique function-based index that defines a conditional unique constraint. See "Using a Function-based Index to Define Conditional Uniqueness: Example" for more information.

When you specify an enabled unique constraint on an extended data type column, you may receive a "maximum key length exceeded" error when Oracle tries to create the index to enforce uniqueness for the enabled constraint. See "Creating an Index on an Extended Data Type Column" for information on how to work around this issue.

Restrictions on Unique Constraints

Unique constraints are subject to the following restrictions:

- None of the columns in the unique key can be of LOB, `LONG`, `LONG RAW`, `VARRAY`, `NESTED TABLE`, `OBJECT`, `REF`, `TIMESTAMP WITH TIME ZONE`, or user-defined type. However, the unique key can contain a column of `TIMESTAMP WITH LOCAL TIME ZONE`.
- A composite unique key cannot have more than 32 columns.
- You cannot designate the same column or combination of columns as both a primary key and a unique key.
- You cannot specify a unique key when creating a subview in an inheritance hierarchy. The unique key can be specified only for the top-level (root) view.
- When you specify a unique constraint for an external table, you must specify the `RELY` and `DISABLE` constraint states. See External Table Constraints for more information.

> **See Also:**
>
> "Unique Key Example" and Composite Unique Key Example

Primary Key Constraints

A **primary key** constraint designates a column as the primary key of a table or view. A **composite primary key** designates a combination of columns as the primary key. When you define a primary key constraint inline, you need only the `PRIMARY KEY` keywords. When you define a primary key constraint out of line, you must also specify one or more columns. You must define a composite primary key out of line.

To satisfy a primary key constraint:

- No primary key value can appear in more than one row in the table.
- No column that is part of the primary key can contain a null.

When you create a primary key constraint:

- Oracle Database uses an existing index if it contains a unique set of values before enforcing the primary key constraint. The existing index can be defined as unique or nonunique. When a DML operation is performed, the primary key constraint is enforced using this existing index.
- If no existing index can be used, then Oracle Database generates a unique index.

When you drop a primary key constraint:

- If the primary key was created using an existing index, then the index is not dropped.
- If the primary key was created using a system-generated index, then the index is dropped.

When you designate an extended data type column as an enabled primary key, you may receive a "maximum key length exceeded" error when Oracle tries to create the index to enforce uniqueness for the enabled constraint. See "Creating an Index on an Extended Data Type Column" for information on how to work around this issue.

Primary key constraints are sensitive to declared collations of their key columns. See Collation Sensitivity of Constraints for more details.

Restrictions on Primary Key Constraints

Primary constraints are subject to the following restrictions:

- A table or view can have only one primary key.
- None of the columns in the primary key can be LOB, `LONG`, `LONG RAW`, `VARRAY`, `NESTED TABLE`, `BFILE`, `REF`, `TIMESTAMP WITH TIME ZONE`, or user-defined type. However, the primary key can contain a column of `TIMESTAMP WITH LOCAL TIME ZONE`.
- The size of the primary key cannot exceed approximately one database block.
- A composite primary key cannot have more than 32 columns.
- You cannot designate the same column or combination of columns as both a primary key and a unique key.
- You cannot specify a primary key when creating a subview in an inheritance hierarchy. The primary key can be specified only for the top-level (root) view.
- When you specify a primary key constraint for an external table, you must specify the `RELY` and `DISABLE` constraint states. See External Table Constraints for more information.

> **See Also:**
>
> "Primary Key Example" and "Composite Primary Key Example"

Foreign Key Constraints

A **foreign key constraint** (also called a **referential integrity constraint**) designates a column as the foreign key and establishes a relationship between that foreign key and a specified primary or unique key, called the **referenced key**. A **composite foreign key** designates a combination of columns as the foreign key.

The table or view containing the foreign key is called the **child** object, and the table or view containing the referenced key is called the **parent** object. The foreign key and the referenced key can be in the same table or view. In this case, the parent and child tables are the same. If you identify only the parent table or view and omit the column name, then the foreign key automatically references the primary key of the parent table or view. The corresponding column or columns of the foreign key and the referenced key must match in order, data types, and declared collations.

Foreign key constraints are sensitive to declared collations of the referenced primary or unique key columns. See Collation Sensitivity of Constraints for more details.

You can define a foreign key constraint on a single key column either inline or out of line. You must specify a composite foreign key and a foreign key on an attribute out of line.

To satisfy a composite foreign key constraint, the composite foreign key must refer to a composite unique key or a composite primary key in the parent table or view, or the value of at least one of the columns of the foreign key must be null.

You can designate the same column or combination of columns as both a foreign key and a primary or unique key. You can also designate the same column or combination of columns as both a foreign key and a cluster key.

You can define multiple foreign keys in a table or view. Also, a single column can be part of more than one foreign key.

Restrictions on Foreign Key Constraints

Foreign key constraints are subject to the following restrictions:

- None of the columns in the foreign key can be of LOB, `LONG`, `LONG RAW`, `VARRAY`, `NESTED TABLE`, `BFILE`, `REF`, `TIMESTAMP WITH TIME ZONE`, or user-defined type. However, the primary key can contain a column of `TIMESTAMP WITH LOCAL TIME ZONE`.
- The referenced unique or primary key constraint on the parent table or view must already be defined.
- A composite foreign key cannot have more than 32 columns.
- The child and parent tables must be on the same database. To enable referential integrity constraints across nodes of a distributed database, you must use database triggers. See CREATE TRIGGER .
- If either the child or parent object is a view, then the constraint is subject to all restrictions on view constraints. See "View Constraints ".
- You cannot define a foreign key constraint in a `CREATE TABLE` statement that contains an `AS` *`subquery`* clause. Instead, you must create the table without the constraint and then add it later with an `ALTER TABLE` statement.

- When a table has a foreign key, and the parent of the foreign key is an index-organized table, a session that updates a row that contains the foreign key can hang when another session is updating a non-key column in the parent table.
- When you specify a foreign key constraint for an external table, you must specify the `RELY` and `DISABLE` constraint states. See External Table Constraints for more information.

> **See Also:**
>
> - *Oracle Database Development Guide* for more information on using constraints
> - "Foreign Key Constraint Example" and "Composite Foreign Key Constraint Example"

references_clause

Foreign key constraints use the *`references_clause`* syntax. When you specify a foreign key constraint inline, you need only the *`references_clause`*. When you specify a foreign key constraint out of line, you must also specify the `FOREIGN KEY` keywords and one or more columns.

ON DELETE Clause

The `ON DELETE` clause lets you determine how Oracle Database automatically maintains referential integrity if you remove a referenced primary or unique key value. If you omit this clause, then Oracle does not allow you to delete referenced key values in the parent table that have dependent rows in the child table.

- Specify `CASCADE` if you want Oracle to remove dependent foreign key values.
- Specify `SET NULL` if you want Oracle to convert dependent foreign key values to `NULL`. You cannot specify this clause for a virtual column, because the values in a virtual column cannot be updated directly. Rather, the values from which the virtual column are derived must be updated.

Restriction on ON DELETE

You cannot specify this clause for a view constraint.

> **See Also:**
>
> "ON DELETE Example"

Check Constraints

A check constraint lets you specify a condition that each row in the table must satisfy. To satisfy the constraint, each row in the table must make the condition either `TRUE` or unknown (due to a null). When Oracle evaluates a check constraint condition for a particular row, any column names in the condition refer to the column values in that row.

The syntax for inline and out-of-line specification of check constraints is the same. However, inline specification can refer only to the column (or the attributes of the column if it is an object column) currently being defined, whereas out-of-line specification can refer to multiple columns or attributes.

Oracle does not verify that conditions of check constraints are not mutually exclusive. Therefore, if you create multiple check constraints for a column, design them carefully so their purposes do not conflict. Do not assume any particular order of evaluation of the conditions.

You can specify a check constraint with a precheck state of `PRECHECK`, if you want to be able to validate the constraint outside of the database using an external JSON schema validator.

The SQL conditions used in `CHECK` constraints that have an equivalent condition in the JSON schema vocabulary are supported.

You can specify `PRECHECK` with existing constraint states `ENABLE` and `VALIDATE` at the same time.

If you do not specify `PRECHECK` or `NOPRECHECK` explicitly, Oracle sets the value of `PRECHECK` or `NOPRECHECK` automatically, based on whether a check constraint can be expressed as JSON schema.

The precheck state is independent from existing constraint states. You can use it with an exisiting constraint to indicate that the constraint can be prevalidated outside the database using the JSON schema.

You can remove the `PRECHECK` constraint state by setting it to `NOPRECHECK` using `ALTER TABLE MODIFY CONSTRAINT`.

If the condition of a check constraint depends on NLS parameters, such as `NLS_DATE_FORMAT`, Oracle evaluates the condition using the database values of the parameters, not the session values. You can find the database values of the NLS parameters in the data dictionary view `NLS_DATABASE_PARAMETERS`. These values are associated with a database by the DDL statement `CREATE DATABASE` and never change afterwards.

Restrictions on Check Constraints

Check constraints are subject to the following restrictions:

- You cannot specify a check constraint for a view. However, you can define the view using the `WITH CHECK OPTION` clause, which is equivalent to specifying a check constraint for the view.
- The condition of a check constraint can refer to any column in the table, but it cannot refer to columns of other tables.
- Conditions of check constraints cannot contain the following constructs:
  - Subqueries and scalar subquery expressions
  - Calls to the functions that are not deterministic (`CURRENT_DATE`, `CURRENT_TIMESTAMP`, `DBTIMEZONE`, `LOCALTIMESTAMP`, `SESSIONTIMEZONE`, `SYSDATE`, `SYSTIMESTAMP`, `UID`, `USER`, and `USERENV`)
  - Calls to user-defined functions
  - Dereferencing of `REF` columns (for example, using the `DEREF` function)
  - Nested table columns or attributes
  - The pseudocolumns `CURRVAL`, `NEXTVAL`, `LEVEL`, or `ROWNUM`
  - Date constants that are not fully specified
  - You cannot specify a check constraint for an external table.

> **See Also:**
>
> - Conditions for additional information and syntax
> - "Check Constraint Examples" and "Attribute-Level Constraints Example"
> - PRECHECK Using JSON Schema
> - CHECK Constraint Examples

REF Constraints

`REF` constraints let you describe the relationship between a column of type `REF` and the object it references.

ref_constraint

`REF` constraints use the *`ref_constraint`* syntax. You define a `REF` constraint either inline or out of line. Out-of-line specification requires you to specify the `REF` column or attribute you are further describing.

- For *`ref_column`*, specify the name of a `REF` column of an object or relational table.
- For *`ref_attribute`*, specify an embedded `REF` attribute within an object column of a relational table.

Both inline and out-of-line specification let you define a scope constraint, a rowid constraint, or a referential integrity constraint on a `REF` column.

If the scope table or referenced table of the `REF` column has a primary-key-based object identifier, then the `REF` column is a **user-defined `REF` column**.

> **See Also:**
>
> - *Oracle Database Object-Relational Developer's Guide* for more information on `REF` data types
> - "Foreign Key Constraints", and "REF Constraint Examples"

SCOPE REF Constraints

In a table with a `REF` column, each `REF` value in the column can conceivably reference a row in a different object table. The `SCOPE` clause restricts the scope of references to a single table, *`scope_table`*. The values in the `REF` column or attribute point to objects in *`scope_table`*, in which object instances of the same type as the `REF` column are stored.

Specify the `SCOPE` clause to restrict the scope of references in the `REF` column to a single table. For you to specify this clause, *`scope_table`* must be in your own schema, or you must have the `READ` or `SELECT` privilege on *`scope_table`*, or you must have the `READ ANY TABLE` or `SELECT ANY TABLE` system privilege. You can specify only one scope table for each `REF` column.

Restrictions on Scope Constraints

Scope constraints are subject to the following restrictions:

- You cannot add a scope constraint to an existing column unless the table is empty.
- You cannot specify a scope constraint for the `REF` elements of a `VARRAY` column.
- You must specify this clause if you specify `AS` *`subquery`* and the subquery returns user-defined `REF` data types.
- You cannot subsequently drop a scope constraint from a `REF` column.
- You cannot specify a scope constraint for an external table.

Rowid REF Constraints

Specify `WITH ROWID` to store the rowid along with the `REF` value in *`ref_column`* or *`ref_attribute`*. Storing the rowid with the `REF` value can improve the performance of dereferencing operations, but will also use more space. Default storage of `REF` values is without rowids.

> **See Also:**
>
> The function DEREF for an example of dereferencing

Restrictions on Rowid Constraints

Rowid constraints are subject to the following restrictions:

- You cannot define a rowid constraint for the `REF` elements of a `VARRAY` column.
- You cannot subsequently drop a rowid constraint from a `REF` column.
- If the `REF` column or attribute is scoped, then this clause is ignored and the rowid is not stored with the `REF` value.
- You cannot specify a rowid constraint for an external table.

Referential Integrity Constraints on REF Columns

The *`references_clause`* of the *`ref_constraint`* syntax lets you define a foreign key constraint on the `REF` column. This clause also implicitly restricts the scope of the `REF` column or attribute to the referenced table. However, whereas a foreign key constraint on a non-`REF` column references an actual column in the parent table, a foreign key constraint on a `REF` column references the implicit object identifier column of the parent table.

If you do not specify `a` constraint name, then Oracle generates a system name for the constraint of the form `SYS_Cn`.

If you add a referential integrity constraint to an existing `REF` column that is already scoped, then the referenced table must be the same as the scope table of the `REF` column. If you later drop the referential integrity constraint, then the `REF` column will remain scoped to the referenced table.

As is the case for foreign key constraints on other types of columns, you can use the *`references_clause`* alone for inline declaration. For out-of-line declaration you must also specify the `FOREIGN KEY` keywords plus one or more `REF` columns or attributes.

> **See Also:**
>
> *Oracle Database Object-Relational Developer's Guide* for more information on object identifiers

Restrictions on Foreign Key Constraints on REF Columns

Foreign key constraints on `REF` columns have the following additional restrictions:

- Oracle implicitly adds a scope constraint when you add a referential integrity constraint to an existing unscoped `REF` column. Therefore, all the restrictions that apply for scope constraints also apply in this case.
- You cannot specify a column after the object name in the *`references_clause`*.

Collation Sensitivity of Constraints

Starting with Oracle Database 12*c* Release 2 (12.2), primary key, unique, and foreign key constraints are sensitive to declared collations of their key columns. A primary or unique key character column value from a new or updated row is compared with values in existing rows using the declared collation of the key column. For example, if the declared collation of the key column is the case-insensitive collation `BINARY_CI`, a new or updated row may be rejected if the new key column value differs from some existing key value only by case. The collation `BINARY_CI` treats character values differing only by case as equal.

A foreign key character column value is compared to parent primary or unique key column values using the declared collation of the parent key column. For example, if the declared collation of the key column is the case-insensitive collation `BINARY_CI`, a new or updated child row may be accepted even if there is no identical parent key value for the corresponding foreign key value, provided there exists a value differing only by case.

The declared collation of a foreign key column must be the same as the collation of the corresponding parent key column.

Columns in a composite key of a constraint may have different declared collations.

When the declared collation of a key column of a constraint is a pseudo-collation, the constraint uses a corresponding variant of the collation `BINARY`. Pseudo-collations cannot be used directly to compare values for a constraint, because constraints are static and cannot depend on session NLS parameters on which the pseudo-collations depend. Therefore:

- The pseudo-collations `USING_NLS_COMP`, `USING_NLS_SORT`, and `USING_NLS_SORT_CS` use the collation `BINARY`.
- The pseudo-collation `USING_NLS_COMP_CI` uses the collation `BINARY_CI`.
- The pseudo-collation `USING_NLS_COMP_AI` uses the collation `BINARY_AI`.

When the effective collation used by a primary or unique key column is not `BINARY`, Oracle creates a hidden virtual column for this column. The expression of the virtual column calculates collation keys for character values of the original key column. The primary key or unique constraint is internally created on the virtual column instead of the original column. The virtual column is visible in the data dictionary views of the `*_TAB_COLS` family. For each of these hidden virtual columns, the `COLLATED_COLUMN_ID` of the `*_TAB_COLS` views contains the internal sequence number pointing to the corresponding original key column. The hidden virtual columns count to the 1000-column limit of a table, i.e. 1000 if the `MAX_COLUMNS` initialization parameter is set to `STANDARD`, or 4096 columns if `MAX_COLUMNS` is set to `EXTENDED`.

> **See Also:**
>
> - See *Oracle Database Reference* for more on the `MAX_COLUMNS` initialization parameter.
> - Case-Insensitive Constraints Example
> - *Oracle Database Globalization Support Guide* for more details about collations

Specifying Constraint State

You can specify how and when Oracle should enforce the constraint when you define the constraint.

constraint_state

You can use *`constraint_state`* with both inline and out-of-line specification. Except for the clauses `DEFERRABLE` and `INITIALLY`, that may be specified in any order, you must specify the rest of the component clauses in the order shown, and each clause only once.

DEFERRABLE Clause

The `DEFERRABLE` and `NOT DEFERRABLE` parameters indicate whether or not, in subsequent transactions, constraint checking can be deferred until the end of the transaction using the `SET CONSTRAINT(S)` statement. If you omit this clause, then the default is `NOT DEFERRABLE`.

- Specify `NOT DEFERRABLE` to indicate that in subsequent transactions you cannot use the `SET CONSTRAINT[S]` clause to defer checking of this constraint until the transaction is committed. The checking of a `NOT DEFERRABLE` constraint can never be deferred to the end of the transaction.

  If you declare a new constraint `NOT DEFERRABLE`, then it must be valid at the time the `CREATE TABLE` or `ALTER TABLE` statement is committed or the statement will fail.
- Specify `DEFERRABLE` to indicate that in subsequent transactions you can use the `SET CONSTRAINT[S]` clause to defer checking of this constraint until a `COMMIT` statement is submitted. If the constraint check fails, then the database returns an error and the transaction is not committed. This setting in effect lets you disable the constraint temporarily while making changes to the database that might violate the constraint until all the changes are complete.

  > **Note:**
  >
  > The optimizer does not consider indexes on deferrable constraints as usable.

You cannot alter the deferrability of a constraint. Whether you specify either of these parameters, or make the constraint `NOT DEFERRABLE` implicitly by specifying neither of them, you cannot specify this clause in an `ALTER TABLE` statement. You must drop the constraint and re-create it.

> **See Also:**
>
> - SET CONSTRAINT[S] for information on setting constraint checking for a transaction
> - *Oracle Database Administrator's Guide* and *Oracle Database Concepts* for more information about deferred constraints
> - "DEFERRABLE Constraint Examples"

Restriction on [NOT] DEFERRABLE

You cannot specify either of these parameters for a view constraint.

INITIALLY Clause

The `INITIALLY` clause establishes the default checking behavior for constraints that are `DEFERRABLE`. The `INITIALLY` setting can be overridden by a `SET CONSTRAINT(S)` statement in a subsequent transaction.

- Specify `INITIALLY IMMEDIATE` to indicate that Oracle should check this constraint at the end of each subsequent SQL statement. If you do not specify `INITIALLY` at all, then the default is `INITIALLY IMMEDIATE`.

  If you declare a new constraint `INITIALLY IMMEDIATE`, then it must be valid at the time the `CREATE TABLE` or `ALTER TABLE` statement is committed or the statement will fail.
- Specify `INITIALLY DEFERRED` to indicate that Oracle should check this constraint at the end of subsequent transactions.

This clause is not valid if you have declared the constraint to be `NOT DEFERRABLE`, because a `NOT DEFERRABLE` constraint is automatically `INITIALLY IMMEDIATE` and cannot ever be `INITIALLY DEFERRED`.

RELY Clause

The `RELY` and `NORELY` parameters specify whether a constraint in `NOVALIDATE` mode is to be taken into account for query rewrite. Specify `RELY` to activate a constraint in `NOVALIDATE` mode for query rewrite in an unenforced query rewrite integrity mode. The constraint is in `NOVALIDATE` mode, so Oracle does not enforce it. The default is `NORELY`.

Unenforced constraints are generally useful only with materialized views and query rewrite. Depending on the `QUERY_REWRITE_INTEGRITY` mode, query rewrite can use only constraints that are in `VALIDATE` mode, or that are in `NOVALIDATE` mode with the `RELY` parameter set, to determine join information.

Restriction on the RELY Clause

You cannot set a nondeferrable `NOT NULL` constraint to `RELY`.

> **See Also:**
>
> *Oracle Database Data Warehousing Guide* for more information on materialized views and query rewrite

Using Indexes to Enforce Constraints

When defining the state of a unique or primary key constraint, you can specify an index for Oracle to use to enforce the constraint, or you can instruct Oracle to create the index used to enforce the constraint.

using_index_clause

You can specify the *using_index_clause* only when enabling unique or primary key constraints. You can specify the clauses of the *using_index_clause* in any order, but you can specify each clause only once.

- If you specify *schema.index*, then Oracle attempts to enforce the constraint using the specified index. If Oracle cannot find the index or cannot use the index to enforce the constraint, then Oracle returns an error.
- If you specify the *create_index_statement*, then Oracle attempts to create the index and use it to enforce the constraint. If Oracle cannot create the index or cannot use the index to enforce the constraint, then Oracle returns an error.
- If you neither specify an existing index nor create a new index, then Oracle creates the index. In this case:
  - The index receives the same name as the constraint.
  - If *table* is partitioned, then you can specify a locally or globally partitioned index for the unique or primary key constraint.

Restrictions on the *using_index_clause*

The following restrictions apply to the *using_index_clause*:

- You cannot specify this clause for a view constraint.
- You cannot specify this clause for a NOT NULL, foreign key, or check constraint.
- You cannot specify an index (*schema.index*) or create an index (*create_index_statement*) when enabling the primary key of an index-organized table.
- You cannot specify the *parallel_clause* of *index_attributes*.
- The INDEXTYPE IS ... clause of *index_properties* is not valid in the definition of a constraint.

> **See Also:**
>
> - CREATE INDEX for a description of *index_attributes*, the *global_partitioned_index* and *local_partitioned_index* clauses, and for a description of NOSORT and the *logging_clause* in relation to indexes
> - *physical_attributes_clause* and PCTFREE parameters and *storage_clause*
> - "Explicit Index Control Example"

ENABLE Clause

Specify ENABLE if you want the constraint to be applied to the data in the table.

If you enable a unique or primary key constraint, and if no index exists on the key, then Oracle Database creates a unique index. Unless you specify `KEEP INDEX` when subsequently disabling the constraint, this index is dropped and the database rebuilds the index every time the constraint is reenabled.

You can also avoid rebuilding the index and eliminate redundant indexes by creating new primary key and unique constraints initially disabled. Then create (or use existing) nonunique indexes to enforce the constraint. Oracle does not drop a nonunique index when the constraint is disabled, so subsequent `ENABLE` operations are facilitated.

- `ENABLE VALIDATE` specifies that all old and new data also complies with the constraint. An enabled validated constraint guarantees that all data is and will continue to be valid.

  If any row in the table violates the integrity constraint, then the constraint remains disabled and Oracle returns an error. If all rows comply with the constraint, then Oracle enables the constraint. Subsequently, if new data violates the constraint, then Oracle does not execute the statement and returns an error indicating the integrity constraint violation.

  If you place a primary key constraint in `ENABLE VALIDATE` mode, then the validation process will verify that the primary key columns contain no nulls. To avoid this overhead, mark each column in the primary key `NOT NULL` before entering data into the column and before enabling the primary key constraint of the table.

- `ENABLE NOVALIDATE` ensures that all new DML operations on the constrained data comply with the constraint. This clause does not ensure that existing data in the table complies with the constraint.

If you specify neither `VALIDATE` nor `NOVALIDATE`, then the default is `VALIDATE`.

If you change the state of any single constraint from `ENABLE NOVALIDATE` to `ENABLE VALIDATE`, then the operation can be performed in parallel, and does not block reads, writes, or other DDL operations.

Restriction on the ENABLE Clause

You cannot enable a foreign key that references a disabled unique or primary key.

DISABLE Clause

Specify `DISABLE` to disable the integrity constraint. Disabled integrity constraints appear in the data dictionary along with enabled constraints. If you do not specify this clause when creating a constraint, then Oracle automatically enables the constraint.

- `DISABLE VALIDATE` disables the constraint and drops the index on the constraint, but keeps the constraint valid. This feature is most useful in data warehousing situations, because it lets you load large amounts of data while also saving space by not having an index. This setting lets you load data from a nonpartitioned table into a partitioned table using the *`exchange_partition_subpart`* clause of the `ALTER TABLE` statement or using SQL*Loader. All other modifications to the table (inserts, updates, and deletes) by other SQL statements are disallowed.

> **See Also:**
>
> *Oracle Database Data Warehousing Guide* for more information on using this setting

- `DISABLE NOVALIDATE` signifies that Oracle makes no effort to maintain the constraint (because it is disabled) and cannot guarantee that the constraint is true (because it is not being validated).

  You cannot drop a table whose primary key is being referenced by a foreign key even if the foreign key constraint is in `DISABLE NOVALIDATE` state. Further, the optimizer can use constraints in `DISABLE NOVALIDATE` state.

> **See Also:**
>
> *Oracle Database SQL Tuning Guide* for information on when to use this setting

If you specify neither `VALIDATE` nor `NOVALIDATE`, then the default is `NOVALIDATE`.

If you disable a unique or primary key constraint that is using a unique index, then Oracle drops the unique index. Refer to the `CREATE TABLE` *enable_disable_clause* for additional notes and restrictions.

VALIDATE | NOVALIDATE

The behavior of `VALIDATE` and `NOVALIDATE` depends on whether the constraint is enabled or disabled, either explicitly or by default. Therefore, the `VALIDATE` and `NOVALIDATE` keywords are described in the context of "ENABLE Clause" and "DISABLE Clause".

Note on Foreign Key Constraints in NOVALIDATE Mode

When a foreign key constraint is in `NOVALIDATE` mode, if existing data in the table does not comply with the constraint and the `QUERY_REWRITE_INTEGRITY` parameter is not set to `ENFORCED`, then the optimizer may use join elimination during queries on the table. In this case, a query may return table rows with noncompliant foreign key values even if the query contains a join condition that should filter out those rows.

Handling Constraint Exceptions

When defining the state of a constraint, you can specify a table into which Oracle places the rowids of all rows violating the constraint.

exceptions_clause

Use the *exceptions_clause* syntax to define exception handling. If you omit *schema*, then Oracle assumes the exceptions table is in your own schema. If you omit this clause altogether, then Oracle assumes that the table is named `EXCEPTIONS`. The `EXCEPTIONS` table or the table you specify must exist on your local database.

You can create the `EXCEPTIONS` table using one of these scripts:

- `UTLEXCPT.SQL` uses physical rowids. Therefore it can accommodate rows from conventional tables but not from index-organized tables. (See the Note that follows.)
- `UTLEXPT1.SQL` uses universal rowids, so it can accommodate rows from both conventional and index-organized tables.

If you create your own exceptions table, then it must follow the format prescribed by one of these two scripts.

If you are collecting exceptions from index-organized tables based on primary keys (rather than universal rowids), then you must create a separate exceptions table for each index-

organized table to accommodate its primary-key storage. You create multiple exceptions tables with different names by modifying and resubmitting the script.

Restrictions on the *exceptions_clause*

The following restrictions apply to the *exceptions_clause*:

- You cannot specify this clause for a view constraint.
- You cannot specify this clause in a `CREATE TABLE` statement, because no rowids exist until *after* the successful completion of the statement.

> **See Also:**
>
> - The `DBMS_IOT` package in *Oracle Database PL/SQL Packages and Types Reference* for information on the SQL scripts
> - *Oracle Database Performance Tuning Guide* for information on eliminating migrated and chained rows

View Constraints

Oracle does not enforce view constraints. However, operations on views are subject to the integrity constraints defined on the underlying base tables. This means that you can enforce constraints on views through constraints on base tables.

Notes on View Constraints

View constraints are a subset of table constraints and are subject to the following restrictions:

- You can specify only unique, primary key, and foreign key constraints on views. However, you can define the view using the `WITH CHECK OPTION` clause, which is equivalent to specifying a check constraint for the view.
- View constraints are supported only in `DISABLE NOVALIDATE` mode. You cannot specify any other mode. You must specify the keyword `DISABLE` when you declare the view constraint. You need not specify `NOVALIDATE` explicitly, as it is the default.
- The `RELY` and `NORELY` parameters are optional. View constraints, because they are unenforced, are usually specified with the `RELY` parameter to make them more useful. The `RELY` or `NORELY` keyword must precede the `DISABLE` keyword.
- Because view constraints are not enforced directly, you cannot specify `INITIALLY DEFERRED` or `DEFERRABLE`.
- You cannot specify the *using_index_clause*, the *exceptions_clause* clause, or the `ON DELETE` clause of the *references_clause*.
- You cannot define view constraints on attributes of an object column.

External Table Constraints

Starting with Oracle Database 12*c* Release 2 (12.2), you can specify `NOT NULL`, unique, primary key, and foreign key constraints on external tables.

`NOT NULL` constraints on external tables are enforced and prohibit columns from containing nulls.

Unique, primary key, and foreign key constraints are supported on external tables only in `RELY DISABLE` mode. You must specify the keywords `RELY` and `DISABLE` when you create these constraints. These constraints are declarative and are not enforced. They can increase query performance and reduce resource consumption because more optimizer transformations can be taken into account. In order for the optimizer to utilize these `RELY DISABLE` constraints, the `QUERY_REWRITE_INTEGRITY` initialization parameter must be set to either `trusted` or `stale_tolerated`.

Examples

Unique Key Example

The following statement is a variation of the statement that created the sample table `sh.promotions`. It defines inline and implicitly enables a unique key on the `promo_id` column (other constraints are not shown):

```
CREATE TABLE promotions_var1
    ( promo_id         NUMBER(6)
                       CONSTRAINT promo_id_u  UNIQUE
    , promo_name       VARCHAR2(20)
    , promo_category   VARCHAR2(15)
    , promo_cost       NUMBER(10,2)
    , promo_begin_date DATE
    , promo_end_date   DATE
    ) ;
```

The constraint `promo_id_u` identifies the `promo_id` column as a unique key. This constraint ensures that no two promotions in the table have the same ID. However, the constraint does allow promotions without identifiers.

Alternatively, you can define and enable this constraint out of line:

```
CREATE TABLE promotions_var2
    ( promo_id         NUMBER(6)
    , promo_name       VARCHAR2(20)
    , promo_category   VARCHAR2(15)
    , promo_cost       NUMBER(10,2)
    , promo_begin_date DATE
    , promo_end_date   DATE
    , CONSTRAINT promo_id_u UNIQUE (promo_id)
   USING INDEX PCTFREE 20
      TABLESPACE stocks
      STORAGE (INITIAL 8M) );
```

The preceding statement also contains the *`using_index_clause`*, which specifies storage characteristics for the index that Oracle creates to enable the constraint.

Composite Unique Key Example

The following statement defines and enables a composite unique key on the combination of the `warehouse_id` and `warehouse_name` columns of the `oe.warehouses` table:

```
ALTER TABLE warehouses
   ADD CONSTRAINT wh_unq UNIQUE (warehouse_id, warehouse_name)
   USING INDEX PCTFREE 5
   EXCEPTIONS INTO wrong_id;
```

The `wh_unq` constraint ensures that the same combination of `warehouse_id` and `warehouse_name` values does not appear in the table more than once.

The `ADD CONSTRAINT` clause also specifies other properties of the constraint:

- The `USING INDEX` clause specifies storage characteristics for the index Oracle creates to enable the constraint.
- The `EXCEPTIONS INTO` clause causes Oracle to write to the `wrong_id` table information about any rows currently in the `warehouses` table that violate the constraint. If the `wrong_id` exceptions table does not already exist, then this statement will fail.

Primary Key Example

The following statement is a variation of the statement that created the sample table `hr.locations`. It creates the `locations_demo` table and defines and enables a primary key on the `location_id` column (other constraints from the `hr.locations` table are omitted):

```
CREATE TABLE locations_demo
    ( location_id    NUMBER(4) CONSTRAINT loc_id_pk PRIMARY KEY
    , street_address VARCHAR2(40)
    , postal_code    VARCHAR2(12)
    , city           VARCHAR2(30)
    , state_province VARCHAR2(25)
    , country_id     CHAR(2)
    ) ;
```

The `loc_id_pk` constraint, specified inline, identifies the `location_id` column as the primary key of the `locations_demo` table. This constraint ensures that no two locations in the table have the same location number and that no location identifier is `NULL`.

Alternatively, you can define and enable this constraint out of line:

```
CREATE TABLE locations_demo
    ( location_id    NUMBER(4)
    , street_address VARCHAR2(40)
    , postal_code    VARCHAR2(12)
    , city           VARCHAR2(30)
    , state_province VARCHAR2(25)
    , country_id     CHAR(2)
    , CONSTRAINT loc_id_pk PRIMARY KEY (location_id));
```

NOT NULL Example

The following statement alters the `locations_demo` table (created in "Primary Key Example") to define and enable a `NOT NULL` constraint on the `country_id` column:

```
ALTER TABLE locations_demo
   MODIFY (country_id CONSTRAINT country_nn NOT NULL);
```

The constraint `country_nn` ensures that no location in the table has a null `country_id`.

Composite Primary Key Example

The following statement defines a composite primary key on the combination of the `prod_id` and `cust_id` columns of the sample table `sh.sales`:

```
ALTER TABLE sales
    ADD CONSTRAINT sales_pk PRIMARY KEY (prod_id, cust_id) DISABLE;
```

This constraint identifies the combination of the `prod_id` and `cust_id` columns as the primary key of the `sales` table. The constraint ensures that no two rows in the table have the same combination of values for the `prod_id` column and `cust_id` columns.

The constraint clause (`PRIMARY KEY`) also specifies the following properties of the constraint:

- The constraint definition does not include a constraint name, so Oracle generates a name for the constraint.
- The `DISABLE` clause causes Oracle to define the constraint but not enable it.

Foreign Key Constraint Example

The following statement creates the `dept_20` table and defines and enables a foreign key on the `department_id` column that references the primary key on the `department_id` column of the `departments` table:

```
CREATE TABLE dept_20
   (employee_id     NUMBER(4),
    last_name       VARCHAR2(10),
    job_id          VARCHAR2(9),
    manager_id      NUMBER(4),
    hire_date       DATE,
    salary          NUMBER(7,2),
    commission_pct  NUMBER(7,2),
    department_id   CONSTRAINT fk_deptno
                    REFERENCES departments(department_id) );
```

The constraint `fk_deptno` ensures that all departments given for employees in the `dept_20` table are present in the `departments` table. However, employees can have null department numbers, meaning they are not assigned to any department. To ensure that all employees are assigned to a department, you could create a `NOT NULL` constraint on the `department_id` column in the `dept_20` table in addition to the `REFERENCES` constraint.

Before you define and enable this constraint, you must define and enable a constraint that designates the `department_id` column of the `departments` table as a primary or unique key.

The foreign key constraint definition does not use the `FOREIGN KEY` clause, because the constraint is defined inline. The data type of the *`department_id`* column is not needed, because Oracle automatically assigns to this column the data type of the referenced key.

The constraint definition identifies both the parent table and the columns of the referenced key. Because the referenced key is the primary key of the parent table, the referenced key column names are optional.

Alternatively, you can define this foreign key constraint out of line:

```
CREATE TABLE dept_20
   (employee_id     NUMBER(4),
    last_name       VARCHAR2(10),
    job_id          VARCHAR2(9),
    manager_id      NUMBER(4),
    hire_date       DATE,
    salary          NUMBER(7,2),
    commission_pct  NUMBER(7,2),
    department_id,
   CONSTRAINT fk_deptno
      FOREIGN  KEY (department_id)
      REFERENCES  departments(department_id) );
```

The foreign key definitions in both variations of this statement omit the `ON DELETE` clause, causing Oracle to prevent the deletion of a department if any employee works in that department.

ON DELETE Example

This statement creates the `dept_20` table, defines and enables two referential integrity constraints, and uses the `ON DELETE` clause:

```
CREATE TABLE dept_20
   (employee_id     NUMBER(4) PRIMARY KEY,
    last_name       VARCHAR2(10),
    job_id          VARCHAR2(9),
    manager_id      NUMBER(4) CONSTRAINT fk_mgr
                    REFERENCES employees ON DELETE SET NULL,
    hire_date       DATE,
    salary          NUMBER(7,2),
    commission_pct  NUMBER(7,2),
    department_id   NUMBER(2)   CONSTRAINT fk_deptno
                    REFERENCES departments(department_id)
                    ON DELETE CASCADE );
```

Because of the first `ON DELETE` clause, if manager number 2332 is deleted from the `employees` table, then Oracle sets to null the value of `manager_id` for all employees in the `dept_20` table who previously had manager 2332.

Because of the second `ON DELETE` clause, Oracle cascades any deletion of a `department_id` value in the `departments` table to the `department_id` values of its dependent rows of the `dept_20` table. For example, if Department 20 is deleted from the `departments` table, then Oracle deletes all of the employees in Department 20 from the `dept_20` table.

Composite Foreign Key Constraint Example

The following statement defines and enables a foreign key on the combination of the `employee_id` and `hire_date` columns of the `dept_20` table:

```
ALTER TABLE dept_20
   ADD CONSTRAINT fk_empid_hiredate
   FOREIGN KEY (employee_id, hire_date)
   REFERENCES hr.job_history(employee_id, start_date)
   EXCEPTIONS INTO wrong_emp;
```

The constraint `fk_empid_hiredate` ensures that all the employees in the `dept_20` table have `employee_id` and `hire_date` combinations that exist in the `employees` table. Before you define and enable this constraint, you must define and enable a constraint that designates the combination of the `employee_id` and `hire_date` columns of the `employees` table as a primary or unique key.

The `EXCEPTIONS INTO` clause causes Oracle to write information to the `wrong_emp` table about any rows in the `dept_20` table that violate the constraint. If the `wrong_emp` exceptions table does not already exist, then this statement will fail.

Check Constraint Examples

The following statement creates a `divisions` table and defines a `check` constraint in each column of the table:

```
CREATE TABLE divisions
   (div_no    NUMBER  CONSTRAINT check_divno
              CHECK (div_no BETWEEN 10 AND 99)
              DISABLE,
    div_name  VARCHAR2(9)  CONSTRAINT check_divname
              CHECK (div_name = UPPER(div_name))
              DISABLE,
    office    VARCHAR2(10)  CONSTRAINT check_office
              CHECK (office IN ('DALLAS','BOSTON',
              'PARIS','TOKYO'))
              DISABLE);
```

Each constraint restricts the values of the column in which it is defined:

- `check_divno` ensures that no division numbers are less than 10 or greater than 99.
- `check_divname` ensures that all division names are in uppercase.
- `check_office` restricts office locations to Dallas, Boston, Paris, or Tokyo.

Because each `CONSTRAINT` clause contains the `DISABLE` clause, Oracle only defines the constraints and does not enable them.

The following statement creates the `dept_20` table, defining out of line and implicitly enabling a check constraint:

```
CREATE TABLE dept_20
   (employee_id     NUMBER(4) PRIMARY KEY,
    last_name       VARCHAR2(10),
    job_id          VARCHAR2(9),
    manager_id      NUMBER(4),
    salary          NUMBER(7,2),
    commission_pct  NUMBER(7,2),
    department_id   NUMBER(2),
    CONSTRAINT check_sal CHECK (salary * commission_pct <= 5000));
```

This constraint uses an inequality condition to limit an employee's total commission, the product of `salary` and `commission_pct`, to $5000:

- If an employee has non-null values for both salary and commission, then the product of these values must not exceed $5000 to satisfy the constraint.
- If an employee has a null salary or commission, then the result of the condition is unknown and the employee automatically satisfies the constraint.

Because the constraint clause in this example does not supply a constraint name, Oracle generates a name for the constraint.

The following statement defines and enables a primary key constraint, two foreign key constraints, a `NOT NULL` constraint, and two check constraints:

```
CREATE TABLE order_detail
  (CONSTRAINT pk_od PRIMARY KEY (order_id, part_no),
   order_id    NUMBER
      CONSTRAINT fk_oid
         REFERENCES oe.orders(order_id),
   part_no     NUMBER
      CONSTRAINT fk_pno
         REFERENCES oe.product_information(product_id),
   quantity    NUMBER
      CONSTRAINT nn_qty NOT NULL
      CONSTRAINT check_qty CHECK (quantity > 0),
   cost        NUMBER
      CONSTRAINT check_cost CHECK (cost > 0) );
```

The constraints enable the following rules on table data:

- `pk_od` identifies the combination of the `order_id` and `part_no` columns as the primary key of the table. To satisfy this constraint, no two rows in the table can contain the same combination of values in the `order_id` and the `part_no` columns, and no row in the table can have a null in either the `order_id` or the `part_no` column.
- `fk_oid` identifies the `order_id` column as a foreign key that references the `order_id` column in the `orders` table in the sample schema `oe`. All new values added to the column `order_detail.order_id` must already appear in the column `oe.orders.order_id`.

- `fk_pno` identifies the `product_id` column as a foreign key that references the `product_id` column in the `product_information` table owned by `oe`. All new values added to the column `order_detail.product_id` must already appear in the column `oe.product_information.product_id`.
- `nn_qty` forbids nulls in the `quantity` column.
- `check_qty` ensures that values in the `quantity` column are always greater than zero.
- `check_cost` ensures the values in the cost column are always greater than zero.

This example also illustrates the following points about constraint clauses and column definitions:

- Out-of-line constraint definition can appear before or after the column definitions. In this example, the out-of-line definition of the `pk_od` constraint precedes the column definitions.
- A column definition can contain multiple inline constraint definitions. In this example, the definition of the `quantity` column contains the definitions of both the `nn_qty` and `check_qty` constraints.
- A table can have multiple `CHECK` constraints. Multiple `CHECK` constraints, each with a simple condition enforcing a single business rule, are preferable to a single `CHECK` constraint with a complicated condition enforcing multiple business rules. When a constraint is violated, Oracle returns an error identifying the constraint. Such an error more precisely identifies the violated business rule if the identified constraint enables a single business rule.

Create a Table with PRECHECK: Example

The following example creates a table `Product` with `PRECHECK` constraints on columns `Price`, `Color`, `Description`, constant `NUMBER`, and constraint `TC1` :

```
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)
);
Table PRODUCT created.
```

Add Precheck State to a New Constraint using ALTER TABLE:

```
ALTER TABLE Product MODIFY (Name VARCHAR2(50) CHECK
  (regexp_like(Name, '^Product')) PRECHECK);
```

Add Precheck to an Existing Costraint State :

```
ALTER TABLE Product MODIFY CONSTRAINT TC2 PRECHECK;
```

Remove an Existing Precheck State:

```
ALTER TABLE Product MODIFY CONSTRAINT TC1 NOPRECHECK;
```

Check PRECHECK State in USER_CONSTRAINTS: Example

Given the following table `Product`:

```
SQL> CREATE TABLE Product(
       Id NUMBER NOT NULL PRIMARY KEY,
       Name VARCHAR2(50),
       Category VARCHAR2(10) NOT NULL,
       Price NUMBER CHECK (mod(price,4) = 0 and 10 <> price),
       Color NUMBER CHECK (Color >= 10 and Color <=50) PRECHECK,
       Description VARCHAR2(50) CHECK (Length(Description) <= 40),
       Created_At DATE,
       Updated_At DATE,
       CONSTRAINT TC1 CHECK (Color > 0 AND Price > 10),
       CONSTRAINT TC2 CHECK (CATEGORY IN ('Home', 'Apparel')) NOPRECHECK,
       CONSTRAINT TC3 CHECK (Created_At > Updated_At)
     );

Table PRODUCT created.
```

You can check the `PRECHECK` state in `USER_CONSTRAINTS` as follows:

```
SELECT CONSTRAINT_NAME, SEARCH_CONDITION, PRECHECK
  FROM USER_CONSTRAINTS
  WHERE table_name='PRODUCT' and constraint_type='C';
```

The result is:

```
CONSTRAINT_NAME                    SEARCH_CONDITION                    PRECHECK
_____________________              ____________________________________
_____________
SYS_C008676 "ID"                   IS NOT NULL
SYS_C008677 "CATEGORY"             IS NOT NULL
SYS_C008678  mod(price,4)          = 0 and 10 <> price                  PRECHECK
SYS_C008679 Color                  >= 10 and Color <=50                 PRECHECK
SYS_C008680 Length(Description)    <= 40                                PRECHECK
TC1                                Color > 0 AND Price > 10             PRECHECK
TC2                                CATEGORY IN ('Home', 'Apparel')      NOPRECHECK
TC3                                Created_At > Updated_At              NOPRECHECK

8 rows selected.
```

Several constraints are automatically set to a value in both inline and out-of-line constraints.

Case-Insensitive Constraints Example

The following statements create two tables in a parent-child relationship. The parent table is a product description table and the child table is a product component description table. Unique constraints are defined to assure that product and description values are unambiguous. For illustrative purposes, the product and component ID are case-insensitive character values. (In real-world applications, primary key IDs are usually numeric or case-normalized.)

```
CREATE TABLE products
    ( product_id VARCHAR2(20) COLLATE BINARY_CI
          CONSTRAINT product_pk PRIMARY KEY
    , description VARCHAR2(1000) COLLATE BINARY_CI
          CONSTRAINT product_description_unq UNIQUE
    );

CREATE TABLE product_components
    ( component_id VARCHAR2(40) COLLATE BINARY_CI
          CONSTRAINT product_component_pk PRIMARY KEY
```

```
    , product_id CONSTRAINT product_component_fk REFERENCES products(product_id)
    , description VARCHAR2(1000) COLLATE BINARY_CI
          CONSTRAINT product_component_descr_unq UNIQUE
    );
```

Note that if you do not specify the data type or the collation for a foreign key column, then they are inherited from the parent key column.

The following statements add a product and its components into the tables:

```
INSERT INTO products(product_id, description)
            VALUES('BICY0001', 'Men''s bicycle, fr 21", wh 24", gear 3x7');
INSERT INTO product_components(component_id, product_id, description)
            VALUES('BICY0001_FRAME01', 'BICY0001', 'Aluminium frame 21"');
INSERT INTO product_components(component_id, product_id, description)
            VALUES('BICY0001_WHEEL01', 'bicy0001', 'Wheels 24"');
INSERT INTO product_components(component_id, product_id, description)
            VALUES('BICY0001_GEAR01', 'Bicy0001', 'Front derailleur 3 chainrings');
INSERT INTO product_components(component_id, product_id, description)
            VALUES('BICY0001_gear02', 'BiCy0001', 'Rear derailleur 7 chainrings');
```

Note the different case of the product ID in different component rows. Because the primary key on the product ID is declared as case-insensitive, all possible letter case combinations of the same ID are considered equal.

The following statement demonstrates that it is not possible to enter another product with the same description differing only by case. It fails with the error `ORA-00001: unique constraint (`*schema*`.PRODUCT_DESCRIPTION_UNQ) violated`.

```
INSERT INTO products(product_id, description)
            VALUES('BICY0002', 'MEN''S BICYCLE, fr 21", wh 24", gear 3x7');
```

Similarly, the following statement demonstrates that the primary key contraint of the product table is case-insensitive and does not allow values differing only by case. It fails with the error `ORA-00001: unique constraint (`*schema*`.PRODUCT_PK) violated`.

```
INSERT INTO products(component_id, product_id, description)
            VALUES('bicy0001', 'Women''s bicycle, fr 21", wh 24", gear 2x6');
```

The following statement demonstrates that it is not possible to enter another component with the same description differing only by case. It fails with the error `ORA-00001: unique constraint (`*schema*`.PRODUCT_COMPONENT_DESCR_UNQ) violated`.

```
INSERT INTO product_components(component_id, product_id, description)
            VALUES('BICY0001_gear03', 'BiCy0001', 'REAR DERAILLEUR 7 CHAINRINGS');
```

Attribute-Level Constraints Example

The following example guarantees that a value exists for both the `first_name` and `last_name` attributes of the `name` column in the `students` table:

```
CREATE TYPE person_name AS OBJECT
   (first_name VARCHAR2(30), last_name VARCHAR2(30));
/

CREATE TABLE students (name person_name, age INTEGER,
   CHECK (name.first_name IS NOT NULL AND
          name.last_name IS NOT NULL));
```

REF Constraint Examples

The following example creates a duplicate of the sample schema object type `cust_address_typ`, and then creates a table containing a `REF` column with a `SCOPE` constraint:

```
CREATE TYPE cust_address_typ_new AS OBJECT
    ( street_address     VARCHAR2(40)
    , postal_code        VARCHAR2(10)
    , city               VARCHAR2(30)
    , state_province     VARCHAR2(10)
    , country_id         CHAR(2)
    );
/
CREATE TABLE address_table OF cust_address_typ_new;

CREATE TABLE customer_addresses (
   add_id NUMBER,
   address REF cust_address_typ_new
   SCOPE IS address_table);
```

The following example creates the same table but with a referential integrity constraint on the `REF` column that references the object identifier column of the parent table:

```
CREATE TABLE customer_addresses (
   add_id NUMBER,
   address REF cust_address_typ REFERENCES address_table);
```

The following example uses the type `department_typ` and the table `departments_obj_t`, created in "Creating Object Tables: Examples". A table with a scoped `REF` is then created.

```
CREATE TABLE employees_obj
   ( e_name   VARCHAR2(100),
     e_number NUMBER,
     e_dept   REF department_typ SCOPE IS departments_obj_t );
```

The following statement creates a table with a `REF` column which has a referential integrity constraint defined on it:

```
CREATE TABLE employees_obj
   ( e_name   VARCHAR2(100),
     e_number NUMBER,
     e_dept   REF department_typ REFERENCES departments_obj_t);
```

Explicit Index Control Example

The following statement shows another way to create a unique (or primary key) constraint that gives you explicit control over the index (or indexes) Oracle uses to enforce the constraint:

```
CREATE TABLE promotions_var3
    ( promo_id         NUMBER(6)
    , promo_name       VARCHAR2(20)
    , promo_category   VARCHAR2(15)
    , promo_cost       NUMBER(10,2)
    , promo_begin_date DATE
    , promo_end_date   DATE
    , CONSTRAINT promo_id_u UNIQUE (promo_id, promo_cost)
         USING INDEX (CREATE UNIQUE INDEX promo_ix1
            ON promotions_var3 (promo_id, promo_cost))
    , CONSTRAINT promo_id_u2 UNIQUE (promo_cost, promo_id)
         USING INDEX promo_ix1);
```

This example also shows that you can create an index for one constraint and use that index to create and enable another constraint in the same statement.

DEFERRABLE Constraint Examples

The following statement creates table `games` with a `NOT DEFERRABLE INITIALLY IMMEDIATE` constraint check (by default) on the `scores` column:

```
CREATE TABLE games (scores NUMBER CHECK (scores >= 0));
```

To define a unique constraint on a column as `INITIALLY DEFERRED DEFERRABLE`, issue the following statement:

```
CREATE TABLE games
  (scores NUMBER, CONSTRAINT unq_num UNIQUE (scores)
   INITIALLY DEFERRED DEFERRABLE);
```

deallocate_unused_clause

Purpose

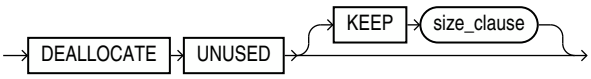
Use the *deallocate_unused_clause* to explicitly deallocate unused space at the end of a database object segment and make the space available for other segments in the tablespace.

You can deallocate unused space using the following statements:

- `ALTER CLUSTER` (see ALTER CLUSTER)
- `ALTER INDEX`: to deallocate unused space from the index, an index partition, or an index subpartition (see ALTER INDEX)
- `ALTER MATERIALIZED VIEW`: to deallocate unused space from the overflow segment of an index-organized materialized view (see ALTER MATERIALIZED VIEW)
- `ALTER TABLE`: to deallocate unused space from the table, a table partition, a table subpartition, the mapping table of an index-organized table, the overflow segment of an index-organized table, or a LOB storage segment (see ALTER TABLE)

Syntax

***deallocate_unused_clause*::=**



(*size_clause*::=)

Semantics

This section describes the semantics of the *deallocate_unused_clause*. For additional information, refer to the SQL statement in which you set or reset this clause for a particular database object.

You cannot specify both the *deallocate_unused_clause* and the *allocate_extent_clause* in the same statement.

Oracle Database frees only unused space above the high water mark (the point beyond which database blocks have not yet been formatted to receive data). Oracle deallocates unused space beginning from the end of the object and moving toward the beginning of the object to the high water mark.

If an extent is completely contained in the deallocation, then the whole extent is freed for reuse. If an extent is partially contained in the deallocation, then the used part up to the high water mark becomes the extent, and the remaining unused space is freed for reuse.

Oracle credits the amount of the released space to the user quota for the tablespace in which the deallocation occurs.

The exact amount of space freed depends on the values of the `INITIAL`, `MINEXTENTS`, and `NEXT` storage parameters. Refer to the *storage_clause* for a description of these parameters.

KEEP *integer*

Specify the number of bytes above the high water mark that the segment of the database object is to have after deallocation.

- If you omit `KEEP` and the high water mark is above the size of `INITIAL` and `MINEXTENTS`, then all unused space above the high water mark is freed. When the high water mark is less than the size of `INITIAL` or `MINEXTENTS`, then all unused space above `MINEXTENTS` is freed.
- If you specify `KEEP`, then the specified amount of space is kept and the remaining space is freed. When the remaining number of extents is less than `MINEXTENTS`, then Oracle adjusts `MINEXTENTS` to the new number of extents. If the initial extent becomes smaller than `INITIAL`, then Oracle adjusts `INITIAL` to the new size.
- In either case, Oracle sets the value of the `NEXT` storage parameter to the size of the last extent that was deallocated.

file_specification

Purpose

Use one of the *file_specification* forms to specify a file as a data file or temp file, or to specify a group of one or more files as a redo log file group. If you are storing your files in Oracle Automatic Storage Management (Oracle ASM) disk groups, then you can further specify the file as a disk group file.

A *file_specification* can appear in the following statements:

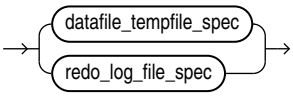
- `CREATE CONTROLFILE` (see CREATE CONTROLFILE)
- `CREATE DATABASE` (see CREATE DATABASE)
- `ALTER DATABASE` (see ALTER DATABASE)
- `CREATE TABLESPACE` (see CREATE TABLESPACE)
- `ALTER TABLESPACE` (see ALTER TABLESPACE)
- `ALTER DISKGROUP` (see ALTER DISKGROUP)

Prerequisites

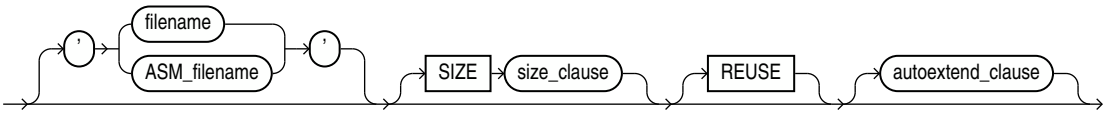
You must have the privileges necessary to issue the statement in which the file specification appears.

Syntax

***file_specification*::=**

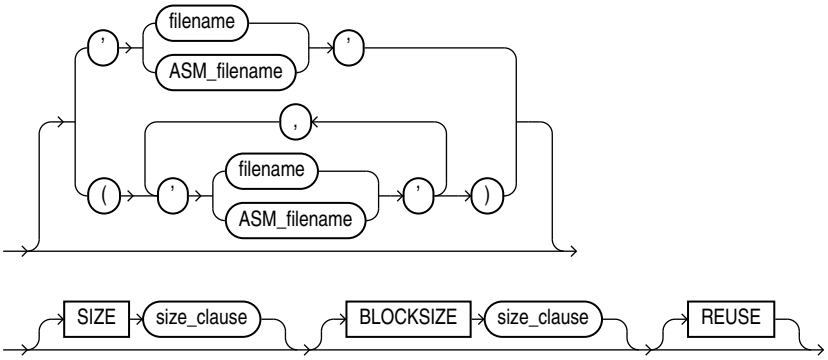


***datafile_tempfile_spec*::=**



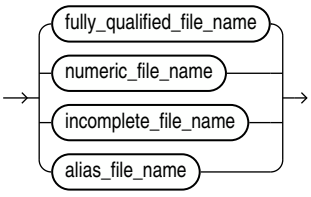
(*size_clause*::=)

***redo_log_file_spec*::=**

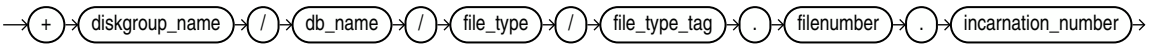


(*size_clause*::=)

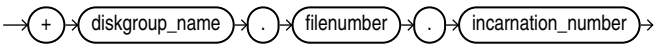
***ASM_filename*::=**



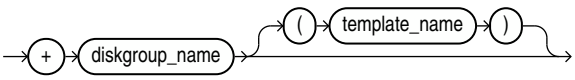
***fully_qualified_file_name*::=**



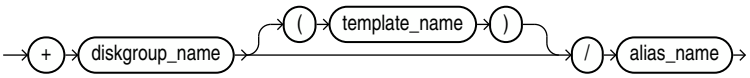
***numeric_file_name*::=**



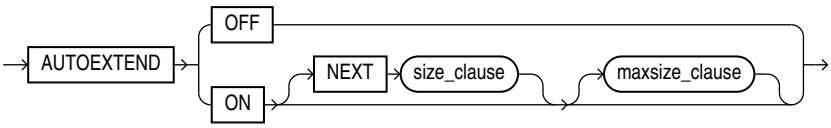
***incomplete_file_name*::=**



***alias_file_name*::=**

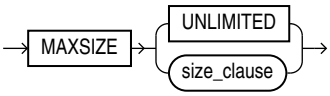


***autoextend_clause*::=**



(*size_clause*::=)

***maxsize_clause*::=**



(*size_clause*::=)

Semantics

This section describes the semantics of `file_specification`. For additional information, refer to the SQL statement in which you specify a data file, temp file, redo log file, or Oracle ASM disk group or disk group file.

datafile_tempfile_spec

Use this clause to specify the attributes of data files and temp files if your database storage is in a file system or in Oracle ASM disk groups.

redo_log_file_spec

Use this clause to specify the attributes of redo log files if your database storage is in a file system or in Oracle ASM disk groups.

filename

Use *filename* for files stored in a file system. The *filename* can specify either a new file or an existing file. For a *new* file:

- If you are *not* using Oracle Managed Files, then you must specify both *filename* and the SIZE clause or the statement fails. When you specify a filename without a size, Oracle attempts to reuse an existing file and returns an error if the file does not exist.
- If you *are* using Oracle Managed Files, then *filename* is optional, as are the remaining clauses of the specification. In this case, Oracle Database creates a unique name for the file and saves it in the directory specified by one of the following initialization parameters:
  - The DB_RECOVERY_FILE_DEST (for logfiles and control files)
  - The DB_CREATE_FILE_DEST initialization parameter (for any type of file)
  - The DB_CREATE_ONLINE_LOG_DEST_*n* initialization parameter, which takes precedence over DB_CREATE_FILE_DEST and DB_RECOVERY_FILE_DEST for log files.

For an *existing* file, specify the name of either a data file, temp file, or a redo log file member. The *filename* can contain only single-byte characters from 7-bit ASCII or EBCDIC character sets. Multibyte characters are not valid.

The filename can include a path prefix. If you do not specify such a path prefix, then the database adds the path prefix for the default storage location, which is platform dependent.

A redo log file group can have one or more members (copies). Each *filename* must be fully specified according to the conventions for your operating system.

The way the database interprets *filename* also depends on whether you specify it with the SIZE and REUSE clauses.

- If you specify *filename* only, or with the REUSE clause but without the SIZE clause, then the file must already exist.
- If you specify *filename* with SIZE but without REUSE, then the file must be a new file.
- If you specify *filename* with both SIZE and REUSE, then the file can be either new or existing. If the file exists, then it is reused with the new size. If it does not exist, then the database ignores the REUSE keyword and creates a new file of the specified size.

> **See Also:**
>
> *Oracle Automatic Storage Management Administrator's Guide* for more information on Oracle Managed Files, "Specifying a Data File: Example", and "Specifying a Log File: Example"

ASM_filename

Use a form of ASM_filename for files stored in Oracle ASM disk groups. You can create or refer to data files, temp files, and redo log files with this syntax.

All forms of ASM_filename begin with the plus sign (+) followed by the name of the disk group. You can determine the names of all Oracle ASM disk groups by querying the V$ASM_DISKGROUP view.

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

fully_qualified_file_name

When you create a file in an Oracle ASM disk group, the file receives a system-generated fully qualified Oracle ASM filename. You can use this form only when referring to an existing Oracle ASM file. Therefore, if you are using this form during file creation, you must also specify `REUSE`.

- *db_name* is the value of the `DB_UNIQUE_NAME` initialization parameter. This name is equivalent to the name of the database on which the file resides, but the parameter distinguishes between primary and standby databases, if both exist.
- *file_type* and *file_type_tag* indicate the type of database file. Table 8-1 lists all of the file types and their corresponding Oracle ASM tags.
- *filenumber* and *incarnation_number* are system-generated identifiers to guarantee uniqueness.

You can determine the fully qualified names of Oracle ASM files by querying the dynamic performance view appropriate for the file type (for example `V$DATAFILE` for data files, `V$CONTROLFILE` for control files, and so on). You can also obtain the *filenumber* and *incarnation_number* portions of the fully qualified names by querying the `V$ASM_FILE` view.

Table 8-1 Oracle File Types and Oracle ASM File Type Tags

Oracle ASM *file_type*	Description	Oracle ASM *file_type_tag*	Comments
CONTROLFILE	Control files and backup control files	Current Backup	—
DATAFILE	Data files and data file copies	*tsname*	Tablespace into which the file is added
ONLINELOG	Online logs	group_*group#*	—
ARCHIVELOG	Archive logs	thread_*thread#*_seq_*sequence#*	—
TEMPFILE	Temp files	*tsname*	Tablespace into which the file is added
BACKUPSET	Data file and archive log backup pieces; data file incremental backup pieces	*hasspfile_timestamp*	*hasspfile* can take one of two values: *s* indicates that the backup set includes the spfile; *n* indicates that the backup set does not include the spfile.
PARAMETERFILE	Persistent parameter files	spfile	—
DATAGUARDCONFIG	Data Guard configuration file	*db_unique_name*	Data Guard uses the value of the `DB_UNIQUE_NAME` initialization parameter.
FLASHBACK	Flashback logs	log_*log#*	—
CHANGETRACKING	Block change tracking data	ctf	Used during incremental backups

Table 8-1 (Cont.) Oracle File Types and Oracle ASM File Type Tags

Oracle ASM *file_type*	Description	Oracle ASM *file_type_tag*	Comments
DUMPSET	Data Pump dumpset	*user_obj#_file#*	Dump set files encode the user name, the job number that created the dump set, and the file number as part of the tag.
XTRANSPORT	Data file convert	*tsname*	—
AUTOBACKUP	Automatic backup files	*hasspfile_timestamp*	*hasspfile* can take one of two values: *s* indicates that the backup set includes the spfile; *n* indicates that the backup set does not include the spfile.

numeric_file_name

A numeric Oracle ASM filename is similar to a fully qualified filename except that it uses only the unique *filenumber.incarnation_number* string. You can use this form only to refer to an existing file. Therefore, if you are using this form during file creation, you must also specify REUSE.

incomplete_file_name

Incomplete Oracle ASM filenames are used during file creation only. If you specify the disk group name alone, then Oracle ASM uses the appropriate default template for the file type. For example, if you are creating a data file in a CREATE TABLESPACE statement, Oracle ASM uses the default DATAFILE template to create an Oracle ASM data file. If you specify the disk group name with a template, then Oracle ASM uses the specified template to create the file. In both cases, Oracle ASM also creates a fully qualified filename.

template_name

A template is a named collection of attributes. You can create templates and apply them to files in a disk group. You can determine the names of all Oracle ASM template names by querying the V$ASM_TEMPLATE data dictionary view. Refer to *diskgroup_template_clauses* for instructions on creating Oracle ASM templates.

You can specify *template* only during file creation. It appears in the incomplete and alias name forms of the *ASM_filename* diagram:

- If you specify *template* immediately after the disk group name, then Oracle ASM uses the specified template to create the file, and gives the file a fully qualified filename.
- If you specify template after specifying an alias, then Oracle ASM uses the specified template to create the file, gives the file a fully qualified filename, and also creates the alias so that you can subsequently use it to refer to the file. If the alias you specify refers to an existing file, then Oracle ASM ignores the template specification unless you also specify REUSE.

> **See Also:**
>
> *diskgroup_template_clauses* for information about the default templates

alias_file_name

An alias is a user-friendly name for an Oracle ASM file. You can use alias filenames during file creation or reference. You can specify a template with an alias, but only during file creation. To determine the alias names for Oracle ASM files, query the `V$ASM_ALIAS` data dictionary view.

If you are specifying an alias during file creation, then refer to *diskgroup_directory_clauses* and *diskgroup_alias_clauses* for instructions on specifying the full alias name.

SIZE Clause

Specify the size of the file in bytes. Use `K`, `M`, `G`, or `T` to specify the size in kilobytes, megabytes, gigabytes, or terabytes.

- For undo tablespaces, you must specify the `SIZE` clause for each data file. For other tablespaces, you can omit this parameter if the file already exists, or if you are creating an Oracle Managed File.
- If you omit this clause when creating an Oracle Managed File, then Oracle creates a 100M file.
- The size of a tablespace must be one block greater than the sum of the sizes of the objects contained in it.

> **See Also:**
>
> *Oracle Database Administrator's Guide* for information on automatic undo management and undo tablespaces and "Adding a Log File: Example"

BLOCKSIZE Clause

Specify `BLOCKSIZE` to override the operating system-dependent sector size. If you omit this clause, then the database uses the operating system-dependent sector size as the block size.

When you add a redo log file to a 512-byte sector disk or to a 4KB sector disk with 512-byte emulation, the blocksize of the new file must be the original platform base block size or 4KB.

- If the redo log file is being added to a 512-byte sector disk, then you must specify 512 or 1024 (or 1K) as the block size, depending on your platform.
- If the redo log file is being added to a 4KB sector disk (native), then you must specify either 4096 or 4K as the block size.
- If the redo log file is being added to a 4KB sector disk with 512-byte emulation, then you can specify either 512, 1024 (or 1K), or 4096 (or 4K) as the block size, depending on your platform.

All logs within a log group must have the same block size. Two log groups created on separate disks can have different block sizes. However, the mixed configuration introduces overhead at every log switch. Oracle recommends that you create all log files with the same block size.

This clause is useful when the 4K sector size is in use, but you want to optimize disk space use rather than performance. In such a case you can override the operating system sector size by specifying `BLOCKSIZE` 512 or, for HP-UX, `BLOCKSIZE` 1024.

> **See Also:**
>
> "Adding a Log File: Example"

REUSE

Specify `REUSE` to allow Oracle to reuse an existing file.

- If the file already exists, then Oracle reuses the filename and applies the new size (if you specify `SIZE`) or retains the original size.
- If the file does not exist, then Oracle ignores this clause and creates the file.

Restriction on the REUSE Clause

You cannot specify `REUSE` unless you have specified *`filename`*.

Whenever Oracle uses an existing file, the previous contents of the file are lost.

> **See Also:**
>
> "Adding a Data File: Example" and "Adding a Log File: Example"

autoextend_clause

The *`autoextend_clause`* is valid for data files and temp files but not for redo log files. Use this clause to enable or disable the automatic extension of a new or existing data file or temp file. If you omit this clause, then:

- For Oracle Managed Files:
  - If you specify `SIZE`, then Oracle Database creates a file of the specified size with `AUTOEXTEND` disabled.
  - If you do not specify `SIZE`, then the database creates a 100M file with `AUTOEXTEND` enabled. When autoextension is required, the database extends the file by its original size or 100MB, whichever is smaller. You can override this default behavior by specifying the `NEXT` clause.
- For user-managed files, with or without `SIZE` specified, Oracle creates a file with `AUTOEXTEND` disabled.

ON

Specify `ON` to enable autoextend.

OFF

Specify `OFF` to turn off autoextend if is turned on. When you turn off autoextend, the values of `NEXT` and `MAXSIZE` are set to zero. If you turn autoextend back on in a subsequent statement, then you must reset these values.

NEXT

Use the NEXT clause to specify the size in bytes of the next increment of disk space to be allocated automatically when more extents are required. The default is the size of one data block.

MAXSIZE

Use the MAXSIZE clause to specify the maximum disk space allowed for automatic extension of the data file.

UNLIMITED

Use the UNLIMITED clause if you do not want to limit the disk space that Oracle can allocate to the data file or temp file.

Restriction on the *autoextend_clause*

You cannot specify this clause as part of the *datafile_tempfile_spec* in a CREATE CONTROLFILE statement or in an ALTER DATABASE CREATE DATAFILE clause.

Examples

Specifying a Log File: Example

The following statement creates a database named payable that has two redo log file groups, each with two members, and one data file:

```
CREATE DATABASE payable
   LOGFILE GROUP 1 ('diska:log1.log', 'diskb:log1.log') SIZE 50K,
           GROUP 2 ('diska:log2.log', 'diskb:log2.log') SIZE 50K
   DATAFILE 'diskc:dbone.dbf' SIZE 30M;
```

The first file specification in the LOGFILE clause specifies a redo log file group with the GROUP value 1. This group has members named 'diska:log1.log' and 'diskb:log1.log', each 50 kilobytes in size.

The second file specification in the LOGFILE clause specifies a redo log file group with the GROUP value 2. This group has members named 'diska:log2.log' and 'diskb:log2.log', also 50 kilobytes in size.

The file specification in the DATAFILE clause specifies a data file named 'diskc:dbone.dbf', 30 megabytes in size.

Each file specification specifies a value for the SIZE parameter and omits the REUSE clause, so none of these files can already exist. Oracle must create them.

Adding a Log File: Example

The following statement adds another redo log file group with two members to the payable database:

```
ALTER DATABASE payable
   ADD LOGFILE GROUP 3 ('diska:log3.log', 'diskb:log3.log')
   SIZE 50K REUSE;
```

The file specification in the ADD LOGFILE clause specifies a new redo log file group with the GROUP value 3. This new group has members named 'diska:log3.log' and 'diskb:log3.log', each 50 kilobytes in size. Because the file specification specifies the REUSE clause, each member can (but need not) already exist.

The following statement adds a logfile group 5 with member log files on migration target disks `4k_disk_a` and `4k_disk_b`. After executing this statement, you can switch existing log files on disks with 512-byte block size to logs with 4K block size using the *switch_logfile_clause*.

```
ALTER DATABASE ADD LOGFILE GROUP 5
     ('4k_disk_a:log5.log', '4k_disk_b:log5.log')
     SIZE 100M BLOCKSIZE 4096 REUSE;
```

Specifying a Data File: Example

The following statement creates a tablespace named `stocks` that has three data files:

```
CREATE TABLESPACE stocks
   DATAFILE 'stock1.dbf' SIZE 10M,
            'stock2.dbf' SIZE 10M,
            'stock3.dbf' SIZE 10M;
```

The file specifications for the data files specify files named '`diskc:stock1.dbf`', '`diskc:stock2.dbf`', and '`diskc:stock3.dbf`'.

Adding a Data File: Example

The following statement alters the `stocks` tablespace and adds a new data file:

```
ALTER TABLESPACE stocks
   ADD DATAFILE 'stock4.dbf' SIZE 10M REUSE;
```

The file specification specifies a data file named '`stock4.dbf`'. If the filename does not exist, then Oracle simply ignores the `REUSE` keyword.

Using a Fully Qualified Oracle ASM Data File Name: Example

When using Oracle ASM, the following syntax shows how to use the *fully_qualified_file_name* clause to bring online a data file in a hypothetical database, `testdb`:

```
ALTER DATABASE testdb
   DATAFILE '+dgroup_01/testdb/datafile/system.261.1' ONLINE;
```

logging_clause

Purpose

The *logging_clause* lets you specify whether certain DML operations will be logged in the redo log file (`LOGGING`) or not (`NOLOGGING`).

You can specify the *logging_clause* in the following statements:

- `CREATE TABLE` and `ALTER TABLE`: for logging of the table, a table partition, a LOB segment, or the overflow segment of an index-organized table (see CREATE TABLE and ALTER TABLE).

> **Note:**
>
> Logging specified for a LOB column can differ from logging set at the table level. If you specify `LOGGING` at the table level and `NOLOGGING` for a LOB column, then DML changes to the base table row are logged, but DML changes to the LOB data are not logged.

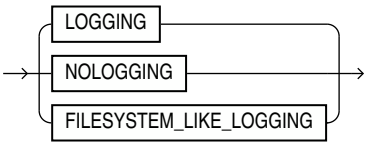
- `CREATE INDEX` and `ALTER INDEX`: for logging of the index or an index partition (see CREATE INDEX and ALTER INDEX).
- `CREATE MATERIALIZED VIEW` and `ALTER MATERIALIZED VIEW`: for logging of the materialized view, one of its partitions, or a LOB segment (see CREATE MATERIALIZED VIEW and ALTER MATERIALIZED VIEW).
- `CREATE MATERIALIZED VIEW LOG` and `ALTER MATERIALIZED VIEW LOG`: for logging of the materialized view log or one of its partitions (see CREATE MATERIALIZED VIEW LOG and ALTER MATERIALIZED VIEW LOG).
- `CREATE TABLESPACE` and `ALTER TABLESPACE`: to set or modify the default logging characteristics for all objects created in the tablespace (see CREATE TABLESPACE and ALTER TABLESPACE).
- `CREATE PLUGGABLE DATABASE` and `ALTER PLUGGABLE DATABASE`: to set or modify the default logging characteristics for all tablespaces created in the pluggable database (PDB) (see CREATE PLUGGABLE DATABASE and ALTER PLUGGABLE DATABASE).

You can also specify `LOGGING` or `NOLOGGING` for the following operations:

- Rebuilding an index (using `CREATE INDEX ... REBUILD`)
- Moving a table (using `ALTER TABLE ... MOVE`)

Syntax

***logging_clause*::=**



Semantics

This section describes the semantics of the *`logging_clause`*. For additional information, refer to the SQL statement in which you set or reset logging characteristics for a particular database object.

- If you specify `LOGGING`, then the creation of a database object, as well as subsequent inserts into the object, will be logged in the redo log file.
- If you specify `NOLOGGING`, then the creation of a database object, as well as subsequent conventional inserts, will be logged in the redo log file. Direct-path inserts will not be logged.
  - For a **nonpartitioned object**, the value specified for this clause is the actual physical attribute of the segment associated with the object.

  - For **partitioned objects**, the value specified for this clause is the default physical attribute of the segments associated with all partitions specified in the `CREATE` statement (and in subsequent `ALTER ... ADD PARTITION` statements), unless you specify the logging attribute in the `PARTITION` description.
  - For SecureFiles LOBs, the `NOLOGGING` setting is converted internally to `FILESYSTEM_LIKE_LOGGING`.
  - `CACHE NOLOGGING` is not allowed for BasicFiles LOBs.
- The `FILESYSTEM_LIKE_LOGGING` clause is valid only for logging of SecureFiles LOB segments. You cannot specify this setting for BasicFiles LOBs. Specify this setting if you want to log only metadata changes. This setting is similar to the metadata journaling of file systems, which reduces mean time to recovery from failures. The `LOGGING` setting, for SecureFiles LOBs, is similar to the data journaling of file systems. Both the `LOGGING` and `FILESYSTEM_LIKE_LOGGING` settings provide a complete transactional file system by way of SecureFiles.

> **Note:**
>
> For LOB segments, with the `NOLOGGING` and `FILESYSTEM_LIKE_LOGGING` settings it is possible for data to be changed on disk during a backup operation, resulting in an inconsistent backup. To avoid this situation, ensure that changes to LOB segments are saved in the redo log file by setting `LOGGING` for LOB storage. Alternatively, change the database to `FORCE LOGGING` mode so that changes to *all* LOB segments are saved in the redo.

If the object for which you are specifying the logging attributes resides in a database or tablespace in force logging mode, then Oracle Database ignores any `NOLOGGING` setting until the database or tablespace is taken out of force logging mode.

If the database is running in `ARCHIVELOG` mode, then media recovery from a backup made before the `LOGGING` operation re-creates the object. However, media recovery from a backup made before the `NOLOGGING` operation does not re-create the object.

The size of a redo log generated for an operation in `NOLOGGING` mode is significantly smaller than the log generated in `LOGGING` mode.

In `NOLOGGING` mode, data is modified with minimal logging (to mark new extents `INVALID` and to record dictionary changes). When applied during media recovery, the extent invalidation records mark a range of blocks as logically corrupt, because the redo data is not fully logged. Therefore, if you cannot afford to lose the database object, then you should take a backup after the `NOLOGGING` operation.

`NOLOGGING` is supported in only a subset of the locations that support `LOGGING`. Only the following operations support the `NOLOGGING` mode:

DML:

- Direct-path `INSERT` (serial or parallel) resulting either from an `INSERT` or a `MERGE` statement. `NOLOGGING` is not applicable to any `UPDATE` operations resulting from the `MERGE` statement.
- Direct Loader (SQL*Loader)

DDL:

- CREATE TABLE ... AS SELECT (In NOLOGGING mode, the creation of the table will be logged, but direct-path inserts will not be logged.)
- CREATE TABLE ... *LOB_storage_clause* ... *LOB_parameters* ... CACHE | NOCACHE | CACHE READS
- ALTER TABLE ... *LOB_storage_clause* ... *LOB_parameters* ... CACHE | NOCACHE | CACHE READS (to specify logging of newly created LOB columns)
- ALTER TABLE ... *modify_LOB_storage_clause* ... *modify_LOB_parameters* ... CACHE | NOCACHE | CACHE READS (to change logging of existing LOB columns)
- ALTER TABLE ... MOVE
- ALTER TABLE ... (all partition operations that involve data movement)
  - ALTER TABLE ... ADD PARTITION (hash partition only)
  - ALTER TABLE ... MERGE PARTITIONS
  - ALTER TABLE ... SPLIT PARTITION
  - ALTER TABLE ... MOVE PARTITION
  - ALTER TABLE ... MODIFY PARTITION ... ADD SUBPARTITION
  - ALTER TABLE ... MODIFY PARTITION ... COALESCE SUBPARTITION
- CREATE INDEX
- ALTER INDEX ... REBUILD
- ALTER INDEX ... REBUILD [SUB]PARTITION
- ALTER INDEX ... SPLIT PARTITION

For **objects other than LOBs**, if you omit this clause, then the logging attribute of the object defaults to the logging attribute of the tablespace in which it resides.

For **LOBs**, if you omit this clause, then:

- If you specify CACHE, then LOGGING is used (because you cannot have CACHE NOLOGGING).
- If you specify NOCACHE or CACHE READS, then the logging attribute defaults to the logging attribute of the tablespace in which it resides.

NOLOGGING does not apply to LOBs that are stored internally (in the table with row data). If you specify NOLOGGING for LOBs with values less than 4000 bytes and you have not disabled STORAGE IN ROW, then Oracle ignores the NOLOGGING specification and treats the LOB data the same as other table data.

parallel_clause

Purpose

The *parallel_clause* lets you parallelize the creation of a database object and set the default degree of parallelism for subsequent queries of and DML operations on the object.

You can specify the *parallel_clause* in the following statements:

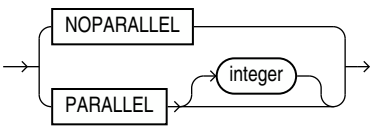
- CREATE TABLE: to set parallelism for the table (see CREATE TABLE).
- ALTER TABLE (see ALTER TABLE):
  - To change parallelism for the table

  - To parallelize the operations of adding, coalescing, exchanging, merging, splitting, truncating, dropping, or moving a table partition
- `CREATE CLUSTER` and `ALTER CLUSTER`: to set or alter parallelism for a cluster (see CREATE CLUSTER and ALTER CLUSTER).
- `CREATE INDEX`: to set parallelism for the index (see CREATE INDEX).
- `ALTER INDEX` (see ALTER INDEX):
  - To change parallelism for the index
  - To parallelize the rebuilding of the index or the splitting of an index partition
- `CREATE MATERIALIZED VIEW`: to set parallelism for the materialized view (see CREATE MATERIALIZED VIEW).
- `ALTER MATERIALIZED VIEW` (see ALTER MATERIALIZED VIEW):
  - To change parallelism for the materialized view
  - To parallelize the operations of adding, coalescing, exchanging, merging, splitting, truncating, dropping, or moving a materialized view partition
  - To parallelize the operations of adding or moving materialized view subpartitions
- `CREATE MATERIALIZED VIEW LOG`: to set parallelism for the materialized view log (see CREATE MATERIALIZED VIEW LOG).
- `ALTER MATERIALIZED VIEW LOG` (see ALTER MATERIALIZED VIEW LOG):
  - To change parallelism for the materialized view log
  - To parallelize the operations of adding, coalescing, exchanging, merging, splitting, truncating, dropping, or moving a materialized view log partition
- `ALTER DATABASE ... RECOVER`: to recover the database (see ALTER DATABASE).
- `ALTER DATABASE ... standby_database_clauses`: to parallelize operations on the standby database (see ALTER DATABASE).

> **See Also:**
>
> *Oracle Database PL/SQL Packages and Types Reference* for information on the `DBMS_PARALLEL_EXECUTE` package, which provides methods to apply table changes in chunks of rows. Changes to each chunk are independently committed when there are no errors.

Syntax

***parallel_clause*::=**



Semantics

This section describes the semantics of the *parallel_clause*. For additional information, refer to the SQL statement in which you set or reset parallelism for a particular database object or operation.

> **Note:**
>
> The syntax of the *parallel_clause* supersedes syntax appearing in earlier releases of Oracle. The superseded syntax is still supported for backward compatibility, but may result in slightly different behavior from that documented.

The database interprets the *parallel_clause* based on the setting of the PARALLEL_DEGREE_POLICY initialization parameter. When that parameter is set to AUTO, the *parallel_clause* is ignored entirely, and the optimizer determines the best degree of parallelism for all statements. When PARALLEL_DEGREE_POLICY is set to either MANUAL or LIMITED, the *parallel_clause* is interpreted as follows:

NOPARALLEL

Specify NOPARALLEL for serial execution. This is the default.

PARALLEL

Specify PARALLEL for parallel execution.

- If PARALLEL_DEGREE_POLICY is set to MANUAL, then the optimizer calculates a degree of parallelism equal to the number of CPUs available on all participating instances times the value of the PARALLEL_THREADS_PER_CPU initialization parameter.
- If PARALLEL_DEGREE_POLICY is set to LIMITED, then the optimizer determines the best degree of parallelism.

PARALLEL *integer*

Specification of *integer* indicates the **degree of parallelism**, which is the number of parallel threads used in the parallel operation. Each parallel thread may use one or two parallel execution servers.

Notes on the *parallel_clause*

The following notes apply to the *parallel_clause*:

- Parallelism is disabled for DML operations on tables on which you have defined a trigger or referential integrity constraint.
- Parallelism is not supported for UPDATE or DELETE operations on index-organized tables.
- When you specify the *parallel_clause* during creation of a table, if the table contains any columns of LOB or user-defined object type, then subsequent INSERT, UPDATE, DELETE or MERGE operations that modify the LOB or object type column are executed serially without notification. Subsequent queries, however, will be executed in parallel.
- A parallel hint overrides the effect of the *parallel_clause*.
- DML statements and CREATE TABLE ... AS SELECT statements that reference remote objects can run in parallel. However, the remote object must really be on a remote database. The

reference cannot loop back to an object on the local database, for example, by way of a synonym on the remote database pointing back to an object on the local database.

- DML operations on tables with LOB columns can be parallelized. However, intrapartition parallelism is not supported.

> **See Also:**
>
> *Oracle Database VLDB and Partitioning Guide* for more information on parallelized operations, and "Creating a Table: Parallelism Examples"

physical_attributes_clause

Purpose

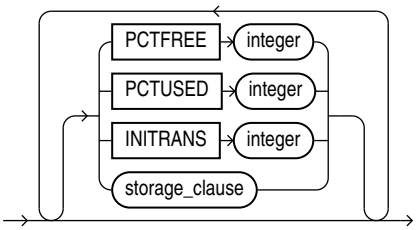
The *physical_attributes_clause* lets you specify the value of the PCTFREE, PCTUSED, and INITRANS parameters and the storage characteristics of a table, cluster, index, or materialized view.

You can specify the *physical_attributes_clause* in the following statements:

- CREATE CLUSTER and ALTER CLUSTER: to set or change the physical attributes of the cluster and all tables in the cluster (see CREATE CLUSTER and ALTER CLUSTER).
- CREATE TABLE: to set the physical attributes of the table, a table partition, the OIDINDEX of an object table, or the overflow segment of an index-organized table (see CREATE TABLE).
- ALTER TABLE: to change the physical attributes of the table, the default physical attributes of future table partitions, or the physical attributes of existing table partitions (see ALTER TABLE). The following restrictions apply:
  - You cannot specify physical attributes for a temporary table.
  - You cannot specify physical attributes for a clustered table. Tables in a cluster inherit the physical attributes of the cluster.
- CREATE INDEX: to set the physical attributes of an index or index partition (see CREATE INDEX).
- ALTER INDEX: to change the physical attributes of the index, the default physical attributes of future index partitions, or the physical attributes of existing index partitions (see ALTER INDEX).
- CREATE MATERIALIZED VIEW: to set the physical attributes of the materialized view, one of its partitions, or the index Oracle Database generates to maintain the materialized view (see CREATE MATERIALIZED VIEW).
- ALTER MATERIALIZED VIEW: to change the physical attributes of the materialized view, the default physical attributes of future partitions, the physical attributes of an existing partition, or the index Oracle creates to maintain the materialized view (see ALTER MATERIALIZED VIEW).
- CREATE MATERIALIZED VIEW LOG and ALTER MATERIALIZED VIEW LOG: to set or change the physical attributes of the materialized view log (see CREATE MATERIALIZED VIEW LOG and ALTER MATERIALIZED VIEW LOG).

Syntax

***physical_attributes_clause*::=**



(*storage_clause*::=)

Semantics

This section describes the parameters of the *physical_attributes_clause*. For additional information, refer to the SQL statement in which you set or reset these parameters for a particular database object.

PCTFREE *integer*

Specify a whole number representing the percentage of space in each data block of the database object reserved for future updates to rows of the object. The value of PCTFREE must be a value from 0 to 99. A value of 0 means that the entire block can be filled by inserts of new rows. The default value is 10. This value reserves 10% of each block for updates to existing rows and allows inserts of new rows to fill a maximum of 90% of each block.

PCTFREE has the same function in the statements that create and alter tables, partitions, clusters, indexes, materialized views, materialized view logs, and zone maps. The combination of PCTFREE and PCTUSED determines whether new rows will be inserted into existing data blocks or into new blocks. See "How PCTFREE and PCTUSED Work Together".

Restriction on the PCTFREE Clause

When altering an index, you can specify this parameter only in the *modify_index_default_attrs* clause and the *split_index_partition* clause.

PCTUSED *integer*

Specify a whole number representing the minimum percentage of used space that Oracle maintains for each data block of the database object. PCTUSED is specified as a positive integer from 0 to 99 and defaults to 40.

PCTUSED has the same function in the statements that create and alter tables, partitions, clusters, materialized views, materialized view logs, and zone maps.

PCTUSED is not a valid table storage characteristic for an index-organized table.

The sum of PCTFREE and PCTUSED must be equal to or less than 100. You can use PCTFREE and PCTUSED together to utilize space within a database object more efficiently. See "How PCTFREE and PCTUSED Work Together".

Restrictions on the PCTUSED Clause

The PCTUSED parameter is subject to the following restrictions:

- You cannot specify this parameter for an index or for the index segment of an index-organized table.
- This parameter is not useful and is ignored for objects with automatic segment-space management.

> **See Also:**
>
> *Oracle Database Performance Tuning Guide* for information on the performance effects of different values of `PCTUSED` and `PCTFREE` and `CREATE TABLESPACE` *segment_management_clause* for information on automatic segment-space management

How PCTFREE and PCTUSED Work Together

In a newly allocated data block, the space available for inserts is the block size minus the sum of the block overhead and free space (`PCTFREE`). Updates to existing data can use any available space in the block. Therefore, updates can reduce the available space of a block to less than `PCTFREE`.

After a data block is filled to the limit determined by `PCTFREE`, Oracle Database considers the block unavailable for the insertion of new rows until the percentage of that block falls beneath the parameter `PCTUSED`. Until this value is achieved, Oracle Database uses the free space of the data block only for updates to rows already contained in the data block. A block becomes a candidate for row insertion when its used space falls below `PCTUSED`.

> **See Also:**
>
> FREELISTS for information on how `PCTUSED` and `PCTFREE` work with freelist segment space management

INITRANS *integer*

Specify the initial number of concurrent transaction entries allocated within each data block allocated to the database object. This value can range from 1 to 255 and defaults to 1, with the following exceptions:

- The default `INITRANS` value for a cluster is 2 or the default `INITRANS` value of the tablespace in which the cluster resides, whichever is greater.
- The default value for an index is 2.

In general, you should not change the `INITRANS` value from its default.

Each transaction that updates a block requires a transaction entry in the block. This parameter ensures that a minimum number of concurrent transactions can update the block and helps avoid the overhead of dynamically allocating a transaction entry.

The `INITRANS` parameter serves the same purpose in the statements that create and alter tables, partitions, clusters, indexes, materialized views, and materialized view logs.

MAXTRANS Parameter

In earlier releases, the `MAXTRANS` parameter determined the maximum number of concurrent update transactions allowed for each data block in the segment. This parameter has been deprecated. Oracle now automatically allows up to 255 concurrent update transactions for any data block, depending on the available space in the block. Note that the maximum number of concurrent update transactions is based on the size of the block

Existing objects for which a value of `MAXTRANS` has already been set retain that setting. However, if you attempt to change the value for `MAXTRANS`, Oracle ignores the new specification and substitutes the value 255 without returning an error.

storage_clause

The *`storage_clause`* lets you specify storage characteristics for the table, object table `OIDINDEX`, partition, LOB data segment, or index-organized table overflow data segment. This clause has performance ramifications for large tables. Storage should be allocated to minimize dynamic allocation of additional space. Refer to the *storage_clause* for more information.

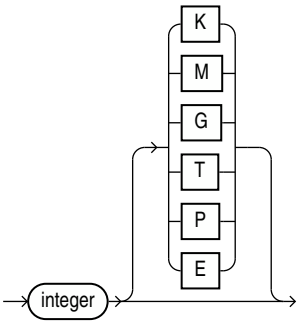
size_clause

Purpose

The *`size_clause`* lets you specify a number of bytes, kilobytes (K), megabytes (M), gigabytes (G), terabytes (T), petabytes (P), or exabytes (E) in any statement that lets you establish amounts of disk or memory space.

Syntax

***size_clause*::=**



Semantics

Use the *`size_clause`* to specify a number or multiple of bytes. If you do not specify any of the multiple abbreviations, then the *`integer`* is interpreted as bytes.

> **Note:**
>
> Not all multiples of bytes are appropriate in all cases, and context-sensitive limitations may apply. In the latter case, Oracle issues an error message.

storage_clause

Purpose

The *storage_clause* lets you specify how Oracle Database should store a permanent database object. Storage parameters for temporary segments always use the default storage parameters for the associated tablespace. Storage parameters affect both how long it takes to access data stored in the database and how efficiently space in the database is used.

> **See Also:**
>
> *Oracle Automatic Storage Management Administrator's Guide* for a discussion of the effects of the storage parameters

When you create a cluster, index, materialized view, materialized view log, rollback segment, table, LOB, varray, nested table, or partition, you can specify values for the storage parameters for the segments allocated to these objects. If you omit any storage parameter, then Oracle uses the value of that parameter specified for the tablespace in which the object resides. If no value was specified for the tablespace, then the database uses default values.

> **Note:**
>
> The specification of storage parameters for objects in locally managed tablespaces is supported for backward compatibility. If you are using locally managed tablespaces, then you can omit these storage parameter when creating objects in those tablespaces.

When you alter a cluster, index, materialized view, materialized view log, rollback segment, table, varray, nested table, or partition, you can change the values of storage parameters. The new values affect only future extent allocations.

The *storage_clause* is part of the *physical_attributes_clause*, so you can specify this clause in any of the statements where you can specify the physical attributes clause (see *physical_attributes_clause*). In addition, you can specify the *storage_clause* in the following statements:

- CREATE CLUSTER and ALTER CLUSTER: to set or change the storage characteristics of the **cluster and all tables in the cluster** (see CREATE CLUSTER and ALTER CLUSTER).
- CREATE INDEX and ALTER INDEX: to set or change the storage characteristics of an index segment created for a **table index or index partition** or an **index segment created for an index used to enforce a primary key or unique constraint** (see CREATE INDEX and ALTER INDEX).
- The ENABLE ... USING INDEX clause of CREATE TABLE or ALTER TABLE: to set or change the storage characteristics of an **index created by the system to enforce a primary key or unique constraint**.
- CREATE MATERIALIZED VIEW and ALTER MATERIALIZED VIEW: to set or change the storage characteristics of a **materialized view, one of its partitions, or the index Oracle**

generates to maintain the materialized view (see CREATE MATERIALIZED VIEW and ALTER MATERIALIZED VIEW).

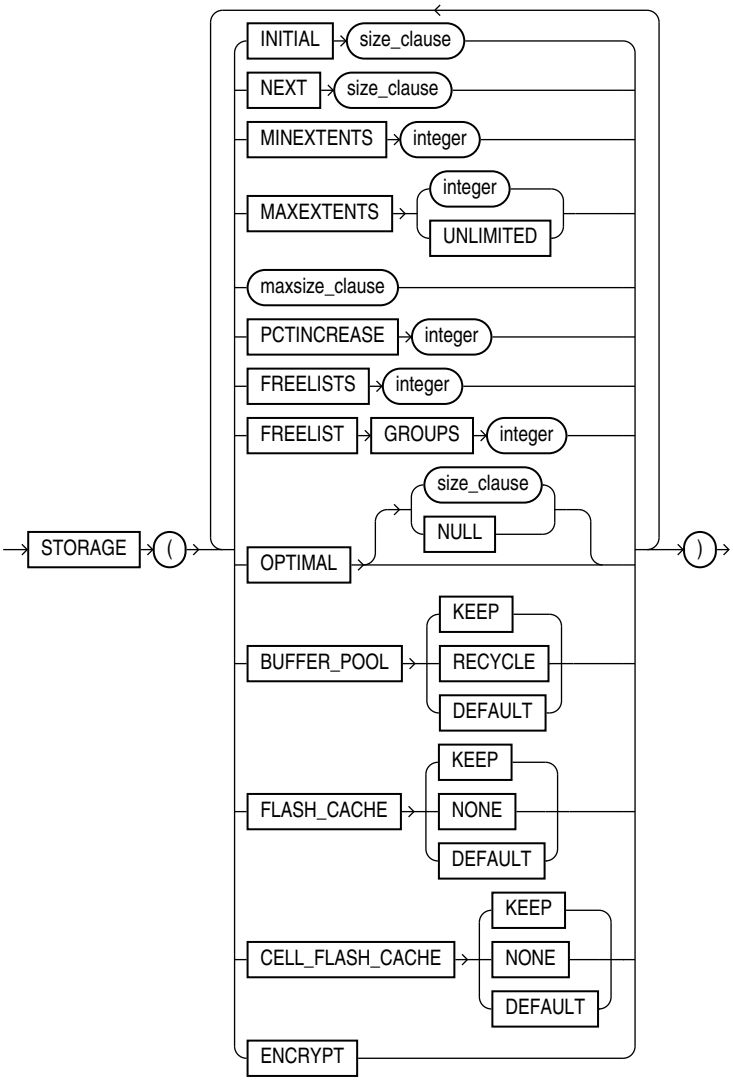
- `CREATE MATERIALIZED VIEW LOG` and `ALTER MATERIALIZED VIEW LOG`: to set or change the storage characteristics of **the materialized view log** (see CREATE MATERIALIZED VIEW LOG and ALTER MATERIALIZED VIEW LOG).
- `CREATE ROLLBACK SEGMENT` and `ALTER ROLLBACK SEGMENT`: to set or change the storage characteristics of a **rollback segment** (see CREATE ROLLBACK SEGMENT and ALTER ROLLBACK SEGMENT).
- `CREATE TABLE` and `ALTER TABLE`: to set the storage characteristics of a **LOB or varray data segment** of the nonclustered table or one of its partitions or subpartitions, or the **storage table of a nested table** (see CREATE TABLE and ALTER TABLE).
- `CREATE TABLESPACE` and `ALTER TABLESPACE`: to set or change the default storage characteristics for **objects created in the tablespace** (see CREATE TABLESPACE and ALTER TABLESPACE). Changes to tablespace storage parameters affect only new objects created in the tablespace or new extents allocated for a segment.
- *`constraint`*: to specify storage for the **index (and its partitions, if it is a partitioned index) used to enforce the constraint** (see *constraint*).

Prerequisites

To change the value of a `STORAGE` parameter, you must have the privileges necessary to use the appropriate `CREATE` or `ALTER` statement.

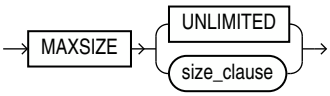
Syntax

***storage_clause*::=**



(*size_clause*::=)

***maxsize_clause*::=**



(*size_clause*::=)

Semantics

This section describes the parameters of the *storage_clause*. For additional information, refer to the SQL statement in which you set or reset these storage parameters for a particular database object.

> **Note:**
>
> The *storage_clause* is interpreted differently for locally managed tablespaces. For locally managed tablespaces, Oracle Database uses INITIAL, NEXT, PCTINCREASE, and MINEXTENTS to compute how many extents are allocated when the object is first created. After object creation, these parameters are ignored. For more information, see CREATE TABLESPACE .

> **See Also:**
>
> "Specifying Table Storage Attributes: Example"

INITIAL

Specify the size of the first extent of the object. Oracle allocates space for this extent when you create the schema object. Refer to *size_clause* for information on that clause.

In locally managed tablespaces, Oracle uses the value of INITIAL, in conjunction with the type of local management—AUTOALLOCATE or UNIFORM—and the values of MINEXTENTS, NEXT and PCTINCREASE, to determine the initial size of the segment.

- With AUTOALLOCATE extent management, Oracle uses the INITIAL setting to optimize the number of extents allocated. Extents of 64K, 1M, 8M, and 64M can be allocated. During segment creation, the system chooses the greatest of these four sizes that is equal to or smaller than INITIAL, and allocates as many extents of that size as are needed to reach the INITIAL setting. For example, if you set INITIAL to 4M, then the database creates four 1M extents.
- For UNIFORM extent management, the number of extents is determined from initial segment size and the uniform extent size specified at tablespace creation time. For example, in a uniform locally managed tablespace with 1M extents, if you specify an INITIAL value of 5M, then Oracle creates five 1M extents.

  Consider this comparison: With AUTOALLOCATE, if you set INITAL to 72K, then the initial segment size will be 128K (greater than INITIAL). The database cannot allocate an extent smaller than 64K, so it must allocate two 64K extents. If you set INITIAL to 72K with a UNIFORM extent size of 24K, then the database will allocate three 24K extents to equal 72K.

In dictionary managed tablespaces, the default initial extent size is 5 blocks, and all subsequent extents are rounded to 5 blocks. If MINIMUM EXTENT was specified at tablespace creation time, then the extent sizes are rounded to the value of MINIMUM EXTENT.

Restriction on INITIAL

You cannot specify INITIAL in an ALTER statement.

NEXT

Specify in bytes the size of the next extent to be allocated to the object. Refer to *size_clause* for information on that clause.

In locally managed tablespaces, any user-supplied value for `NEXT` is ignored and the size of `NEXT` is determined by Oracle if the tablespace is set for autoallocate extent management. In `UNIFORM` tablespaces, the size of `NEXT` is the uniform extent size specified at tablespace creation time.

In dictionary-managed tablespaces, the default value is the size of 5 data blocks. The minimum value is the size of 1 data block. The maximum value depends on your operating system. Oracle rounds values up to the next multiple of the data block size for values less than 5 data blocks. For values greater than 5 data blocks, Oracle rounds up to a value that minimizes fragmentation.

> **See Also:**
>
> *Oracle Database Concepts* for information on how Oracle minimizes fragmentation

PCTINCREASE

In locally managed tablespaces, Oracle Database uses the value of `PCTINCREASE` during segment creation to determine the initial segment size and ignores this parameter during subsequent space allocation.

In dictionary-managed tablespaces, specify the percent by which the third and subsequent extents grow over the preceding extent. The default value is 50, meaning that each subsequent extent is 50% larger than the preceding extent. The minimum value is 0, meaning all extents after the first are the same size. The maximum value depends on your operating system. Oracle rounds the calculated size of each new extent to the nearest multiple of the data block size. If you change the value of the `PCTINCREASE` parameter by specifying it in an `ALTER` statement, then Oracle calculates the size of the next extent using this new value and the size of the most recently allocated extent.

Restriction on PCTINCREASE

You cannot specify `PCTINCREASE` for rollback segments. Rollback segments always have a `PCTINCREASE` value of 0.

MINEXTENTS

In locally managed tablespaces, Oracle Database uses the value of `MINEXTENTS` in conjunction with `PCTINCREASE`, `INITIAL` and `NEXT` to determine the initial segment size.

In dictionary-managed tablespaces, specify the total number of extents to allocate when the object is created. The default and minimum value is 1, meaning that Oracle allocates only the initial extent, except for rollback segments, for which the default and minimum value is 2. The maximum value depends on your operating system.

- In a locally managed tablespace, `MINEXTENTS` is used to compute the initial amount of space allocated, which is equal to `INITIAL` * `MINEXTENTS`. Thereafter this value is set to 1, which is reflected in the `DBA_SEGMENTS` view.
- In a dictionary-managed tablespace, `MINEXTENTS` is simply the minimum number of extents that must be allocated to the segment.

If the `MINEXTENTS` value is greater than 1, then Oracle calculates the size of subsequent extents based on the values of the `INITIAL`, `NEXT`, and `PCTINCREASE` storage parameters.

When changing the value of MINEXTENTS by specifying it in an ALTER statement, you can reduce the value from its current value, but you cannot increase it. Resetting MINEXTENTS to a smaller value might be useful, for example, before a TRUNCATE ... DROP STORAGE statement, if you want to ensure that the segment will maintain a minimum number of extents after the TRUNCATE operation.

Restrictions on MINEXTENTS

The MINEXTENTS storage parameter is subject to the following restrictions:

- MINEXTENTS is not applicable at the tablespace level.
- You cannot change the value of MINEXTENTS in an ALTER statement or for an object that resides in a locally managed tablespace.

MAXEXTENTS

This storage parameter is valid only for objects in dictionary-managed tablespaces. Specify the total number of extents, including the first, that Oracle can allocate for the object. The minimum value is 1 except for rollback segments, which always have a minimum of 2. The default value depends on your data block size.

Restriction on MAXEXTENTS

MAXEXTENTS is ignored for objects residing in a locally managed tablespace, unless the value of ALLOCATION_TYPE is USER for the tablespace in the DBA_TABLESPACES data dictionary view.

> **See Also:**
>
> *Oracle Database Reference* for more information on the DBA_TABLESPACES data dictionary view

UNLIMITED

Specify UNLIMITED if you want extents to be allocated automatically as needed. Oracle recommends this setting as a way to minimize fragmentation.

Do not use this clause for rollback segments. Doing so allows the possibility that long-running rogue DML transactions will continue to create new extents until a disk is full.

> **Note:**
>
> A rollback segment that you create without specifying the *storage_clause* has the same storage parameters as the tablespace in which the rollback segment is created. Thus, if you create a tablespace with MAXEXTENTS UNLIMITED, then the rollback segment will have this same default.

MAXSIZE

The MAXSIZE clause lets you specify the maximum size of the storage element. For LOB storage, MAXSIZE has the following effects

- If you specify RETENTION MAX in *LOB_parameters*, then the LOB segment increases to the specified size before any space can be reclaimed from undo space.

- If you specify RETENTION AUTO, MIN, or NONE in *LOB_parameters*, then the specified size is a hard limit on the LOB segment size and has no bearing on undo retention.

UNLIMITED

Use the UNLIMITED clause if you do not want to limit the disk space of the storage element. This clause is not compatible with a specification of RETENTION MAX in *LOB_parameters*. If you specify both, then the database uses RETENTION AUTO and MAXSIZE UNLIMITED.

FREELISTS

In tablespaces with manual segment-space management, Oracle Database uses the FREELISTS storage parameter to improve performance of space management in OLTP systems by increasing the number of insert points in the segment. In tablespaces with automatic segment-space management, this parameter is ignored, because the database adapts to varying workload.

In tablespaces with manual segment-space management, for objects other than tablespaces and rollback segments, specify the number of free lists for each of the free list groups for the table, partition, cluster, or index. The default and minimum value for this parameter is 1, meaning that each free list group contains one free list. The maximum value of this parameter depends on the data block size. If you specify a FREELISTS value that is too large, then Oracle returns an error indicating the maximum value.

This clause is not valid or useful if you have specified the SECUREFILE parameter of *LOB_parameters*. If you specify both the SECUREFILE parameter and FREELISTS, then the database silently ignores the FREELISTS specification.

Restriction on FREELISTS

You can specify FREELISTS in the *storage_clause* of any statement except when creating or altering a tablespace or rollback segment.

FREELIST GROUPS

In tablespaces with manual segment-space management, Oracle Database uses the value of this storage parameter to statically partition the segment free space in an Oracle Real Application Clusters environment. This partitioning improves the performance of space allocation and deallocation by avoiding inter instance transfer of segment metadata. In tablespaces with automatic segment-space management, this parameter is ignored, because Oracle dynamically adapts to inter instance workload.

In tablespaces with manual segment-space management, specify the number of groups of free lists for the database object you are creating. The default and minimum value for this parameter is 1. Oracle uses the instance number of Oracle Real Application Clusters (Oracle RAC) instances to map each instance to one free list group.

Each free list group uses one database block. Therefore:

- If you do not specify a large enough value for INITIAL to cover the minimum value plus one data block for each free list group, then Oracle increases the value of INITIAL the necessary amount.
- If you are creating an object in a uniform locally managed tablespace, and the extent size is not large enough to accommodate the number of freelist groups, then the create operation will fail.

This clause is not valid or useful if you have specified the SECUREFILE parameter of *LOB_parameters*. If you specify both the SECUREFILE parameter and FREELIST GROUPS, then the database silently ignores the FREELIST GROUPS specification.

Restriction on FREELIST GROUPS

You can specify the FREELIST GROUPS parameter only in CREATE TABLE, CREATE CLUSTER, CREATE MATERIALIZED VIEW, CREATE MATERIALIZED VIEW LOG, and CREATE INDEX statements.

OPTIMAL

The OPTIMAL keyword is relevant only to rollback segments. It specifies an optimal size in bytes for a rollback segment. Refer to *size_clause* for information on that clause.

Oracle tries to maintain this size for the rollback segment by dynamically deallocating extents when their data is no longer needed for active transactions. Oracle deallocates as many extents as possible without reducing the total size of the rollback segment below the OPTIMAL value.

The value of OPTIMAL cannot be less than the space initially allocated by the MINEXTENTS, INITIAL, NEXT, and PCTINCREASE parameters. The maximum value depends on your operating system. Oracle rounds values up to the next multiple of the data block size.

NULL

Specify NULL for no optimal size for the rollback segment, meaning that Oracle never deallocates the extents of the rollback segment. This is the default behavior.

BUFFER_POOL

The BUFFER_POOL clause lets you specify a default buffer pool or cache for a schema object. All blocks for the object are stored in the specified cache.

- If you define a buffer pool for a partitioned table or index, then the partitions inherit the buffer pool from the table or index definition unless overridden by a partition-level definition.
- For an index-organized table, you can specify a buffer pool separately for the index segment and the overflow segment.

Restrictions on the BUFFER_POOL Parameter

BUFFER_POOL is subject to the following restrictions:

- You cannot specify this clause for a cluster table. However, you can specify it for a cluster.
- You cannot specify this clause for a tablespace or a rollback segment.

KEEP

Specify KEEP to put blocks from the segment into the KEEP buffer pool. Maintaining an appropriately sized KEEP buffer pool lets Oracle retain the schema object in memory to avoid I/O operations. KEEP takes precedence over any NOCACHE clause you specify for a table, cluster, materialized view, or materialized view log.

RECYCLE

Specify RECYCLE to put blocks from the segment into the RECYCLE pool. An appropriately sized RECYCLE pool reduces the number of objects whose default pool is the RECYCLE pool from taking up unnecessary cache space.

DEFAULT

Specify DEFAULT to indicate the default buffer pool. This is the default for objects not assigned to KEEP or RECYCLE.

> **See Also:**
>
> *Oracle Database Performance Tuning Guide* for more information about using multiple buffer pools

FLASH_CACHE

The `FLASH_CACHE` clause lets you override the automatic buffer cache policy and specify how specific schema objects are cached in flash memory. To use this clause, Database Smart Flash Cache (flash cache) must be configured on your system. The flash cache is an extension of the database buffer cache that is stored on a flash disk, a storage device that uses flash memory. Because flash memory is faster than magnetic disks, the database can improve performance by caching buffers in the flash cache instead of reading from magnetic disk.

KEEP

Specify `KEEP` if you want the schema object buffers to remain cached in the flash cache as long as the flash cache is large enough.

NONE

Specify `NONE` to ensure that the schema object buffers are never cached in the flash cache. This allows you to reserve the flash cache space for more frequently accessed objects.

DEFAULT

Specify `DEFAULT` if you want the schema object buffers to be written to the flash cache when they are aged out of main memory, and then be aged out of the flash cache with the standard buffer cache replacement algorithm. This is the default if flash cache is configured and you do not specify `KEEP` or `NONE`.

> **Note:**
>
> Database Smart Flash Cache is available only in Solaris and Oracle Linux.

> **See Also:**
>
> - *Oracle Database Concepts* for more information about Database Smart Flash Cache
> - *Oracle Database Administrator's Guide* to learn how to configure Database Smart Flash Cache

ENCRYPT

This clause is valid only when you are creating a tablespace. Specify `ENCRYPT` to encrypt the entire tablespace. You must also specify the `ENCRYPTION` clause in the `CREATE TABLESPACE` statement.

> **Note:**
>
> The `ENCRYPT` clause is supported for backward compatibility. However, beginning with Oracle Database 12*c* Release 2 (12.2), you can instead specify `ENCRYPT` in the *`tablespace_encryption_clause`*. Refer to the *tablespace_encryption_clause* of `CREATE TABLESPACE` for more information.

Example

Specifying Table Storage Attributes: Example

The following statement creates a table and provides storage parameter values:

```
CREATE TABLE divisions
    (div_no     NUMBER(2),
     div_name   VARCHAR2(14),
     location   VARCHAR2(13) )
     STORAGE  ( INITIAL 8M MAXSIZE 1G );
```

The following statement queries the table for the size of the first extent:

```
SELECT INITIAL_EXTENT FROM USER_TABLES WHERE TABLE_NAME='DIVISIONS';

INITIAL_EXTENT
--------------
       8388608
```

Oracle allocates space for the table based on the `STORAGE` parameter values as follows:

- The `INITIAL` value is 8M, so the size of the first extent is 8 megabytes.
- The `MAXSIZE` value is 1G, so the maximum size of the storage element is 1 gigabyte.

annotations_clause

Purpose

Annotations provide a mechanism to store application metadata centrally in the database, so that they can be shared across applications, modules and microservices.

You can add annotations to any supported schema objects that you own at creation time via `CREATE` statements.

On supported schema objects that you have alter privileges on, you can add and drop annotations via `ALTER` statements. You do not need to qualify the annotation name with the schema name. Whenever a schema object drops an annotation, or when a schema object is dropped altogether, the usage of the annotation is updated to reflect the drop.

An individual annotation has a name and an optional value. Both the name and the value are freeform text fields. Annotations are additive, meaning that multiple annotations can be specified for the same schema object in a single DDL.

When an annotation name is specified for a schema object for the first time, an annotation is automatically created. Supported schema objects include tables, views, materialized views, and indexes. The annotation is represented as a subordinate element to the database object to which it has been added. Whenever a schema object drops an annotation, or when a schema object is dropped altogether, the usage of the annotation is updated to reflect the drop.

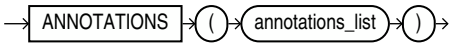
Dictionary views track the list of annotations and their usage across all schema objects. You can query dictionary views `USER|ALL|DBA_ANNOTATIONS_USAGE` to list the annotations for a schema object.

Prerequisites

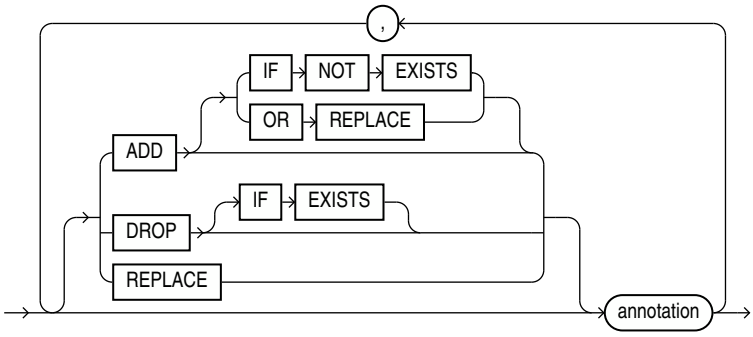
You must own the schema object or have `ALTER` privileges on the schema object in order to specify annotations on the object.

Syntax

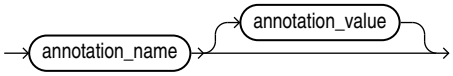
***annotations_clause*::=**



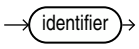
***annotations_list*::=**



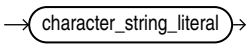
***annotation*::=**



***annotation_name*::=**



***annotation_value*::=**



Semantics

annotations_clause

Specify `ADD`, `DROP`, or `REPLACE` to create, remove, or change annotations respectively.

- `ADD` creates *`annotation_name`*. This is the default when no keyword is specified before annotation. If the object already has an annotation with this name, the statement raises an error.

  Use `ADD IF NOT EXISTS` to allow the statement to complete without error. If *`annotation_name`* is already present, it keeps its original value when using the `IF NOT EXISTS` clause.

  `ADD [IF NOT EXISTS]` is the only valid option to use with `CREATE` statements.

- `DROP` removes *`annotation_name`* from the object. If the object has no annotation with this name, the statement raises an error. Use `DROP IF EXISTS` to allow the statement to complete without error. This clause is only valid in `ALTER` statements.

- `REPLACE` changes *`annotation_value`* for *`annotation_name`* to the supplied value. If you omit the value, this removes any existing value for *`annotation_name`*. If *`annotation_name`* does not exist the statement will raise an error. This clause is only valid in `ALTER` statements.

The *`annotation_name`* is an identifier that can have up to 1024 characters. If the annotation name is a reserved word it must be provided in double quotes. When a double quoted identifier is used, the identifier can also contain whitespace characters. However, identifiers that contain only whitespace characters are not accepted.

An annotation is either a name-value pair or a name by itself. The name and the optional value are freeform text fields. Value can have a maximum of 4000 characters. An annotation `Display_Label, 'Employee Salary'` has a name and a value, whereas an annotation `UI_Hidden` has only a name and it does not need a value. `UI_Hidden` is a standalone annotation used to specify that the column should be hidden.

Examples

Add Annotations to a Table

The following example adds two operations with values *`Sort`* and *`Group`*, and a standalone *`Hidden`* without a value, to table *`t1`*:

```
CREATE TABLE t1 (T NUMBER) ANNOTATIONS(Operations '["Sort", "Group"]', Hidden);
```

The annotation can be preceded by the keyword `ADD` which is the default operation if nothing is specified as the following example shows:

```
CREATE TABLE t1 (T NUMBER) ANNOTATIONS (ADD Hidden);
```

Alter Annotations at the Table Level

The following example drops all annotations from `t1`:

```
ALTER TABLE t1 ANNOTATIONS(DROP Operations, DROP Hidden);
```

Add Annotations to Table Columns

```
CREATE TABLE t1 (T NUMBER ANNOTATIONS(Operations 'Sort' , Hidden) );
```

Add Annotations to Table and Columns

```
CREATE TABLE employee (
  id NUMBER(5)
    ANNOTATIONS(Identity, Display 'Employee ID', "Group" 'Emp_Info'),
  ename VARCHAR2(50)
    ANNOTATIONS(Display 'Employee Name', "Group" 'Emp_Info'),
  sal NUMBER
    ANNOTATIONS(Display 'Employee Salary', UI_Hidden)
) ANNOTATIONS (Display 'Employee Table');
```

Alter Annotations at the Column Level

```
ALTER TABLE employee
  MODIFY ename ANNOTATIONS (
    DROP "Group",
    DROP IF EXISTS missing_annotation,
    REPLACE Display 'Emp name'
  );
```