19
SQL Statements: MERGE to UPDATE

This chapter contains the following SQL statements:

MERGE

Purpose

Use the `MERGE` statement to select rows from one or more sources for update or insertion into a table or view. You can specify conditions to determine whether to update or insert into the target table or view.

This statement is a convenient way to combine multiple operations. It lets you avoid multiple `INSERT`, `UPDATE`, and `DELETE` DML statements.

`MERGE` is a deterministic statement. You cannot update the same row of the target table multiple times in the same `MERGE` statement.

Note:

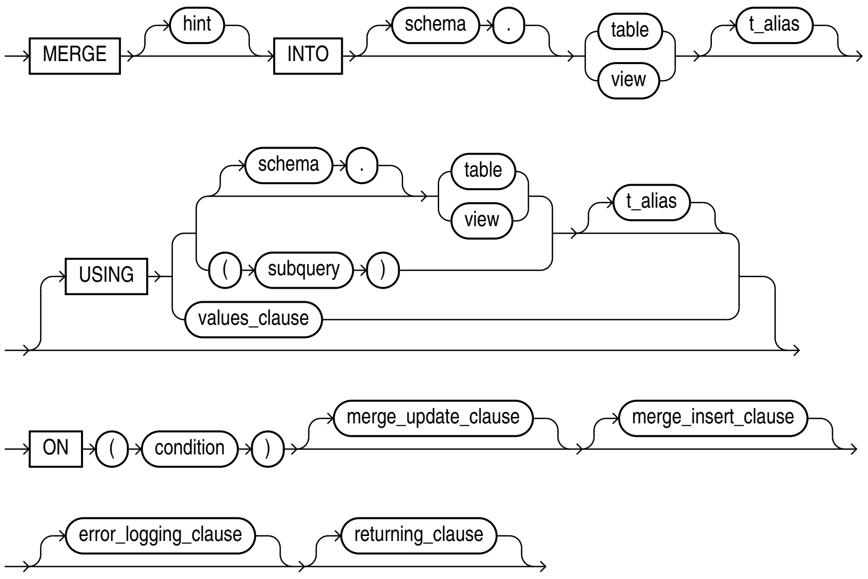
In previous releases of Oracle Database, when you created an Oracle Virtual Private Database policy on an application that included the `MERGE INTO` statement, the `MERGE INTO` statement would be prevented with an `ORA-28132: Merge into syntax does not support security policies` error, due to the presence of the Virtual Private Database policy. Beginning with Oracle Database 11*g* Release 2 (11.2.0.2), you can create policies on applications that include `MERGE INTO` operations. To do so, in the `DBMS_RLS.ADD_POLICY statement_types` parameter, include the `INSERT`, `UPDATE`, and `DELETE` statements, or just omit the `statement_types` parameter altogether. Refer to *Oracle Database Security Guide* for more information on enforcing policies on specific SQL statement types.

Prerequisites

You must have the `INSERT` and `UPDATE` object privileges on the target table and the `SELECT` object privilege on the source objects. To specify the `DELETE` clause of the *`merge_update_clause`*, you must also have the `DELETE` object privilege on the target table or view.

Syntax

***merge*::=**

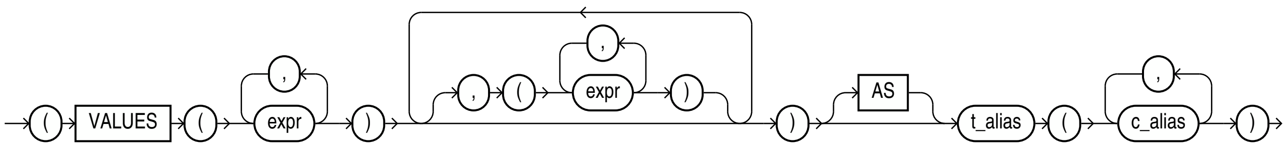


Note:

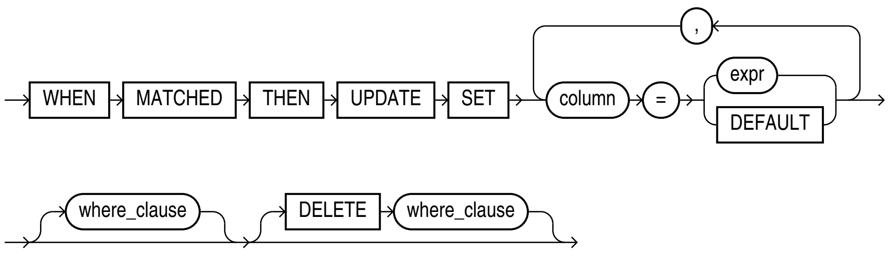
You must specify at least one of the clauses *`merge_update_clause`* or *`merge_insert_clause`*.

(*values_clause*::=, *merge_update_clause*::=, *merge_insert_clause*::=, *error_logging_clause*::=

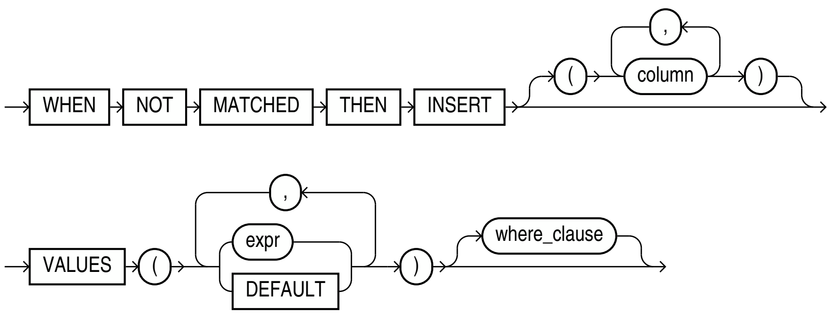
***values_clause*::=**



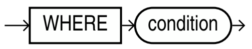
***merge_update_clause*::=**



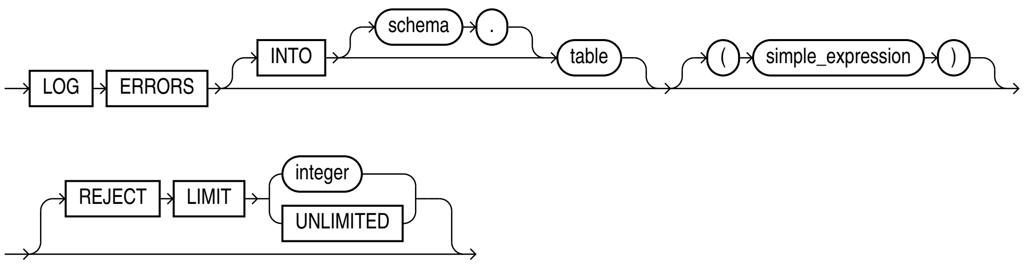
***merge_insert_clause*::=**



***where_clause*::=**



***error_logging_clause*::=**



Semantics

INTO Clause

Use the `INTO` clause to specify the target table or view you are updating or inserting into. In order to merge data into a view, the view must be updatable. Refer to "Notes on Updatable Views" for more information.

Restriction on Target Views

You cannot specify a target view on which an `INSTEAD OF` trigger has been defined.

USING Clause

Use the `USING` clause to specify the source you are updating or inserting from.

values_clause

For semantics of the `values_clause` please see the `values_clause` of the `SELECT` statement *values_clause* .

ON Clause

Use the `ON` clause to specify the condition upon which the `MERGE` operation either updates or inserts. For each row in the target table for which the search condition is true, Oracle Database updates the row with corresponding data from the source . If the condition is not true for any rows, then the database inserts into the target table based on the corresponding source row.

merge_update_clause

The `merge_update_clause` specifies the new column values of the target table or view. Oracle performs this update if the condition of the `ON` clause is true. If the update clause is executed, then all update triggers defined on the target table are activated.

Specify the `where_clause` if you want the database to execute the update operation only if the specified condition is true. The condition can refer to either the data source or the target table. If the condition is not true, then the database skips the update operation when merging the row into the table.

Specify the `DELETE` `where_clause` to clean up data in a table while populating or updating it. The only rows affected by this clause are those rows in the destination table that are updated by the merge operation. The `DELETE WHERE` condition evaluates the updated value, not the original value that was evaluated by the `UPDATE SET` ... `WHERE` condition. If a row of the destination table meets the `DELETE` condition but is not included in the join defined by the `ON` clause, then it is not deleted. Any delete triggers defined on the target table will be activated for each row deletion.

You can specify this clause by itself or with the `merge_insert_clause`. If you specify both, then they can be in either order.

Restrictions on the *merge_update_clause*

This clause is subject to the following restrictions:

- You cannot update a column that is referenced in the `ON` `condition` clause.
- You cannot specify `DEFAULT` when updating a view.

merge_insert_clause

The *merge_insert_clause* specifies values to insert into the column of the target table if the condition of the ON clause is false. If the insert clause is executed, then all insert triggers defined on the target table are activated. If you omit the column list after the INSERT keyword, then the number of columns in the target table must match the number of values in the VALUES clause.

To insert all of the source rows into the table, you can use a **constant filter predicate** in the ON clause condition. An example of a constant filter predicate is ON (0=1). Oracle Database recognizes such a predicate and makes an unconditional insert of all source rows into the table. This approach is different from omitting the *merge_update_clause*. In that case, the database still must perform a join. With constant filter predicate, no join is performed.

Specify the *where_clause* if you want Oracle Database to execute the insert operation only if the specified condition is true. The condition can refer only to the data source columns. Oracle Database skips the insert operation for all rows for which the condition is not true.

You can specify the *merge_insert_clause* by itself or with the *merge_update_clause*. If you specify both, then they can be in either order.

Restriction on the *merge_insert_clause*

You cannot specify DEFAULT when inserting into a view.

error_logging_clause

The *error_logging_clause* has the same behavior in a MERGE statement as in an INSERT statement. Refer to the INSERT statement *error_logging_clause* for more information.

See Also:

"Inserting Into a Table with Error Logging: Example"

Examples

Merging into a Table: Example

The following example uses the bonuses table in the sample schema oe with a default bonus of 100. It then inserts into the bonuses table all employees who made sales, based on the sales_rep_id column of the oe.orders table. Finally, the human resources manager decides that employees with a salary of $8000 or less should receive a bonus. Those who have not made sales get a bonus of 1% of their salary. Those who already made sales get an increase in their bonus equal to 1% of their salary. The MERGE statement implements these changes in one step:

```
CREATE TABLE bonuses (employee_id NUMBER, bonus NUMBER DEFAULT 100);
INSERT INTO bonuses(employee_id)
   (SELECT e.employee_id FROM hr.employees e, oe.orders o
   WHERE e.employee_id = o.sales_rep_id
   GROUP BY e.employee_id);

SELECT * FROM bonuses ORDER BY employee_id;

EMPLOYEE_ID      BONUS
```

```
----------- ----------
        153        100
        154        100
        155        100
        156        100
        158        100
        159        100
        160        100
        161        100
        163        100

MERGE INTO bonuses D
   USING (SELECT employee_id, salary, department_id FROM hr.employees
   WHERE department_id = 80) S
   ON (D.employee_id = S.employee_id)
   WHEN MATCHED THEN UPDATE SET D.bonus = D.bonus + S.salary*.01
     DELETE WHERE (S.salary > 8000)
   WHEN NOT MATCHED THEN INSERT (D.employee_id, D.bonus)
     VALUES (S.employee_id, S.salary*.01)
     WHERE (S.salary <= 8000);

SELECT * FROM bonuses ORDER BY employee_id;

EMPLOYEE_ID      BONUS
----------- ----------
        153        180
        154        175
        155        170
        159        180
        160        175
        161        170
        164         72
        165         68
        166         64
        167         62
        171         74
        172         73
        173         61
        179         62
```

Conditional Insert and Update: Example

The following example conditionally inserts and updates table data by using the `MERGE` statement.

The following statements create two tables named `people_source` and `people_target` and populate them with names:

```
CREATE TABLE people_source (
  person_id  INTEGER NOT NULL PRIMARY KEY,
  first_name VARCHAR2(20) NOT NULL,
  last_name  VARCHAR2(20) NOT NULL,
  title      VARCHAR2(10) NOT NULL
);

CREATE TABLE people_target (
  person_id  INTEGER NOT NULL PRIMARY KEY,
  first_name VARCHAR2(20) NOT NULL,
  last_name  VARCHAR2(20) NOT NULL,
  title      VARCHAR2(10) NOT NULL
);

INSERT INTO people_target VALUES (1, 'John', 'Smith', 'Mr');
INSERT INTO people_target VALUES (2, 'alice', 'jones', 'Mrs');
INSERT INTO people_source VALUES (2, 'Alice', 'Jones', 'Mrs.');
INSERT INTO people_source VALUES (3, 'Jane', 'Doe', 'Miss');
INSERT INTO people_source VALUES (4, 'Dave', 'Brown', 'Mr');

COMMIT;
```

The following statement compares the contents of `people_target` and `people_source` by using the `person_id` column. The values in the `people_target` table are updated when there is a match in the `people_source` table:

```
MERGE INTO people_target pt
USING people_source ps
ON    (pt.person_id = ps.person_id)
WHEN MATCHED THEN UPDATE
  SET pt.first_name = ps.first_name,
      pt.last_name = ps.last_name,
      pt.title = ps.title;
```

The following statements display the contents of the `people_target` table and perform a rollback:

```
SELECT * FROM people_target;

PERSON_ID  FIRST_NAME           LAST_NAME            TITLE
---------- -------------------- -------------------- ----------
     1   John              Smith            Mr
     2   Alice             Jones            Mrs.

ROLLBACK;
```

This statement compares the contents of the `people_target` and `people_source` tables by using the `person_id` column. The values in the `people_target` table are updated only when there is no match in the `people_source` table:

```
MERGE INTO people_target pt
USING people_source ps
ON    (pt.person_id = ps.person_id)
WHEN NOT MATCHED THEN INSERT
  (pt.person_id, pt.first_name, pt.last_name, pt.title)
  VALUES (ps.person_id, ps.first_name, ps.last_name, ps.title);
```

The following statements display the contents of the `people_target` table and perform a rollback:

```
SELECT * FROM people_target;

PERSON_ID  FIRST_NAME          LAST_NAME          TITLE
---------- -------------------- -------------------- ----------
     1  John            Smith            Mr
     2  alice           jones            Mrs
     3  Jane            Doe              Miss
     4  Dave            Brown            Mr

ROLLBACK;
```

The following statement compares the contents of the `people_target` and `people_source` tables by using the `person_id` column and conditionally inserts and updates data in the `people_target` table. For each matching row in the `people_source` table, the values in the `people_target` table are updated by using the values from the `people_source` table. Any unmatched rows from the `people_source` table are added to the `people_target` table:

```
MERGE INTO people_target pt
USING people_source ps
ON    (pt.person_id = ps.person_id)
WHEN MATCHED THEN UPDATE
  SET pt.first_name = ps.first_name,
      pt.last_name = ps.last_name,
      pt.title = ps.title
WHEN NOT MATCHED THEN INSERT
  (pt.person_id, pt.first_name, pt.last_name, pt.title)
  VALUES (ps.person_id, ps.first_name, ps.last_name, ps.title);
```

The following statements display the contents of the `people_target` table and perform a rollback:

```
SELECT * FROM people_target;

PERSON_ID  FIRST_NAME    LAST_NAME          TITLE
---------- -------------   ------------------ ----------
     1  John         Smith         Mr
     2  Alice        Jones         Mrs.
     3  Jane         Doe           Miss
     4  Dave         Brown         Mr

ROLLBACK;
```

The following statement compares the `people_target` and `people_source` tables by using the `person_id` column. When the `person_id` matches, the corresponding rows in the `people_target` table are updated by using values from the `people_source` table. The `DELETE` clause removes all the values in `people_target` where `title` is 'Mrs.'. When the `person_id` does not match, the rows from the `people_source` table are added to the `people_target` table.

The `WHERE` clause ensures that only values that have `title` as ‘Mr’ are added to the `people_target` table:

```
MERGE INTO people_target pt
USING people_source ps
ON    (pt.person_id = ps.person_id)
WHEN MATCHED THEN UPDATE
  SET pt.first_name = ps.first_name,
      pt.last_name = ps.last_name,
      pt.title = ps.title
  DELETE where pt.title  = 'Mrs.'
WHEN NOT MATCHED THEN INSERT
  (pt.person_id, pt.first_name, pt.last_name, pt.title)
  VALUES (ps.person_id, ps.first_name, ps.last_name, ps.title)
  WHERE ps.title = 'Mr';
```

The following statements display the contents of the `people_target` table and perform a rollback:

```
SELECT * FROM people_target;

PERSON_ID   FIRST_NAME        LAST_NAME         TITLE
---------- -------------------- -------------------- ----------
     1    John              Smith             Mr
     4    Dave              Brown             Mr

ROLLBACK;
```

Dealing with Inputs from an Application

Usually applications have to check for the existence of a row first in order to decide whether to `INSERT` a new row, or `UPDATE` an already existing one. The `MERGE` statement eliminates the need for such a check by allowing the use of bind variables inside the `USING` statement as a source.

The following statements demonstrate the use of bind variables to insert a new row into the `people_target` table:

```
var person_id  NUMBER;
var first_name VARCHAR2(20);
var last_name  VARCHAR2(20);
var title      VARCHAR2(10);

exec :person_id := 3;
exec :first_name := 'Gerald';
exec :last_name := 'Walker';
exec :title := 'Mr';

MERGE INTO people_target
  ON (person_id = :person_id)
WHEN MATCHED THEN UPDATE
SET first_name = :first_name,
    last_name = :last_name,
    title = :title
WHEN NOT MATCHED THEN INSERT
    (person_id, first_name, last_name, title)
    VALUES (:person_id, :first_name, :last_name, :title);
```

The following statements display the contents of the `people_target` table and perform a rollback:

```
SELECT * FROM people_target;

 PERSON_ID FIRST_NAME           LAST_NAME            TITLE
---------- -------------------- -------------------- ----------
     1    John                  Smith                Mr
     2    alice                 jones                Mrs
     3    Gerald                Walker               Mr

ROLLBACK;
```

The following statements demonstrate the use of bind variables to update an already existing row in the `people_target`. Note that the `MERGE` statement is identical to the one just used to insert a new row:

```
var person_id  NUMBER;
var first_name VARCHAR2(20);
var last_name  VARCHAR2(20);
var title      VARCHAR2(10);

exec :person_id := 2;
exec :first_name := 'Alice';
exec :last_name := 'Jones';
exec :title := 'Mrs';

MERGE INTO people_target
   ON (person_id = :person_id)
WHEN MATCHED THEN UPDATE
SET first_name = :first_name,
    last_name = :last_name,
    title = :title
WHEN NOT MATCHED THEN INSERT
    (person_id, first_name, last_name, title)
    VALUES (:person_id, :first_name, :last_name, :title);
```

The following statements display the contents of the `people_target` table and perform a rollback:

```
SELECT * FROM people_target;

PERSON_ID FIRST_NAME        LAST_NAME            TITLE
---------- -------------------- -------------------- ----------
     1    John                Smith                Mr
     2    Alice               Jones                Mrs
```

```
ROLLBACK;
```

NOAUDIT (Traditional Auditing)

See Also:

Traditional auditing is desupported in 23ai. Databases migrated from earlier versions may still have traditional auditing policies configured. Those policies should be migrated to unified auditing. When migrated, the traditional audit policies may be removed with the `NOAUDIT` statement.

This section describes the `NOAUDIT` statement for **traditional auditing**, which is the same auditing functionality used in releases earlier than Oracle Database 12*c*.

Beginning with Oracle Database 12*c*, Oracle introduces **unified auditing**, which provides a full set of enhanced auditing features. For backward compatibility, traditional auditing is still supported. However, Oracle recommends that you plan the migration of your existing audit settings to the new unified audit policy syntax. For new audit requirements, Oracle recommends that you use the new unified auditing. Traditional auditing may be desupported in a future major release.

See Also:

NOAUDIT (Unified Auditing) for a description of the `NOAUDIT` statement for unified auditing

Purpose

Use the `NOAUDIT` statement to stop auditing operations previously enabled by the `AUDIT` statement.

The `NOAUDIT` statement must have the same syntax as the previous `AUDIT` statement. Further, it reverses the effects only of that particular statement. For example, suppose one `AUDIT` statement A enables auditing for a specific user. A second statement B enables auditing for all users. A `NOAUDIT` statement C to disable auditing for all users reverses statement B. However, statement C leaves statement A in effect and continues to audit the user that statement A specified.

Prerequisites

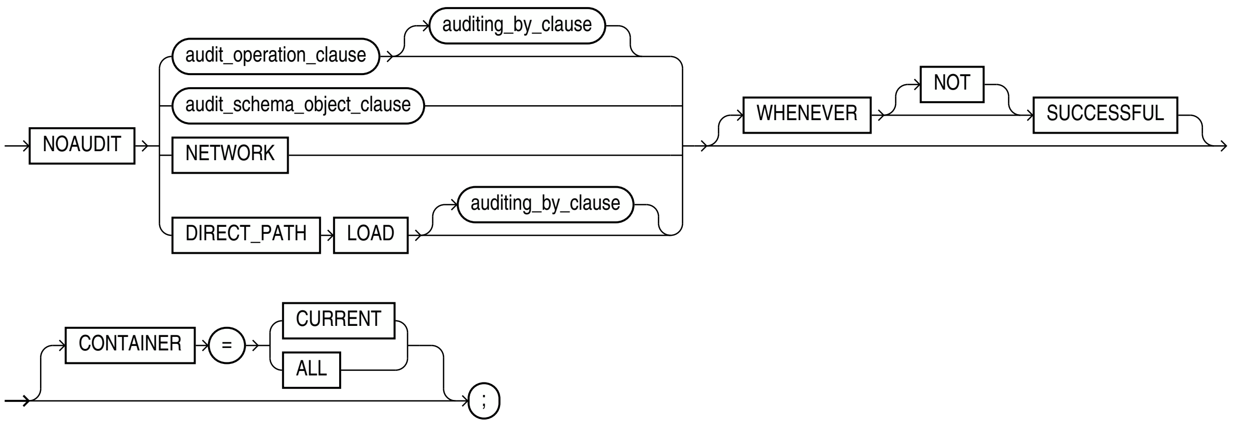
To stop auditing of SQL statements, you must have the `AUDIT SYSTEM` system privilege.

To stop auditing of schema objects, you must be the owner of the object on which you stop auditing or you must have the `AUDIT ANY` system privilege. In addition, if the object you chose for auditing is a directory, then even if you created it, you must have the `AUDIT ANY` system privilege.

To specify the `CONTAINER` clause, you must be connected to a multitenant container database (CDB). To specify `CONTAINER = ALL`, the current container must be the root and you must have the commonly granted `AUDIT SYSTEM` privilege in order to stop auditing for the issuances of a SQL statement, or the commonly granted `AUDIT ANY` privilege in order to stop auditing for the operations on a schema object. To specify `CONTAINER = CURRENT`, the current container must be a pluggable database (PDB) and you must have the locally granted `AUDIT SYSTEM` privilege in order to stop auditing the issuances of a SQL statement, or the locally granted `AUDIT ANY` privilege in order to stop auditing operations on a schema object.

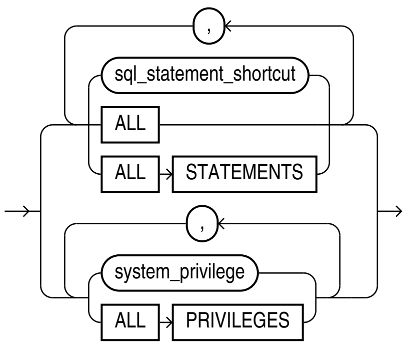
Syntax

noaudit::=

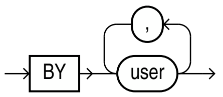


(*audit_operation_clause*::=, *auditing_by_clause*::=, *audit_schema_object_clause*::=)

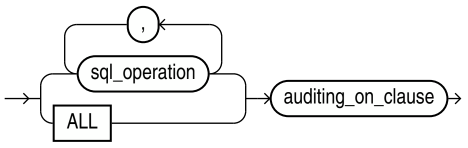
audit_operation_clause::=



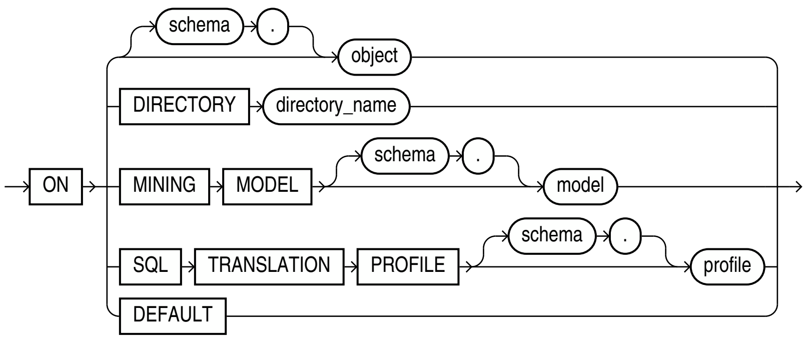
auditing_by_clause::=



***audit_schema_object_clause*::=**



***auditing_on_clause*::=**



Semantics

audit_operation_clause

Use the *`audit_operation_clause`* to stop auditing of a particular SQL statement.

statement_option

For *`sql_statement_shortcut`*, specify the shortcut for the SQL statements for which auditing is to be stopped.

ALL

Specify `ALL` to stop auditing of all statement options currently being audited because of an earlier `AUDIT ALL` . . . statement. You cannot use this clause to reverse an earlier `AUDIT ALL STATEMENTS` . . . statement.

ALL STATEMENTS

Specify `ALL STATEMENTS` to reverse an earlier `AUDIT ALL STATEMENTS` . . . statement. You cannot use this clause to reverse an earlier `AUDIT ALL` . . . statement.

system_privilege

For *`system_privilege`*, specify the system privilege for which auditing is to be stopped. Refer to Table 18-2 for a list of the system privileges and the statements they authorize.

ALL PRIVILEGES

Specify `ALL PRIVILEGES` to stop auditing of all system privileges currently being audited.

auditing_by_clause

Use the *auditing_by_clause* to stop auditing only for SQL statements issued by the specified users in their subsequent sessions. If you omit this clause, then Oracle Database stops auditing for all users' statements, except for the situation described for WHENEVER SUCCESSFUL.

audit_schema_object_clause

Use the *audit_schema_object_clause* to stop auditing of a particular database object.

sql_operation

For *sql_operation*, specify the type of operation for which auditing is to be stopped on the object specified in the ON clause.

ALL

Specify ALL as a shortcut equivalent to specifying all SQL operations applicable for the type of object.

auditing_on_clause

The *auditing_on_clause* lets you specify the particular schema object for which auditing is to be stopped.

- For object, specify the object name of a table, view, sequence, stored procedure, function, or package, materialized view, or library. If you do not qualify *object* with *schema*, then Oracle Database assumes the object is in your own schema.
- The DIRECTORY clause lets you specify the name of the directory on which auditing is to be stopped.
- The SQL TRANSLATION PROFILE clause lets you specify the SQL translation profile on which auditing is to be stopped.
- Specify DEFAULT to remove the specified object options as default object options for subsequently created objects.

NETWORK

Use this clause to discontinue auditing of database link usage and logins.

DIRECT_PATH LOAD

Use this clause to discontinue auditing of SQL*Loader direct path loads.

WHENEVER [NOT] SUCCESSFUL

Specify WHENEVER SUCCESSFUL to stop auditing only for SQL statements and operations on schema objects that complete successfully.

Specify WHENEVER NOT SUCCESSFUL to stop auditing only for SQL statements and operations that result in Oracle Database errors.

If you omit this clause, then the database stops auditing for all statements or operations, regardless of success or failure.

CONTAINER Clause

Use the CONTAINER clause to specify the scope of the NOAUDIT command.

- Specify `CONTAINER = CURRENT` to stop auditing in the PDB to which you are connected. If you specify the *`auditing_by_clause`*, then *`user`* must be a common user or local user in the current PDB. If you specify the *`auditing_on_clause`*, then the objects must be local objects in the current PDB.
- Specify `CONTAINER = ALL` to stop auditing across the entire CDB. If you specify the *`auditing_by_clause`*, then *`user`* must be a common user. If you do not specify the *`auditing_by_clause`*, then auditing is stopped for all common users and all local users in each PDB. If you specify the *`auditing_on_clause`*, then the objects must be common objects.

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

Examples

Stop Auditing of SQL Statements Related to Roles: Example

If you have chosen auditing for every SQL statement that creates or drops a role, then you can stop auditing of such statements by issuing the following statement:

```
NOAUDIT ROLE;
```

Stop Auditing of Updates or Queries on Objects Owned by a Particular User: Example

If you have chosen auditing for any statement that queries or updates any table issued by the users `hr` and `oe`, then you can stop auditing for queries by `hr` by issuing the following statement:

```
NOAUDIT SELECT TABLE BY hr;
```

The preceding statement stops auditing only queries by `hr`, so the database continues to audit queries and updates by `oe` as well as updates by `hr`.

Stop Auditing of Statements Authorized by a Particular Object Privilege: Example

To stop auditing on all statements that are authorized by `DELETE ANY TABLE` system privilege, issue the following statement:

```
NOAUDIT DELETE ANY TABLE;
```

Stop Auditing of Queries on a Particular Object: Example

If you have chosen auditing for every SQL statement that queries the `employees` table in the schema `hr`, then you can stop auditing for such queries by issuing the following statement:

```
NOAUDIT SELECT
   ON hr.employees;
```

Stop Auditing of Queries that Complete Successfully: Example

You can stop auditing for queries that complete successfully by issuing the following statement:

```
NOAUDIT SELECT
   ON hr.employees
   WHENEVER SUCCESSFUL;
```

This statement stops auditing only for successful queries. Oracle Database continues to audit queries resulting in Oracle Database errors.

NOAUDIT (Unified Auditing)

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

Purpose

Use the `NOAUDIT` statement to:

- Disable a unified audit policy for all users or for specified users
- Exclude the values of context attributes from audit records

Changes made to the audit policy become effective immediately in the current session and in all active sessions without re-login.

See Also:

- AUDIT (Unified Auditing)
- CREATE AUDIT POLICY (Unified Auditing)
- ALTER AUDIT POLICY (Unified Auditing)
- DROP AUDIT POLICY (Unified Auditing)

Prerequisites

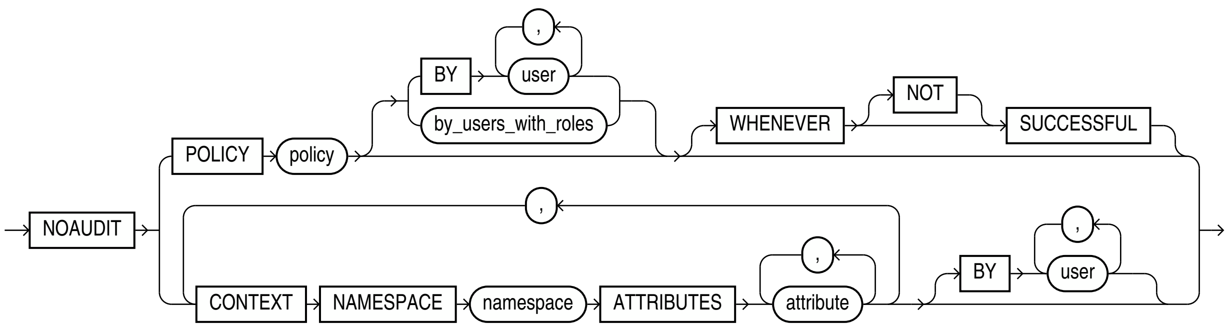
You must have the `AUDIT SYSTEM` system privilege or the `AUDIT_ADMIN` role.

If you are connected to a multitenant container database (CDB), then to disable a common unified audit policy, the current container must be the root and you must have the commonly granted `AUDIT SYSTEM` privilege or the `AUDIT_ADMIN` common role. To disable a local unified audit policy, the current container must be the container in which the audit policy was created and you must have the commonly granted `AUDIT SYSTEM` privilege or the `AUDIT_ADMIN` common role, or you must have the locally granted `AUDIT SYSTEM` privilege or the `AUDIT_ADMIN` local role in the container.

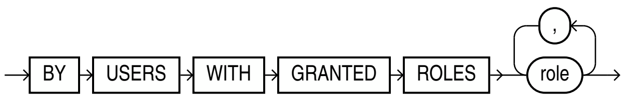
To specify the `NOAUDIT CONTEXT` ... statement when connected to a CDB, you must have the commonly granted `AUDIT SYSTEM` privilege or the `AUDIT_ADMIN` common role, or you must have the locally granted `AUDIT SYSTEM` privilege or the `AUDIT_ADMIN` local role in the current session's container.

Syntax

***unified_noaudit*::=**



***by_users_with_roles*::=**



Semantics

policy

Specify the name of the unified audit policy you want to disable.

You can find descriptions of all unified audit policies by querying the `AUDIT_UNIFIED_POLICIES` view and descriptions of all *enabled* unified audit policies by querying the `AUDIT_UNIFIED_ENABLED_POLICIES` view.

See Also:

Oracle Database Reference for more information on the `AUDIT_UNIFIED_POLICIES` and `AUDIT_UNIFIED_ENABLED_POLICIES` views

CONTEXT Clause

Specify the `CONTEXT` clause to exclude the values of context attributes in audit records.

- For *`namespace`*, specify the context namespace.
- For *`attribute`*, specify one or more context attributes whose values you want to exclude from audit records.

If you specify the `CONTEXT` clause when the current container is the root of a CDB, then the values of context attributes will be included in audit records only for events executed in the root. If you specify the optional `BY` clause, then *`user`* must be a common user.

If you specify the `CONTEXT` clause when the current container is a pluggable database (PDB), then the values of context attributes will be included in audit records only for events executed in that PDB. If you specify the optional BY clause, then *user* must be a common user or a local user in that PDB.

You can find the application context attributes that are configured to be captured in the audit trail by querying the AUDIT_UNIFIED_CONTEXTS view.

See Also:

Oracle Database Reference for more information on the AUDIT_UNIFIED_CONTEXTS view

BY

You can specify the BY clause for the NOAUDIT POLICY and NOAUDIT CONTEXT statements.

NOAUDIT POLICY ... BY

The behavior of the BY clause depends on whether *policy* is enabled for all users or specific users.

- If *policy* is enabled for all users, then you can disable *policy* for all users by omitting the BY clause. If you specify the BY clause, then the NOAUDIT POLICY statement will have no effect.
- If *policy* is enabled for one or more users (using the AUDIT POLICY ... BY ... statement), then you can:
  - Disable *policy* for one or more of those users by specifying the BY clause followed by the users for whom you want *policy* disabled
  - Completely disable *policy* by specifying the BY clause followed by all of the users for whom *policy* is enabled

  If you do not specify the BY clause, then the NOAUDIT POLICY statement will have no effect.
- If *policy* is enabled for all users except specific users (using the AUDIT POLICY ... EXCEPT ... statement), then you can disable *policy* for all users by omitting the BY clause. If you specify the BY clause, then the NOAUDIT POLICY statement will have no effect.

If *policy* is a common unified audit policy, then *user* must be a common user. If *policy* is a local unified audit policy, then *user* must be a common user or a local user in the container to which you are connected.

NOAUDIT CONTEXT ... BY

The behavior of the BY clause depends on whether *attribute* is configured to be included in audit records for all users or specific users.

- If *attribute* is configured to be included in audit records for all users, then you can exclude *attribute* from audit records for all users by omitting the BY clause. If you specify the BY clause, then the NOAUDIT CONTEXT statement will have no effect.
- If *attribute* is configured to be included in audit records for specific users, then you can exclude *attribute* for one or more of those users by specifying the BY clause followed by the users for whom you want *attribute* excluded. If you do not specify the BY clause, then the NOAUDIT CONTEXT statement will have no effect.

by_users_with_roles

Specify this clause to disable *policy* only for users who have been directly granted the specified roles. If you subsequently grant one of the roles to an additional user, then the policy is automatically disabled for that user.

When you are connected to a CDB, if *policy* is a common unified audit policy, then *role* must be a common role. If *policy* is a local unified audit policy, then role must be a common role or a local role in the container to which you are connected.

Examples

The following examples disable unified audit policies that were created in the CREATE AUDIT POLICY "Examples" and enabled in the AUDIT "Examples".

Disabling a Unified Audit Policy for All Users: Example

Assume that unified audit policy table_pol is enabled for all users. The following statement disables table_pol for all users:

```
NOAUDIT POLICY table_pol;
```

The following statement returns no rows, which verifies that table_pol is disabled for all users:

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

Disabling a Unified Audit Policy for Specific Users: Example

Assume that unified audit policy dml_pol is enabled for users hr and sh, as shown by the following query:

```
SELECT policy_name, enabled_option, entity_name
  FROM audit_unified_enabled_policies
  WHERE policy_name = 'DML_POL'
  ORDER BY entity_name;

POLICY_NAME  ENABLED_OPTION  ENTITY_NAME
-----------  -----------  ---------
DML_POL      BY           HR
DML_POL      BY           SH
```

The following statement disables dml_pol for user hr:

```
NOAUDIT POLICY dml_pol BY hr;
```

The following statement verifies that dml_pol is now enabled for only user sh:

```
SELECT policy_name, enabled_option, entity_name
  FROM audit_unified_enabled_policies
  WHERE policy_name = 'DML_POL';

POLICY_NAME  ENABLED_OPTION  ENTITY_NAME
-----------  -----------  ---------
DML_POL      BY           SH
```

The following statement disables dml_pol for user sh:

```
NOAUDIT POLICY dml_pol BY sh;
```

The following statement returns no rows, which verifies that `dml_pol` is disabled for all users:

```
SELECT *
  FROM audit_unified_enabled_policies
  WHERE policy_name = 'DML_POL';
```

Excluding Values of Context Attributes in Audit Records: Example

The following statement instructs the database to exclude the values of namespace `USERENV` attributes `CURRENT_USER` and `DB_NAME` from all audit records for user `hr`:

```
NOAUDIT CONTEXT NAMESPACE userenv
  ATTRIBUTES current_user, db_name
  BY hr;
```

PURGE

Purpose

Use the `PURGE` statement to:

- Remove a table or index from your recycle bin and release all of the space associated with the object
- Remove part or all of a dropped tablespace or tablespace set from the recycle bin
- Remove the entire recycle bin

Note:

You cannot roll back a `PURGE` statement, nor can you recover an object after it is purged.

To see the contents of your recycle bin, query the `USER_RECYCLEBIN` data dictionary view. You can use the `RECYCLEBIN` synonym instead. The following two statements return the same rows:

```
SELECT * FROM RECYCLEBIN;
SELECT * FROM USER_RECYCLEBIN;
```

See Also:

- *Oracle Database Administrator's Guide* for information on the recycle bin and naming conventions for objects in the recycle bin
- FLASHBACK TABLE for information on retrieving dropped tables from the recycle bin
- *Oracle Database Reference* for information on using the `RECYCLEBIN` initialization parameter to control whether dropped tables go into the recycle bin

Prerequisites

To purge a table, the table must reside in your own schema or you must have the `DROP ANY TABLE` system privilege, or you must have the `SYSDBA` system privilege.

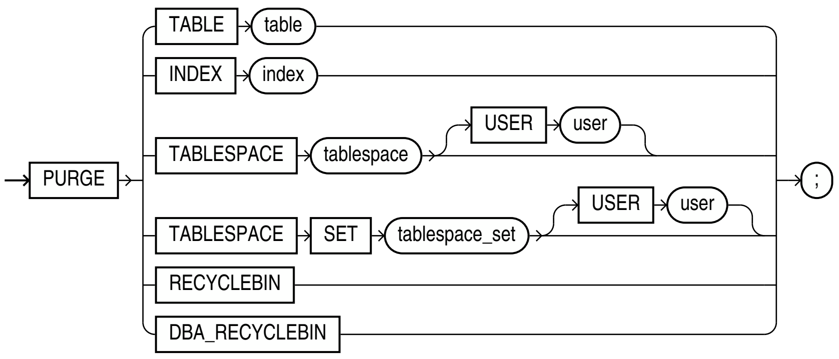
To purge an index, the index must reside in your own schema or you must have the `DROP ANY INDEX` system privilege, or you must have the `SYSDBA` system privilege.

To purge a tablespace or tablespace set, you must have the `DROP TABLESPACE` system privilege, or you must have the `SYSDBA` system privilege.

To purge a tablespace set, you must also be connected to a shard catalog database as an SDB user.
To perform the `PURGE DBA_RECYCLEBIN` operation, you must have the `SYSDBA` or `PURGE DBA_RECYCLEBIN` system privilege.

Syntax

***purge*::=**



Semantics

TABLE or INDEX

Specify the name of the table or index in the recycle bin that you want to purge. You can specify either the original user-specified name or the system-generated name Oracle Database assigned to the object when it was dropped.

- If you specify the user-specified name, and if the recycle bin contains more than one object of that name, then the database purges the object that has been in the recycle bin the longest.
- System-generated recycle bin object names are unique. Therefore, if you specify the system-generated name, then the database purges that specified object.

When the database purges a table, all table partitions, LOBs and LOB partitions, indexes, and other dependent objects of that table are also purged.

TABLESPACE or TABLESPACE SET

Use this clause to purge all the objects residing in the specified tablespace or tablespace set from the recycle bin.

USER *user*

Use this clause to reclaim space in a tablespace or tablespace set for a specified user. This operation is useful when a particular user is running low on disk quota for the specified tablespace or tablespace set.

RECYCLEBIN

Use this clause to purge the current user's recycle bin. Oracle Database will remove all objects from the user's recycle bin and release all space associated with objects in the recycle bin.

DBA_RECYCLEBIN

This clause is valid only if you have the `SYSDBA` or `PURGE DBA_RECYCLEBIN` system privilege. It lets you remove all objects from the system-wide recycle bin, and is equivalent to purging the recycle bin of every user. This operation is useful, for example, before backward migration.

Examples

Remove a File From Your Recycle Bin: Example

The following statement removes the table `test` from the recycle bin. If more than one version of test resides in the recycle bin, then Oracle Database removes the version that has been there the longest:

```
PURGE TABLE test;
```

To determine system-generated name of the table you want removed from your recycle bin, issue a `SELECT` statement on your recycle bin. Using that object name, you can remove the table by issuing a statement similar to the following statement. (The system-generated name will differ from the one shown in the example.)

```
PURGE TABLE RB$$33750$TABLE$0;
```

Remove the Contents of Your Recycle Bin: Example

To remove the entire contents of your recycle bin, issue the following statement:

```
PURGE RECYCLEBIN;
```

RENAME

Purpose

Note:

You cannot roll back a `RENAME` statement.

Use the `RENAME` statement to rename a table, view, sequence, private synonym, or property graph.

- Oracle Database automatically transfers integrity constraints, indexes, and grants on the old object to the new object.
- Oracle Database invalidates all objects that depend on the renamed object, such as views, synonyms, and stored procedures and functions that refer to a renamed table.

See Also:

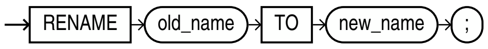
CREATE SYNONYM and DROP SYNONYM

Prerequisites

The object must be in your own schema.

Syntax

***rename*::=**



Semantics

old_name

Specify the name of an existing table, view, sequence, or private synonym.

new_name

Specify the new name to be given to the existing object. The new name must not already be used by another schema object in the same namespace and must follow the rules for naming schema objects.

Restrictions on Renaming Objects

Renaming objects is subject to the following restrictions:

- You cannot rename a public synonym. Instead, drop the public synonym and then re-create the public synonym with the new name.
- You cannot rename a type synonym that has any dependent tables or dependent valid user-defined object types.

See Also:

"Database Object Naming Rules "

Examples

Renaming a Database Object: Example

The following example uses a copy of the sample table `hr.departments`. To change the name of table `departments_new` to `emp_departments`, issue the following statement:

```
RENAME departments_new TO emp_departments;
```

You cannot use this statement directly to rename columns. However, you can rename a column using the `ALTER TABLE` ... *`rename_column_clause`*.

See Also:

rename_column_clause

Another way to rename a column is to use the RENAME statement together with the CREATE TABLE statement with AS *subquery*. This method is useful if you are changing the structure of a table rather than only renaming a column. The following statements re-create the sample table hr.job_history, renaming a column from department_id to dept_id:

```
CREATE TABLE temporary
   (employee_id, start_date, end_date, job_id, dept_id)
AS SELECT
     employee_id, start_date, end_date, job_id, department_id
FROM job_history;

DROP TABLE job_history;

RENAME temporary TO job_history;
```

Any integrity constraints defined on table job_history will be lost in the preceding example. You will have to redefine them on the new job_history table using an ALTER TABLE statement.

REVOKE

Purpose

Use the REVOKE statement to:

- Revoke system privileges from users and roles
- Revoke roles from users, roles, and program units.
- Revoke object privileges for a particular object from users and roles
- Revoke schema privileges from users and roles

Note on Oracle Automatic Storage Management

A user authenticated AS SYSASM can use this statement to revoke the system privileges SYSASM, SYSOPER, and SYSDBA from a user in the Oracle ASM password file of the current node.

Note on Editionable Objects

A REVOKE operation to revoke object privileges on an editionable object actualizes the object in the current edition. See *Oracle Database Development Guide* for more information about editions and editionable objects.

See Also:

- GRANT for information on granting system privileges and roles
- Table 18-4 for a listing of the object privileges for each type of object

Prerequisites

To revoke a **system privilege,** you must have been granted the privilege with the `ADMIN OPTION`. You can revoke any privilege if you have the `GRANT ANY PRIVILEGE` system privilege.

To revoke a **role from a user or another role**, you must have been directly granted the role with the `ADMIN OPTION` or you must have created the role. You can revoke any role if you have the `GRANT ANY ROLE` system privilege.

To revoke a **role from a program unit**, you must be the user `SYS` or you must be the schema owner of the program unit.

To revoke an **object privilege**, one of the following conditions must be met:

- You must previously have granted the object privilege to the user or role.
- You must have the `GRANT ANY OBJECT PRIVILEGE` system privilege.
- You must have the `GRANT ANY OBJECT PRIVILEGE` system privilege. In this case, you can revoke any object privilege that was granted by the object owner or on behalf of the owner by a user with the `GRANT ANY OBJECT PRIVILEGE`. However, you cannot revoke an object privilege that was granted by way of a `WITH GRANT OPTION` grant.
- You can revoke privileges on an object if you have the `GRANT ANY` object privilege. This does not apply to `SYS` objects. The `ANY` keyword in reference to a system privilege means that the user can perform the privilege on any objects owned by any user except for `SYS`.

See Also:

"Revoke Operations that Use GRANT ANY OBJECT PRIVILEGE: Example"

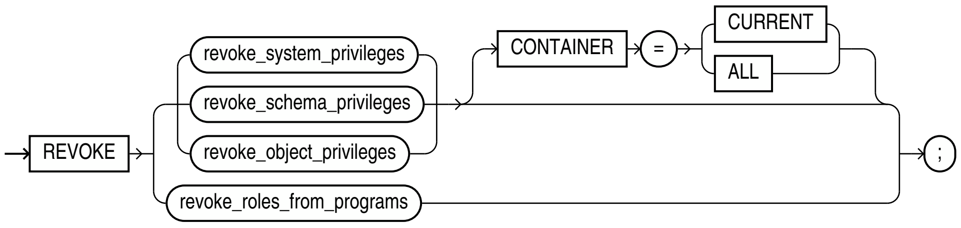
The `REVOKE` statement can revoke only privileges and roles that were previously granted directly with a `GRANT` statement. You cannot use this statement to revoke:

- Privileges or roles not granted to the revokee
- Roles or object privileges granted through the operating system
- Privileges or roles granted to the revokee through roles

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

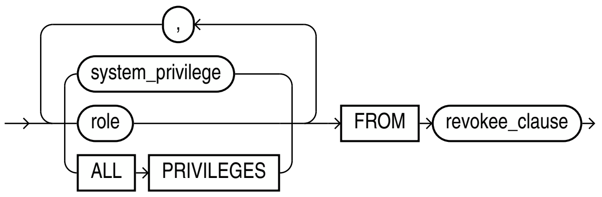
Syntax

***revoke*::=**



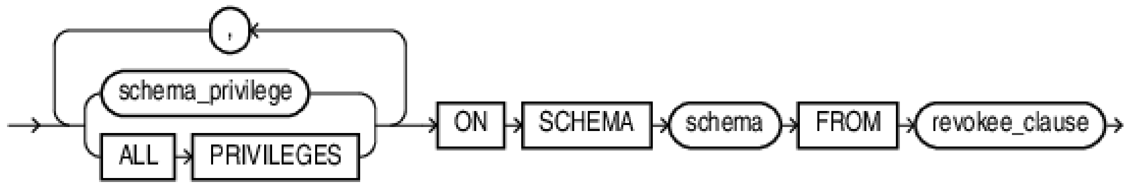
(*revoke_system_privileges::=*, *revoke_object_privileges::=*, *revoke_roles_from_programs::=*)

***revoke_system_privileges*::=**



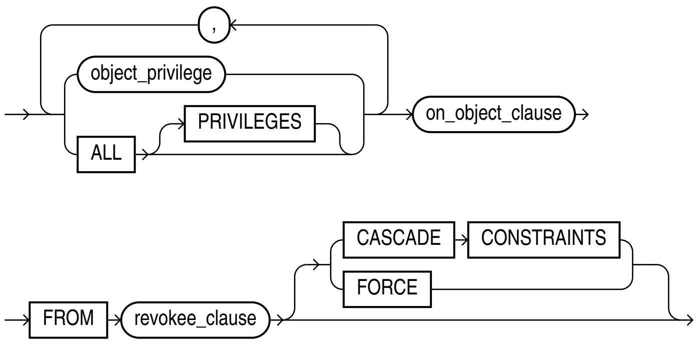
(*revokee_clause::=*)

***revoke_schema_privileges*::=**



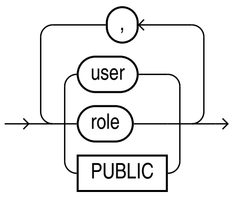
(*revokee_clause::=*)

***revoke_object_privileges*::=**

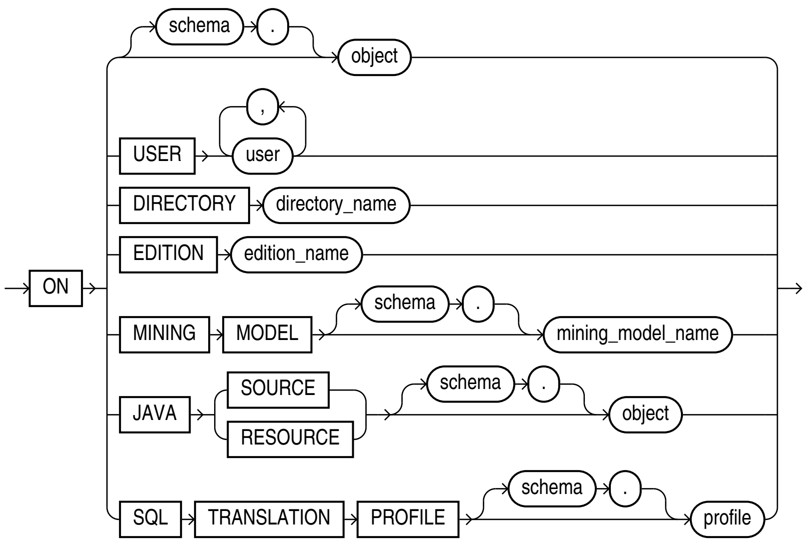


(*on_object_clause::=*, *revokee_clause::=*)

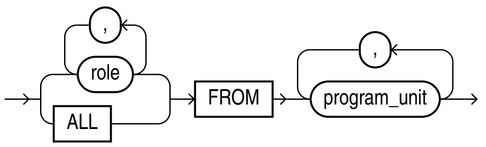
***revokee_clause*::=**



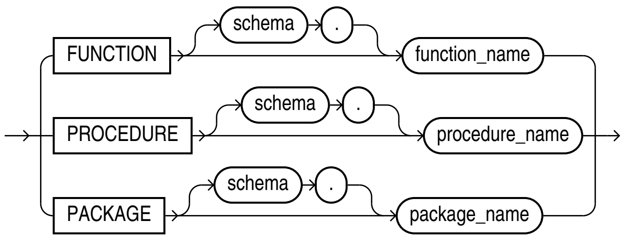
on_object_clause::=



revoke_roles_from_programs::=



program_unit::=



Semantics

revoke_system_privileges

Use these clauses to revoke system privileges.

system_privilege

Specify the system privilege to be revoked. Refer to Table 18-2 for a list of the system privileges.

If you revoke a system privilege from a **user**, then the database removes the privilege from the user's privilege domain. Effective immediately, the user cannot exercise the privilege.

If you revoke a system privilege from a **role**, then the database removes the privilege from the privilege domain of the role. Effective immediately, users with the role enabled cannot exercise the privilege. Also, other users who have been granted the role and subsequently enable the role cannot exercise the privilege.

See Also:

"Revoking a System Privilege from a User: Example" and "Revoking a System Privilege from a Role: Example"

If you revoke a system privilege from **PUBLIC**, then the database removes the privilege from the privilege domain of each user who has been granted the privilege through PUBLIC. Effective immediately, such users can no longer exercise the privilege. However, the privilege is not revoked from users who have been granted the privilege directly or through roles.

Oracle Database provides a shortcut for specifying all system privileges at once: Specify ALL PRIVILEGES to revoke all the system privileges listed in Table 18-2.

Restriction on Revoking System Privileges

A system privilege cannot appear more than once in the list of privileges to be revoked.

role

Specify the role to be revoked.

If you revoke a role from a **user**, then the database makes the role unavailable to the user. If the role is currently enabled for the user, then the user can continue to exercise the privileges in the role's privilege domain as long as it remains enabled. However, the user cannot subsequently enable the role.

If you revoke a role from another **role**, then the database removes the privilege domain of the revoked role from the privilege domain of the revokee role. Users who have been granted and have enabled the revokee role can continue to exercise the privileges in the privilege domain of the revoked role as long as the revokee role remains enabled. However, other users who have been granted the revokee role and subsequently enable it cannot exercise the privileges in the privilege domain of the revoked role.

See Also:

"Revoking a Role from a User: Example" and "Revoking a Role from a Role: Example"

If you revoke a role from **PUBLIC**, then the database makes the role unavailable to all users who have been granted the role through PUBLIC. Any user who has enabled the role can continue to exercise the privileges in its privilege domain as long as it remains enabled. However, users cannot subsequently enable the role. The role is not revoked from users who have been granted the role directly or through other roles.

Restriction on Revoking System Roles

A system role cannot appear more than once in the list of roles to be revoked. For information on the predefined roles, refer to *Oracle Database Security Guide*.

revokee_clause

Use the `revokee_clause` to specify the users or roles from which the system privilege, role, or object privilege is to be revoked.

PUBLIC

Specify `PUBLIC` to revoke the privileges or roles from all users.

revoke_schema_privileges

Use this clause to revoke schema privileges. You can revoke a schema level privilege from any user or any role if you are a user who :

- Owns the schema.
- Has a schema level privilege `WITH ADMIN OPTION`.
- Has the `GRANT ANY SCHEMA PRIVILEGE` system privilege. If you specify `ALL PRIVILEGES` , all the schema level privileges in Table 18-3 are revoked.

revoke_object_privileges

Use these clauses to revoke object privileges.

object_privilege

Specify the object privilege to be revoked. The object privileges, categorized by the type of object to which they apply, are described in Table 18-4.

Note:

Each privilege authorizes some operation. By revoking a privilege, you prevent the revokee from performing that operation. However, multiple users may grant the same privilege to the same user, role, or `PUBLIC`. To remove the privilege from the grantee's privilege domain, all grantors must revoke the privilege. If even one grantor does not revoke the privilege, then the grantee can still exercise the privilege by virtue of that grant.

If you revoke an object privilege from a **user**, then the database removes the privilege from the user's privilege domain. Effective immediately, the user cannot exercise the privilege.

If you revoke an object privilege from a user who has existing column level privileges granted , then those column level privileges will also be revoked.

- If that user has granted that privilege to other users or roles, then the database also revokes the privilege from those other users or roles.
- If that user's schema contains a procedure, function, or package that contains SQL statements that exercise the privilege, then the procedure, function, or package can no longer be executed.
- If that user's schema contains a view on that object, then the database invalidates the view.

- If you revoke the REFERENCES object privilege from a user who has exercised the privilege to define referential integrity constraints, then you must specify the CASCADE CONSTRAINTS clause.

If you revoke an object privilege from a **role**, then the database removes the privilege from the privilege domain of the role. Effective immediately, users with the role enabled cannot exercise the privilege. Other users who have been granted the role cannot exercise the privilege after enabling the role.

If you revoke an object privilege from PUBLIC, then the database removes the privilege from the privilege domain of each user who has been granted the privilege through PUBLIC. Effective immediately, all such users are restricted from exercising the privilege. However, the privilege is not revoked from users who have been granted the privilege directly or through roles.

ALL [PRIVILEGES]

Specify ALL to revoke all object privileges that you have granted to the revokee. (The keyword PRIVILEGES is provided for semantic clarity and is optional.)

If no privileges have been granted on the object, then the database takes no action and does not return an error.

Restriction on Revoking Object Privileges

A privilege cannot appear more than once in the list of privileges to be revoked. A user, a role, or PUBLIC cannot appear more than once in the FROM clause.

See Also:

"Revoking an Object Privilege from a User: Example", "Revoking Object Privileges from PUBLIC: Example", and "Revoking All Object Privileges from a User: Example"

CASCADE CONSTRAINTS

This clause is relevant only if you revoke the REFERENCES privilege or ALL [PRIVILEGES]. It drops any referential integrity constraints that the revokee has defined using the REFERENCES privilege, which might have been granted either explicitly or implicitly through a grant of ALL [PRIVILEGES].

See Also:

"Revoking an Object Privilege with CASCADE CONSTRAINTS: Example"

FORCE

Specify FORCE to revoke the EXECUTE object privilege on user-defined type objects with table or type dependencies. You must use FORCE to revoke the EXECUTE object privilege on user-defined type objects with table dependencies.

If you specify FORCE, then all privileges are revoked, all dependent objects are marked INVALID, data in dependent tables becomes inaccessible, and all dependent function-based indexes are marked UNUSABLE. Regranting the necessary type privilege will revalidate the table.

See Also:

Oracle Database Concepts for detailed information about type dependencies and user-defined object privileges

on_object_clause

The *on_object_clause* identifies the objects on which privileges are to be revoked.

object

Specify the object on which the object privileges are to be revoked. This object can be:

- A table, view, sequence, procedure, stored function, package, or materialized view
- A synonym for a table, view, sequence, procedure, stored function, package, materialized view, or user-defined type
- A library, indextype, or user-defined operator

If you do not qualify object with *schema*, then the database assumes the object is in your own schema.

See Also:

"Revoking an Object Privilege on a Sequence from a User: Example"

If you revoke the READ or SELECT object privilege on the containing table or materialized view of a materialized view, whether the privilege was granted with or without the GRANT OPTION, then the database invalidates the materialized view.

If you revoke the READ or SELECT object privilege on any of the master tables of a materialized view, whether the privilege was granted with or without the GRANT OPTION, then the database invalidates both the materialized view and its containing table or materialized view.

ON USER

Specify the database user you want to revoke privileges from.

See Also:

"Revoking an Object Privilege on a User from a User: Example"

ON DIRECTORY

Specify the name of the directory object on which privileges are to be revoked. You cannot qualify *directory_name* with a schema name.

See Also:

CREATE DIRECTORY and "Revoking an Object Privilege on a Directory from a User: Example"

ON EDITION

Specify the name of the edition on which the `USE` object privilege is to be revoked. You cannot qualify *`edition_name`* with a schema name.

ON MINING MODEL

Specify the name of the mining model on which privileges are to be revoked. If you do not qualify *`mining_model_name`* with *`schema`*, then the database assumes that the mining model is in your own schema.

ON JAVA SOURCE | RESOURCE

Specify the name of the Java source or resource schema object on which privileges are to be revoked. If you do not qualify *`object`* with *`schema`*, then the database assumes that the object is in your own schema.

ON SQL TRANSLATION PROFILE

Specify the name of the SQL translation profile on which privileges are to be revoked. If you do not qualify *`profile`* with *`schema`*, then the database assumes the profile is in your own schema.

revoke_roles_from_programs

Use this clause to revoke code based access control (CBAC) roles from program units.

role

Specify the role you want to revoke.

ALL

Specify `ALL` to revoke all roles that are granted to the program unit.

program_unit

Specify the program unit from which the role is to be revoked. You can specify a PL/SQL function, procedure, or package. If you do not specify *`schema`*, then Oracle Database assumes the function, procedure, or package is in your own schema.

See Also:

Oracle Database Security Guide for more information on revoking CBAC roles from program units

CONTAINER Clause

If the current container is a pluggable database (PDB):

- Specify `CONTAINER = CURRENT` to revoke a locally granted system privilege, object privilege, or role from a local user, common user, local role, or common role. The privilege or role is revoked from the user or role only in the current PDB. This clause does not revoke privileges granted with `CONTAINER = ALL`.

If the current container is the root:

- Specify `CONTAINER = CURRENT` to revoke a locally granted system privilege, object privilege, or role from a common user or common role. The privilege or role is revoked from the user or role only in the root. This clause does not revoke privileges granted with `CONTAINER = ALL`.
- Specify `CONTAINER = ALL` to revoke a commonly granted system privilege, object privilege on a common object, or role from a common user or common role. The privilege or role is revoked from the user or role across the entire CDB. This clause can revoke only a privilege or role granted with `CONTAINER = ALL` from the specified common user or common role. This clause does not revoke privileges granted locally with `CONTAINER = CURRENT`. However, any locally granted privileges that depend on the commonly granted privilege being revoked are also revoked.

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

Examples

Revoking a System Privilege from a User: Example

The following statement revokes the `DROP ANY TABLE` system privilege from the users `hr` and `oe`:

```
REVOKE DROP ANY TABLE
    FROM hr, oe;
```

The users `hr` and `oe` can no longer drop tables in schemas other than their own.

Revoking a Role from a User: Example

The following statement revokes the role `dw_manager` from the user `sh`:

```
REVOKE dw_manager
    FROM sh;
```

The user `sh` can no longer enable the `dw_manager` role.

Revoking a System Privilege from a Role: Example

The following statement revokes the `CREATE TABLESPACE` system privilege from the `dw_manager` role:

```
REVOKE CREATE TABLESPACE
   FROM dw_manager;
```

Enabling the `dw_manager` role no longer allows users to create tablespaces.

Revoking a Role from a Role: Example

To revoke the role `dw_user` from the role `dw_manager`, issue the following statement:

```
REVOKE dw_user
  FROM dw_manager;
```

The `dw_user` role privileges are no longer granted to `dw_manager`.

Revoking an Object Privilege from a User: Example

You can grant `DELETE`, `INSERT`, `READ`, `SELECT`, and `UPDATE` privileges on the table `orders` to the user `hr` with the following statement:

```
GRANT ALL
   ON orders TO hr;
```

To revoke the `DELETE` privilege on `orders` from `hr`, issue the following statement:

```
REVOKE DELETE
   ON orders FROM hr;
```

Revoking All Object Privileges from a User: Example

To revoke the remaining privileges on `orders` that you granted to `hr`, issue the following statement:

```
REVOKE ALL
   ON orders FROM hr;
```

Revoking Object Privileges from PUBLIC: Example

You can grant `SELECT` and `UPDATE` privileges on the view `emp_details_view` to all users by granting the privileges to the role `PUBLIC`:

```
GRANT SELECT, UPDATE
    ON emp_details_view TO public;
```

The following statement revokes `UPDATE` privilege on `emp_details_view` from all users:

```
REVOKE UPDATE
    ON emp_details_view FROM public;
```

Users can no longer update the `emp_details_view` view, although users can still query it. However, if you have also granted the `UPDATE` privilege on `emp_details_view` to any users, either directly or through roles, then these users retain the privilege.

Revoking an Object Privilege on a User from a User: Example

You can grant the user `hr` the `INHERIT PRIVILEGES` privilege on user `sh` with the following statement:

```
GRANT INHERIT PRIVILEGES ON USER sh TO hr;
```

To revoke the `INHERIT PRIVILEGES` privilege on user `sh` from user `hr`, issue the following statement:

```
REVOKE INHERIT PRIVILEGES ON USER sh FROM hr;
```

Revoking an Object Privilege on a Sequence from a User: Example

You can grant the user `oe` the `SELECT` privilege on the `departments_seq` sequence in the schema `hr` with the following statement:

```
GRANT SELECT
    ON hr.departments_seq TO oe;
```

To revoke the `SELECT` privilege on `departments_seq` from `oe`, issue the following statement:

```
REVOKE SELECT
    ON hr.departments_seq FROM oe;
```

However, if the user `hr` has also granted `SELECT` privilege on `departments` to `sh`, then `sh` can still use `departments` by virtue of `hr`'s grant.

Revoking an Object Privilege with CASCADE CONSTRAINTS: Example

You can grant to `oe` the privileges `REFERENCES` and `UPDATE` on the `employees` table in the schema `hr` with the following statement:

```
GRANT REFERENCES, UPDATE
    ON hr.employees TO oe;
```

The user `oe` can exercise the `REFERENCES` privilege to define a constraint in his or her own `dependent` table that refers to the `employees` table in the schema `hr`:

```
CREATE TABLE dependent
(dependno   NUMBER,
 dependname VARCHAR2(10),
 employee   NUMBER
    CONSTRAINT in_emp REFERENCES hr.employees(employee_id) );
```

You can revoke the `REFERENCES` privilege on `hr.employees` from `oe` by issuing the following statement that contains the `CASCADE CONSTRAINTS` clause:

```
REVOKE REFERENCES
    ON hr.employees
    FROM oe
    CASCADE CONSTRAINTS;
```

Revoking `oe`'s `REFERENCES` privilege on `hr.employees` causes Oracle Database to drop the `in_emp` constraint, because `oe` required the privilege to define the constraint.

However, if `oe` has also been granted the `REFERENCES` privilege on `hr.employees` by a user other than you, then the database does not drop the constraint. `oe` still has the privilege necessary for the constraint by virtue of the other user's grant.

Revoking an Object Privilege on a Directory from a User: Example

You can revoke the `READ` object privilege on directory `bfile_dir` from `hr` by issuing the following statement:

```
REVOKE READ ON DIRECTORY bfile_dir FROM hr;
```

Revoke Operations that Use GRANT ANY OBJECT PRIVILEGE: Example

Suppose that the database administrator has granted `GRANT ANY OBJECT PRIVILEGE` to user `sh`. Now suppose that user `hr` grants the update privilege on the `employees` table to `oe`:

```
CONNECT hr
GRANT UPDATE ON employees TO oe WITH GRANT OPTION;
```

This grant gives user `oe` the right to pass the object privilege along to another user:

```
CONNECT oe
GRANT UPDATE ON hr.employees TO pm;
```

User `sh`, who has the `GRANT ANY OBJECT PRIVILEGE`, can now act on behalf of user `hr` and revoke the update privilege from user `oe`, because `oe` was granted the privilege by `hr`:

```
CONNECT sh
REVOKE UPDATE ON hr.employees FROM oe;
```

User `sh` cannot revoke the update privilege from user `pm` explicitly, because `pm` received the grant neither from the object owner (`hr`), nor from `sh`, nor from another user with `GRANT ANY OBJECT PRIVILEGE`, but from user `oe`. However, the preceding statement cascades, removing all privileges that depend on the one revoked. Therefore the object privilege is implicitly revoked from `pm` as well.

ROLLBACK

Purpose

Use the `ROLLBACK` statement to undo work done in the current transaction or to manually undo the work done by an in-doubt distributed transaction.

Note:

Oracle recommends that you explicitly end transactions in application programs using either a `COMMIT` or `ROLLBACK` statement. If you do not explicitly commit the transaction and the program terminates abnormally, then Oracle Database rolls back the last uncommitted transaction.

See Also:

- *Oracle Database Concepts* for information on transactions
- *Oracle Database Heterogeneous Connectivity User's Guide* for information on distributed transactions
- SET TRANSACTION for information on setting characteristics of the current transaction
- COMMIT and SAVEPOINT

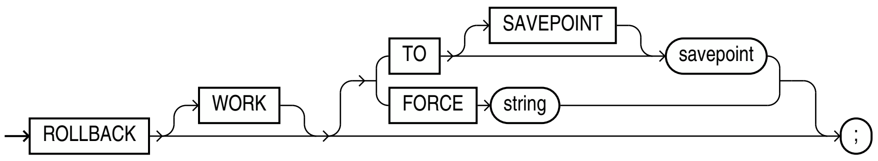
Prerequisites

To roll back your current transaction, no privileges are necessary.

To manually roll back an in-doubt distributed transaction that you originally committed, you must have the `FORCE TRANSACTION` system privilege. To manually roll back an in-doubt distributed transaction originally committed by another user, you must have the `FORCE ANY TRANSACTION` system privilege.

Syntax

***rollback*::=**



Semantics

WORK

The keyword `WORK` is optional and is provided for SQL standard compatibility.

TO SAVEPOINT Clause

Specify the savepoint to which you want to roll back the current transaction. If you omit this clause, then the `ROLLBACK` statement rolls back the entire transaction.

Using `ROLLBACK` without the `TO SAVEPOINT` clause performs the following operations:

- Ends the transaction
- Undoes all changes in the current transaction
- Erases all savepoints in the transaction
- Releases any transaction locks

> **See Also:**
>
> SAVEPOINT

Using `ROLLBACK` with the `TO SAVEPOINT` clause performs the following operations:

- Rolls back just the portion of the transaction after the savepoint. It does not end the transaction.
- Erases all savepoints created after that savepoint. The named savepoint is retained, so you can roll back to the same savepoint multiple times. Prior savepoints are also retained.
- Releases all table and row locks acquired since the savepoint. Other transactions that have requested access to rows locked after the savepoint must continue to wait until the transaction is committed or rolled back. Other transactions that have not already requested the rows can request and access the rows immediately.

Restriction on In-doubt Transactions

You cannot manually roll back an in-doubt transaction to a savepoint.

FORCE Clause

Specify `FORCE` to manually roll back an in-doubt distributed transaction. The transaction is identified by the *`string`* containing its local or global transaction ID. To find the IDs of such transactions, query the data dictionary view `DBA_2PC_PENDING`.

A `ROLLBACK` statement with a `FORCE` clause rolls back only the specified transaction. Such a statement does not affect your current transaction.

> **See Also:**
>
> *Oracle Database Administrator's Guide* for more information on distributed transactions and rolling back in-doubt transactions

Examples

Rolling Back Transactions: Examples

The following statement rolls back your entire current transaction:

```
ROLLBACK;
```

The following statement rolls back your current transaction to savepoint `banda_sal`:

```
ROLLBACK TO SAVEPOINT banda_sal;
```

See "Creating Savepoints: Example" for a full version of the preceding example.

The following statement manually rolls back an in-doubt distributed transaction:

```
ROLLBACK WORK
    FORCE '25.32.87';
```

SAVEPOINT

Purpose

Use the `SAVEPOINT` statement to create a name for a system change number (SCN), to which you can later roll back.

See Also:

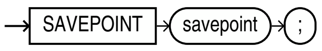
- *Oracle Database Concepts* for information on savepoints.
- ROLLBACK for information on rolling back transactions
- SET TRANSACTION for information on setting characteristics of the current transaction

Prerequisites

None.

Syntax

***savepoint*::=**



Semantics

savepoint

Specify the name of the savepoint to be created.

Savepoint names must be distinct within a given transaction. If you create a second savepoint with the same identifier as an earlier savepoint, then the earlier savepoint is erased. After a savepoint has been created, you can either continue processing, commit your work, roll back the entire transaction, or roll back to the savepoint.

Examples

Creating Savepoints: Example

To update the salary for `Banda` and `Greene` in the sample table `hr.employees`, check that the total department salary does not exceed 314,000, then reenter the salary for `Greene`:

```
UPDATE employees
    SET salary = 7000
    WHERE last_name = 'Banda';
SAVEPOINT banda_sal;

UPDATE employees
    SET salary = 12000
    WHERE last_name = 'Greene';
SAVEPOINT greene_sal;

SELECT SUM(salary) FROM employees;

ROLLBACK TO SAVEPOINT banda_sal;

UPDATE employees
    SET salary = 11000
    WHERE last_name = 'Greene';

COMMIT;
```

SELECT

Purpose

Use a `SELECT` statement or subquery to retrieve data from one or more tables, object tables, views, object views, materialized views, analytic views, or hierarchies.

If part or all of the result of a `SELECT` statement is equivalent to an existing materialized view, then Oracle Database may use the materialized view in place of one or more tables specified in the `SELECT` statement. This substitution is called **query rewrite**. It takes place only if cost optimization is enabled and the `QUERY_REWRITE_ENABLED` parameter is set to `TRUE`. To determine whether query rewrite has occurred, use the `EXPLAIN PLAN` statement.

See Also:

- SQL Queries and Subqueries for general information on queries and subqueries
- *Oracle Database Data Warehousing Guide* for more information on materialized views, query rewrite, and analytic views and hierarchies
- If you are querying JSON data see *Query JSON Data*
- If you are querying XML data see *Querying XML Content Stored in Oracle XML DB*
- EXPLAIN PLAN

Prerequisites

For you to select data from a table, materialized view, analytic view, or hierarchy, the object must be in your own schema or you must have the `READ` or `SELECT` privilege on the table, materialized view, analytic view, or hierarchy.

For you to select rows from the base tables of a view:

- The object must be in your own schema or you must have the `READ` or `SELECT` privilege on it, and
- Whoever owns the schema containing the object must have the `READ` or `SELECT` privilege on the base tables.

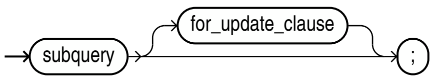
The `READ ANY TABLE` or `SELECT ANY TABLE` system privilege also allows you to select data from any table, materialized view, analytic view, or hierarchy, or the base table of any materialized view, analytic view, or hierarchy.

To specify the `FOR UPDATE` clause, the preceding prerequisites apply with the following exception: The `READ` and `READ ANY TABLE` privileges, where mentioned, do not allow you to specify the `FOR UPDATE` clause.

To issue an Oracle Flashback Query using the *`flashback_query_clause`*, you must have the `READ` or `SELECT` privilege on the objects in the select list. In addition, either you must have `FLASHBACK` object privilege on the objects in the select list, or you must have `FLASHBACK ANY TABLE` system privilege.

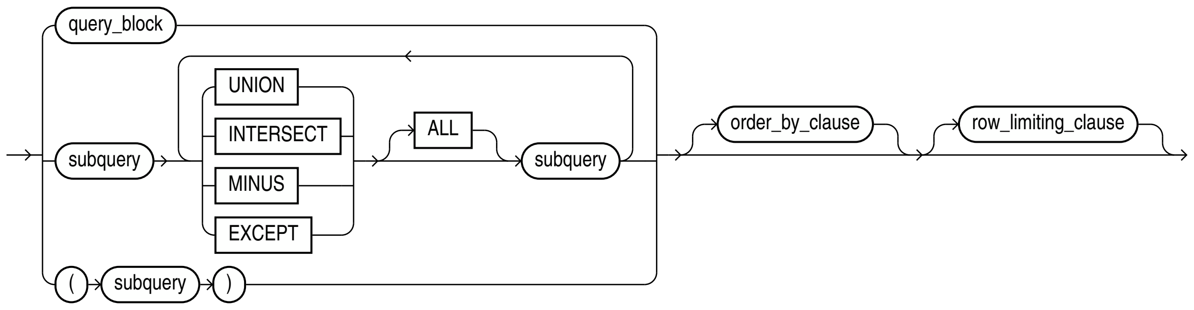
Syntax

***select*::=**



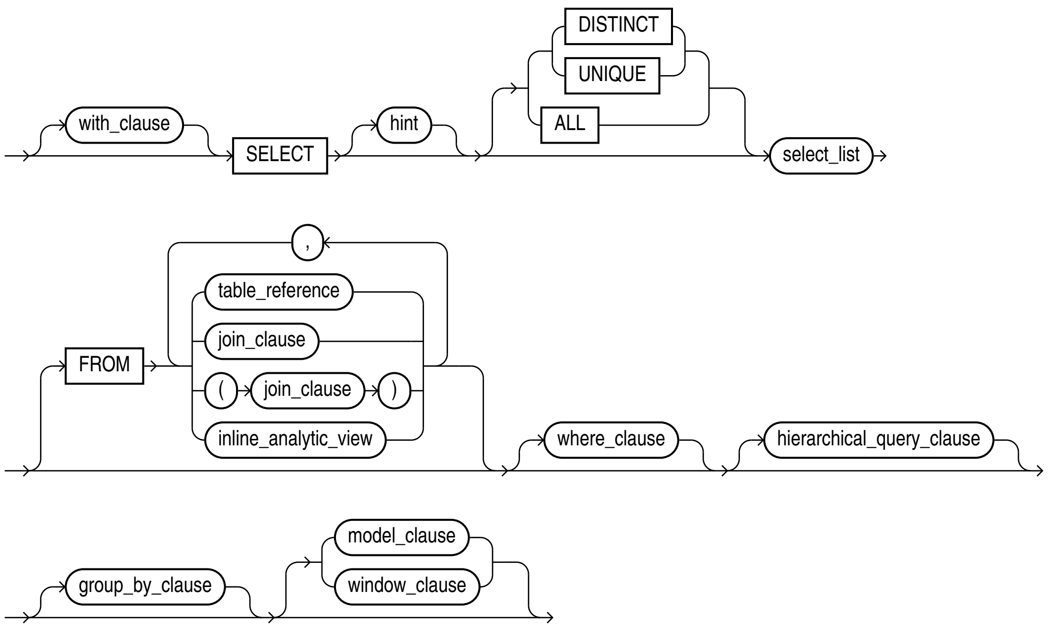
(*subquery*::=, *for_update_clause*::=)

***subquery*::=**



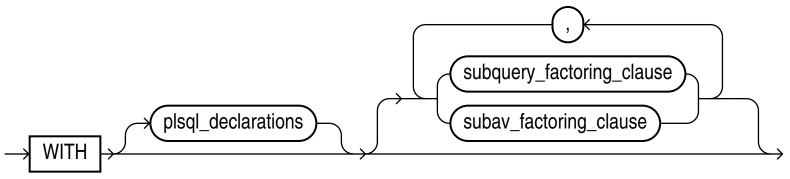
(*query_block*::=, *order_by_clause*::=, *row_limiting_clause*::=)

***query_block*::=**



(*with_clause*::=, *select_list*::=, *table_reference*::=, *join_clause*::=, *inline_analytic_view*, *where_clause*::=, *hierarchical_query_clause*::=, *group_by_clause*::=, *model_clause*::= , *window_clause*::=)

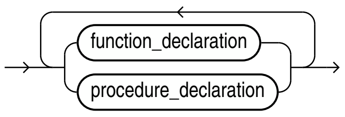
***with_clause*::=**



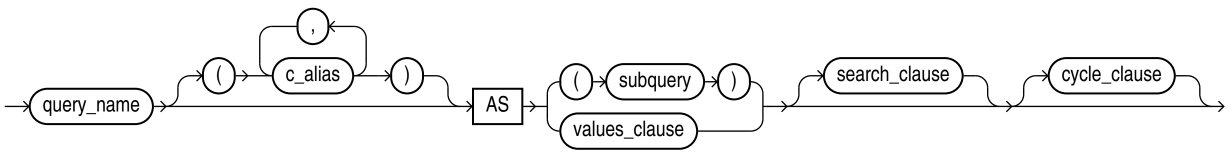
Note:

You cannot specify only the `WITH` keyword. You must specify at least one of the clauses *`plsql_declarations`*, *`subquery_factoring_clause`*, or *`subav_factoring_clause`*.

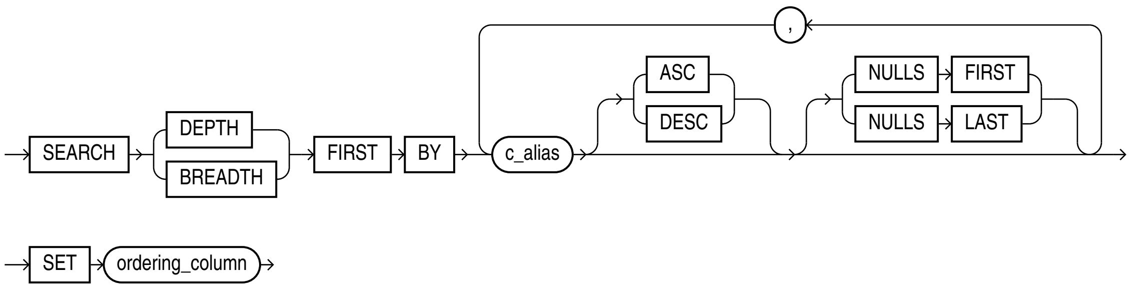
***plsql_declarations*::=**



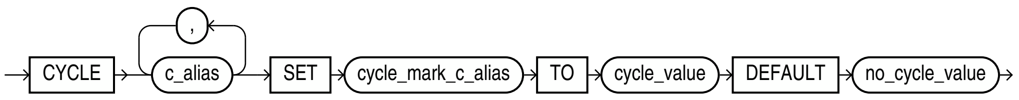
***subquery_factoring_clause*::=**



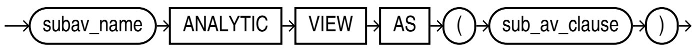
***search_clause*::=**



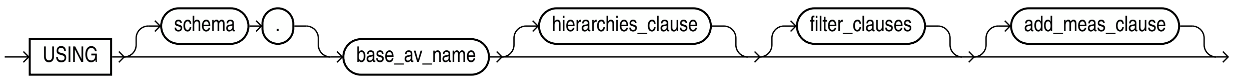
***cycle_clause*::=**



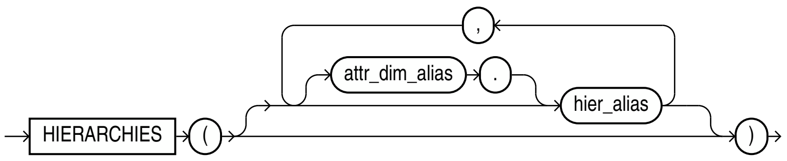
***subav_factoring_clause*::=**



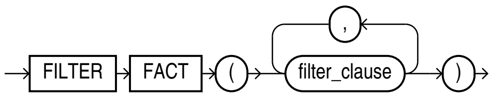
***sub_av_clause*::=**



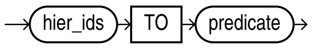
***hierarchies_clause*::=**



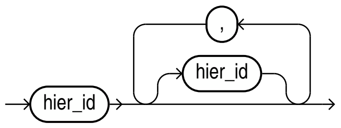
filter_clauses::=



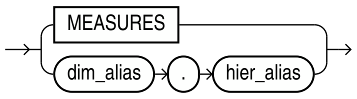
filter_clause::=



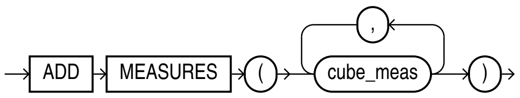
hier_ids::=



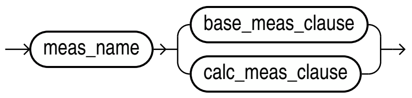
hier_id::=



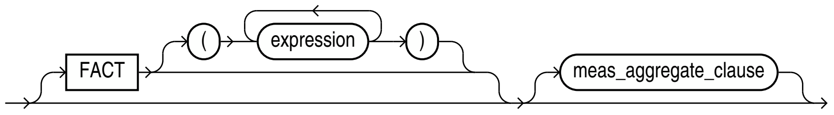
add_meas_clause::=



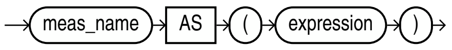
cube_meas::=



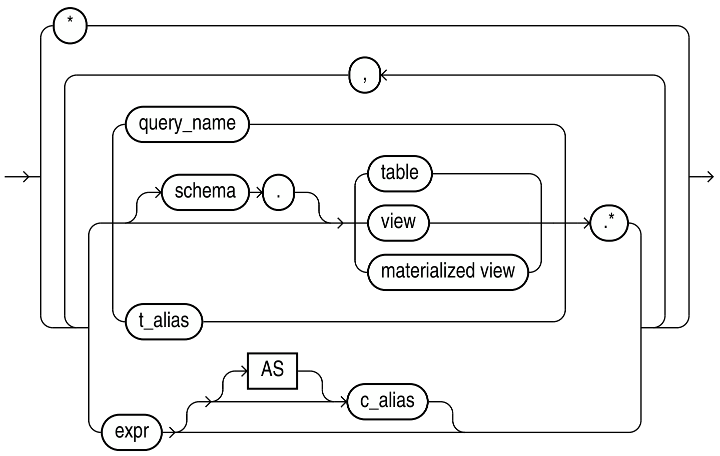
base_meas_clause::=



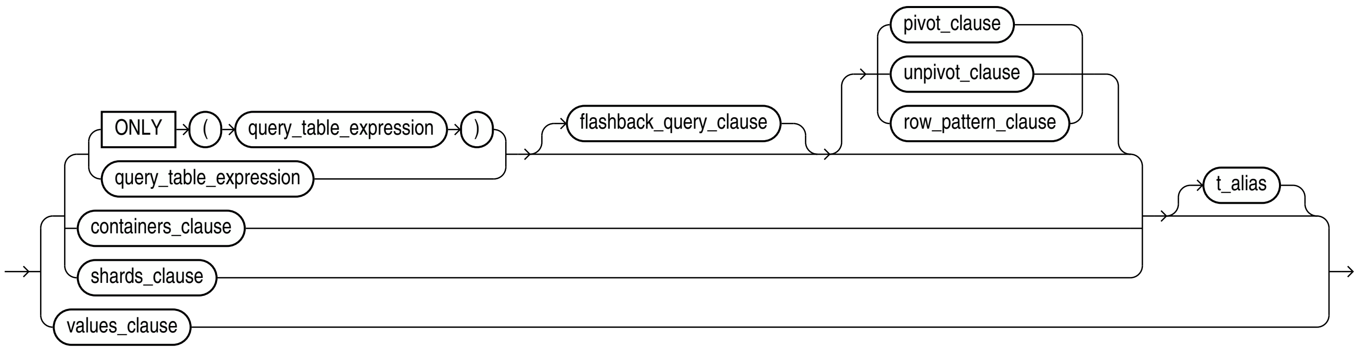
calc_meas_clause::=



select_list::=

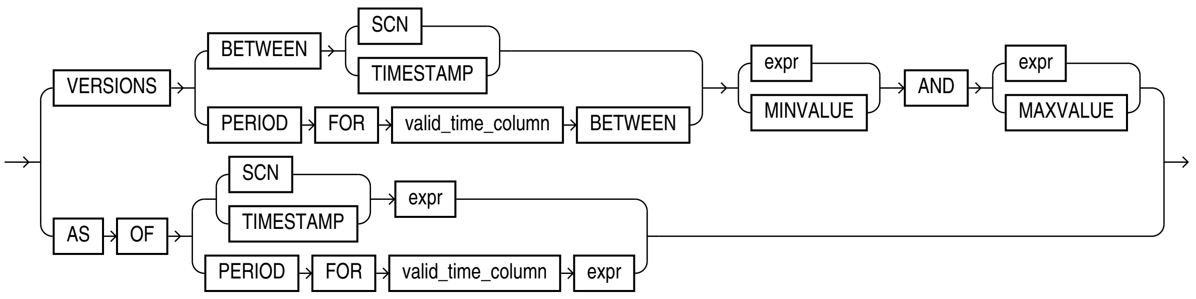


table_reference::=

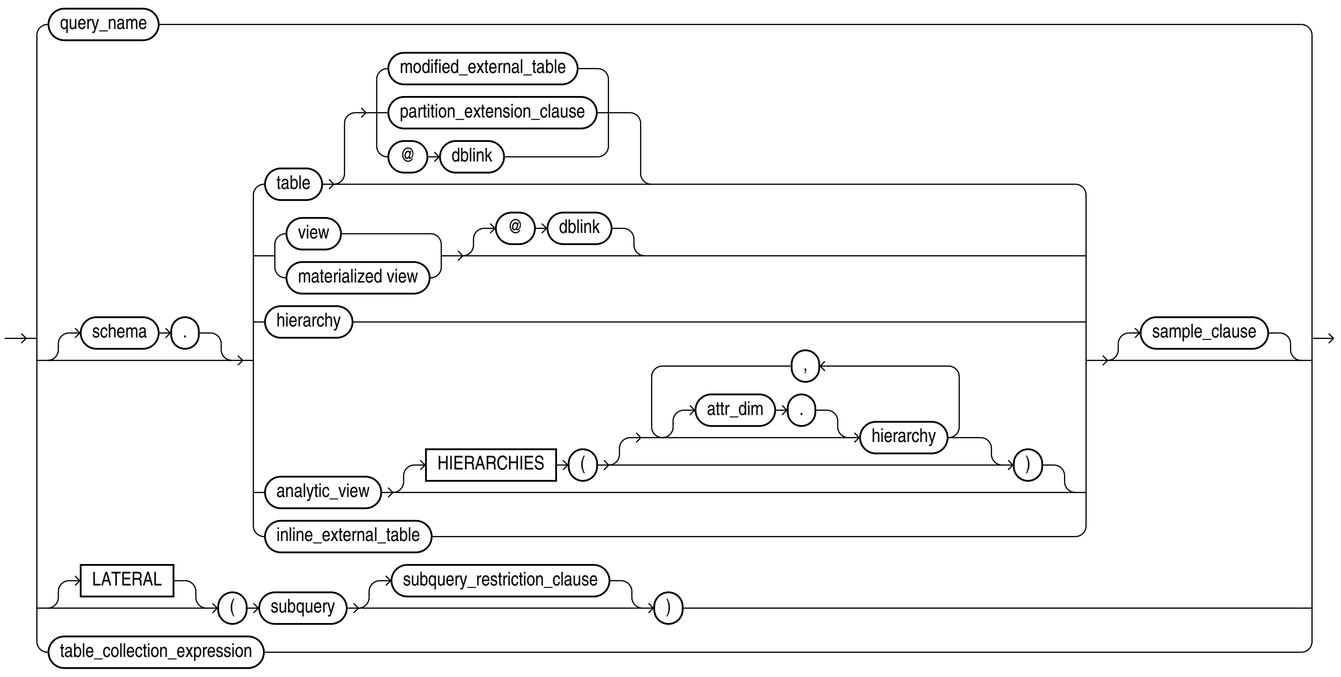


(*query_table_expression*::=, *flashback_query_clause*::=, *pivot_clause*::=, *unpivot_clause*::=, *row_pattern_clause*::=, *containers_clause*::=, shards_clause::=, values_clause::=)

flashback_query_clause::=

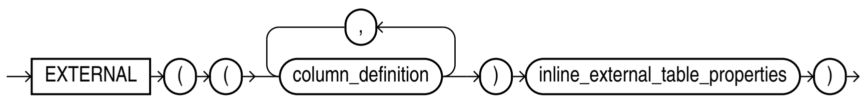


query_table_expression::=

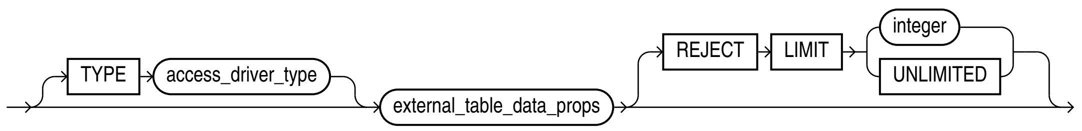


(*analytic_view*, *hierarchy*, *subquery_restriction_clause*::=, *table_collection_expression*::=)

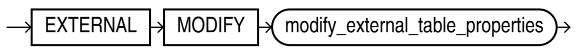
inline_external_table::=



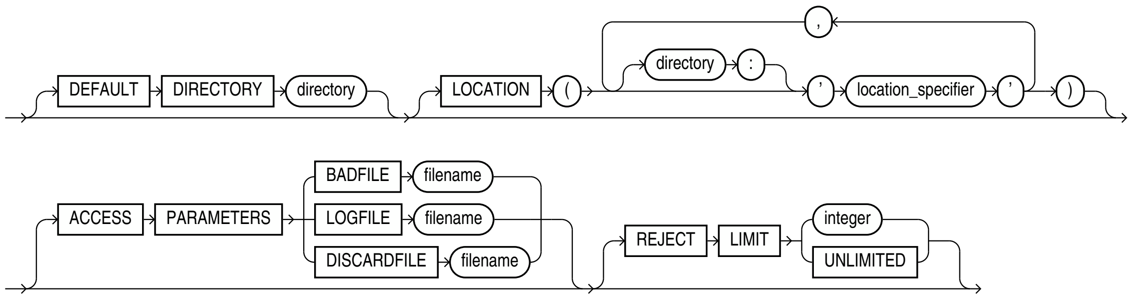
inline_external_table_properties::=



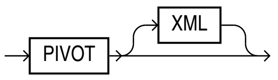
modified_external_table::=

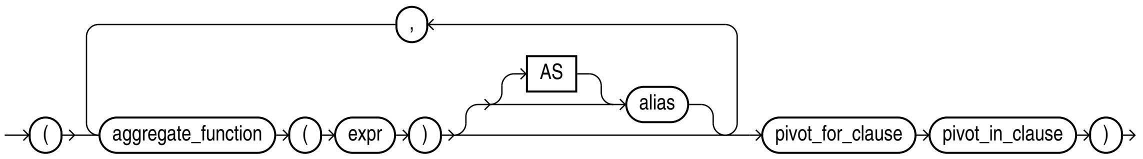


***modify_external_table_properties*::=**

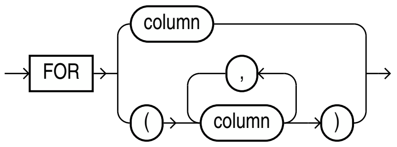


***pivot_clause*::=**

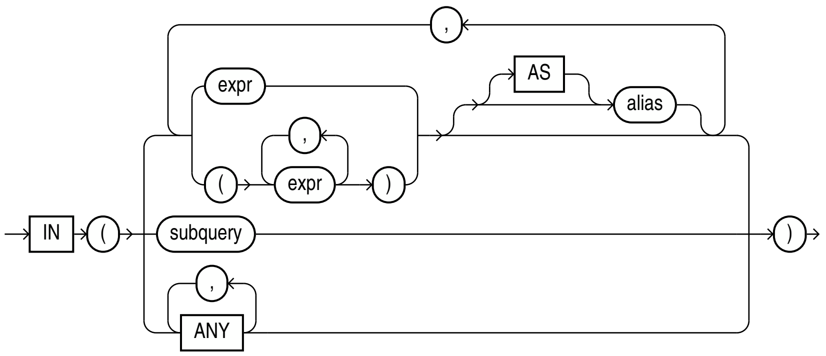




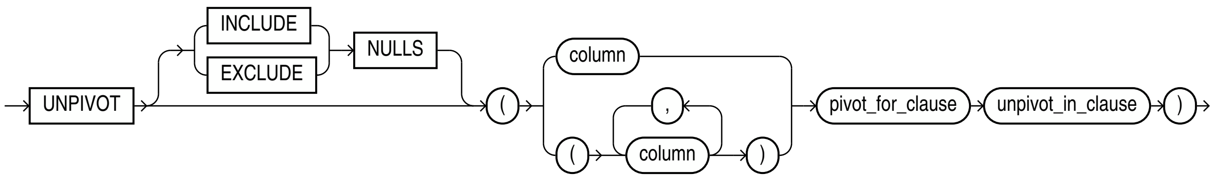
***pivot_for_clause*::=**



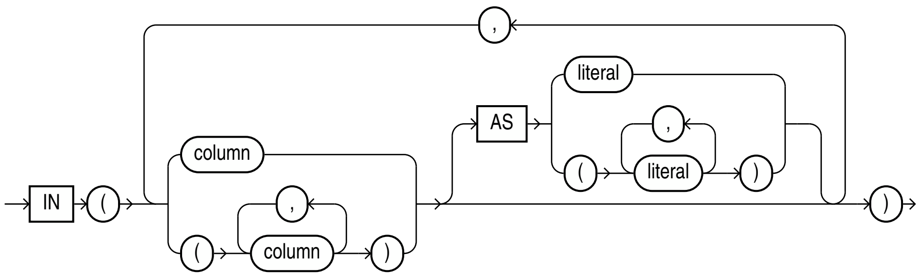
***pivot_in_clause*::=**



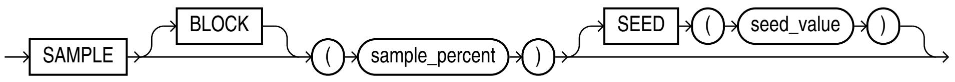
***unpivot_clause*::=**



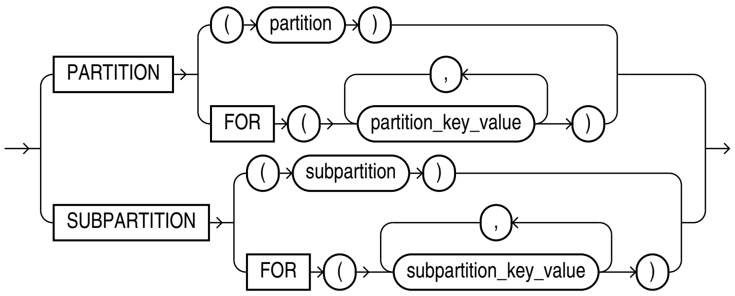
***unpivot_in_clause*::=**



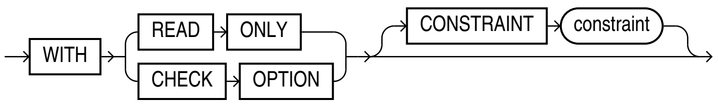
***sample_clause*::=**



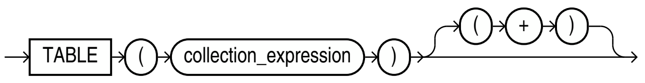
***partition_extension_clause*::=**



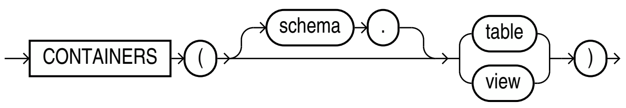
***subquery_restriction_clause*::=**



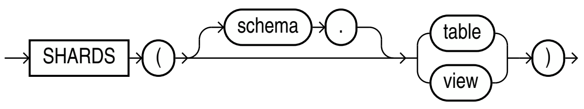
***table_collection_expression*::=**



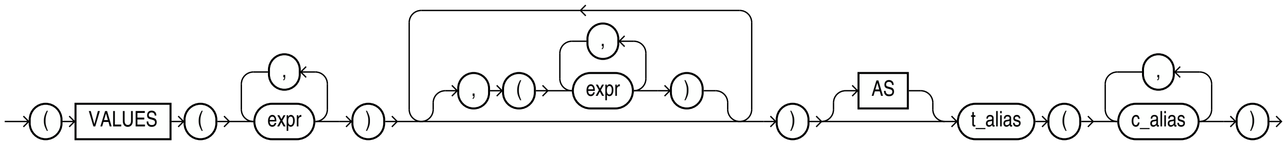
***containers_clause*::=**



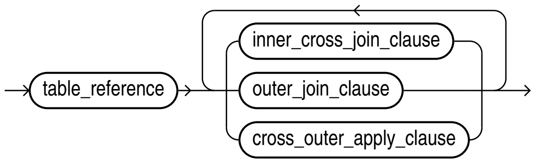
***shards_clause*::=**



***values_clause*::=**

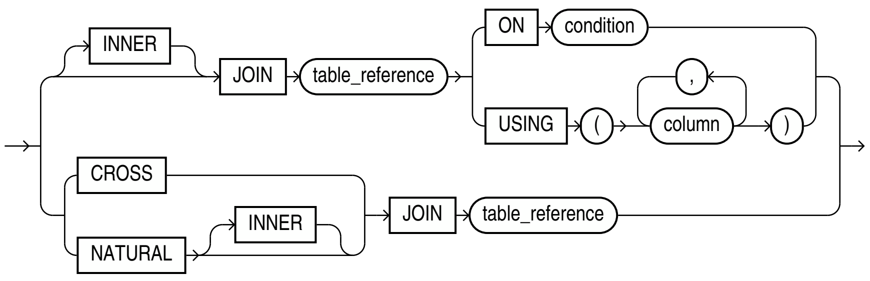


***join_clause*::=**



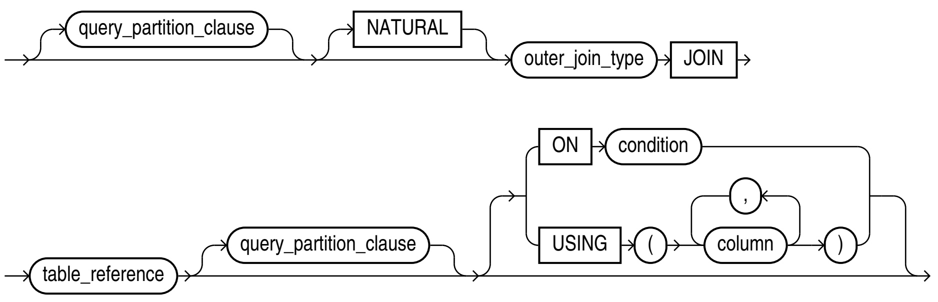
(*inner_cross_join_clause*::=, *outer_join_clause*::=, *cross_outer_apply_clause*::=)

***inner_cross_join_clause*::=**



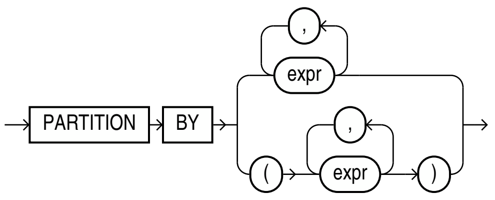
(*table_reference*::=)

***outer_join_clause*::=**

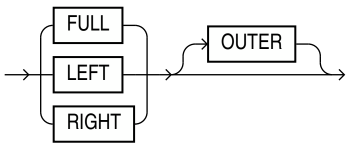


(*query_partition_clause*::=, *outer_join_type*::=, *table_reference*::=)

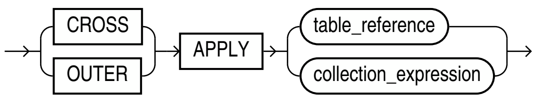
***query_partition_clause*::=**



***outer_join_type*::=**

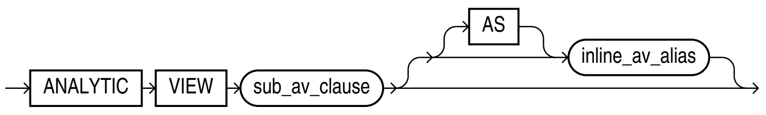


***cross_outer_apply_clause*::=**



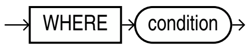
(*table_reference*::=, *query_partition_clause*::=)

inline_analytic_view

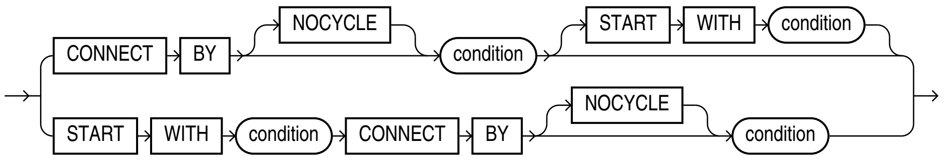


(*sub_av_clause::=*)

***where_clause*::=**

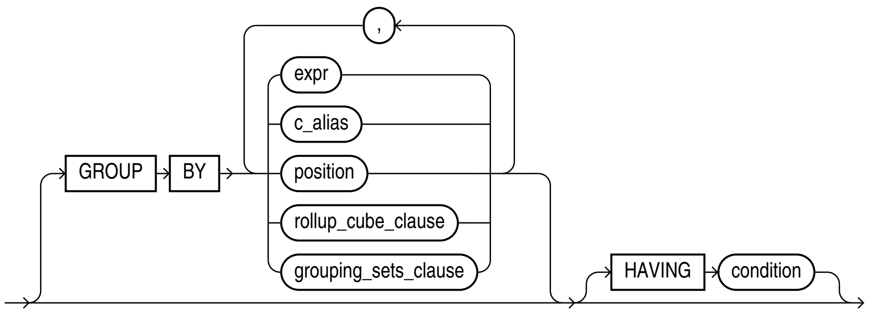


***hierarchical_query_clause*::=**



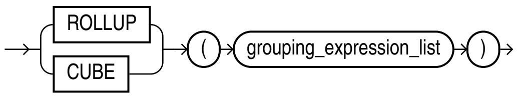
(`condition` can be any condition as described in Conditions)

***group_by_clause*::=**



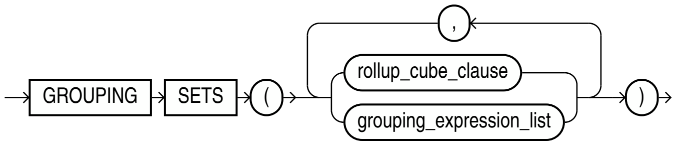
(*rollup_cube_clause::=*, *grouping_sets_clause::=*)

***rollup_cube_clause*::=**



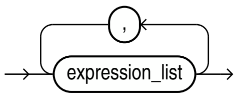
(*grouping_expression_list::=*)

***grouping_sets_clause*::=**

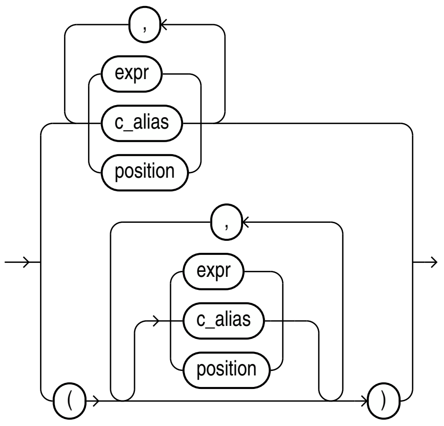


(*rollup_cube_clause*::=, *grouping_expression_list*::=)

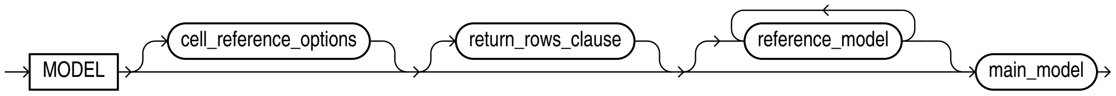
***grouping_expression_list*::=**



***expression_list*::=**

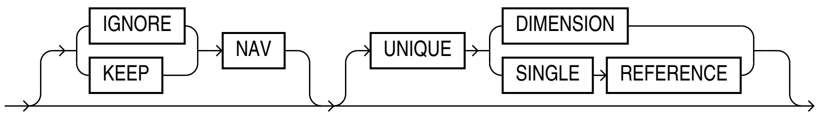


***model_clause*::=**

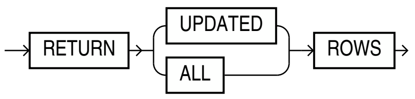


(*cell_reference_options*::=, *return_rows_clause*::=, *reference_model*::=, *main_model*::=)

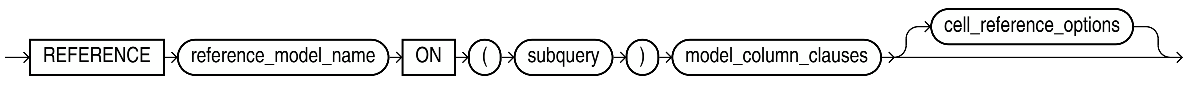
***cell_reference_options*::=**



***return_rows_clause*::=**

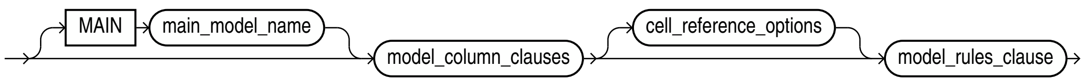


***reference_model*::=**



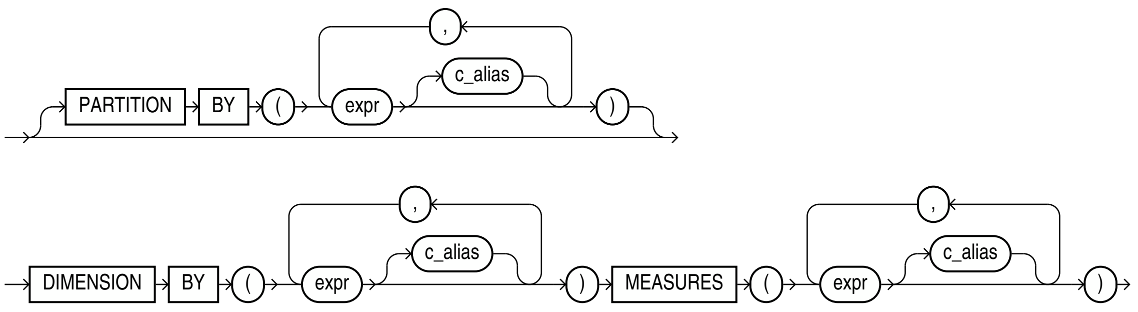
(*model_column_clauses*::=, *cell_reference_options*::=)

***main_model*::=**

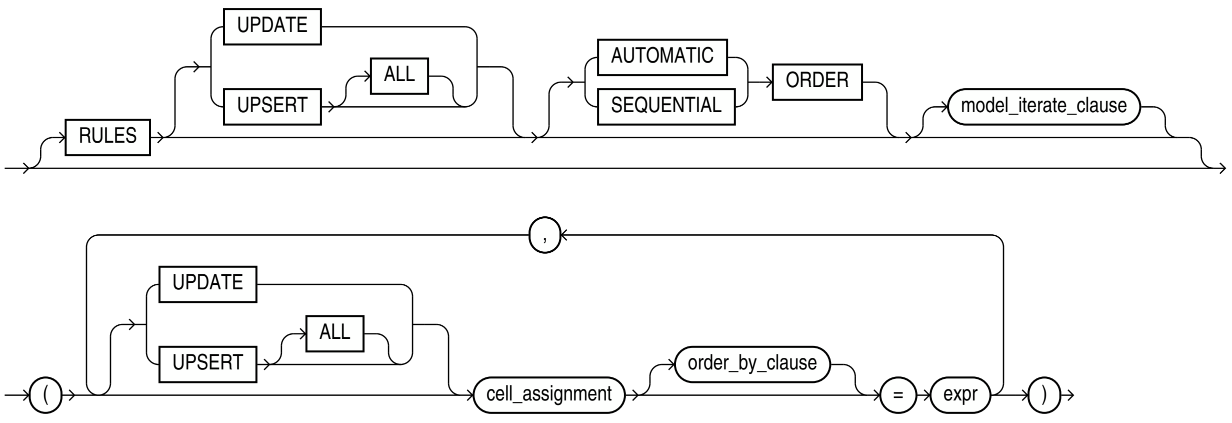


(*model_column_clauses*::=, *cell_reference_options*::=, *model_rules_clause*::=)

***model_column_clauses*::=**

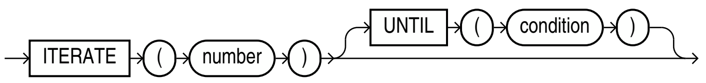


***model_rules_clause*::=**

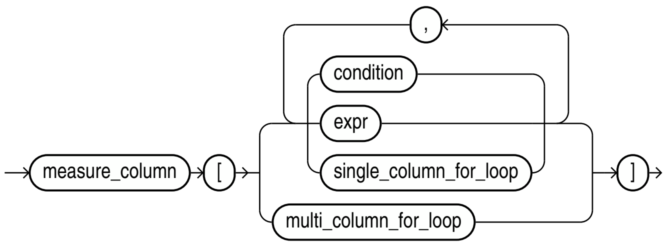


(*model_iterate_clause*::=, *cell_assignment*::=, *order_by_clause*::=)

***model_iterate_clause*::=**

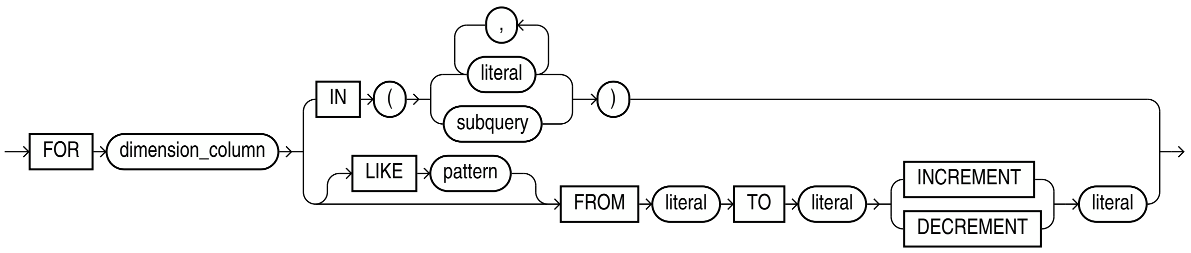


***cell_assignment*::=**

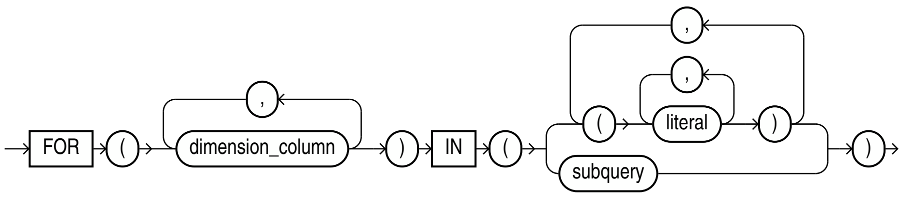


(*single_column_for_loop*::=, *multi_column_for_loop*::=)

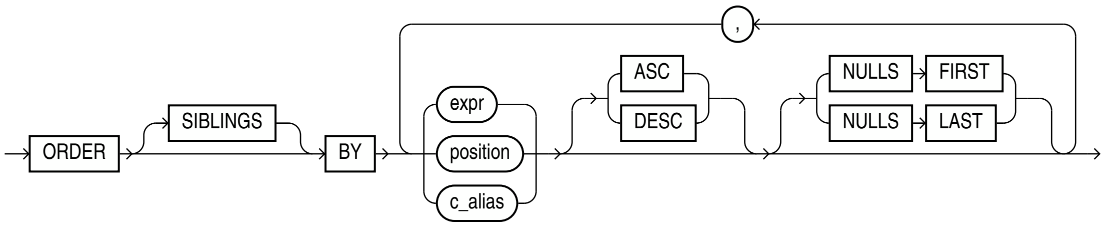
***single_column_for_loop*::=**



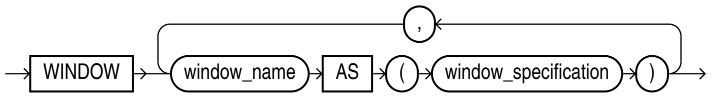
***multi_column_for_loop*::=**



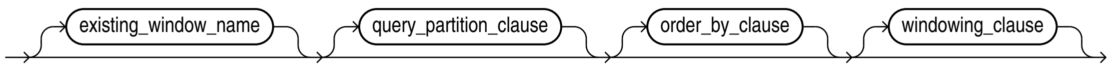
***order_by_clause*::=**



***window_clause*::=**

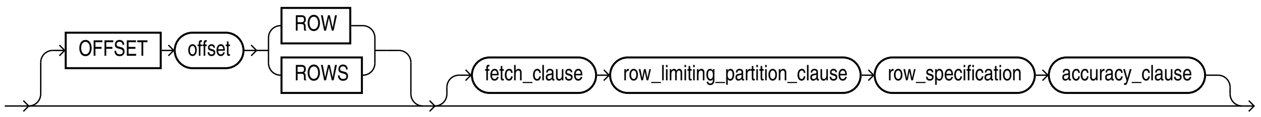


***window_specification*::=**



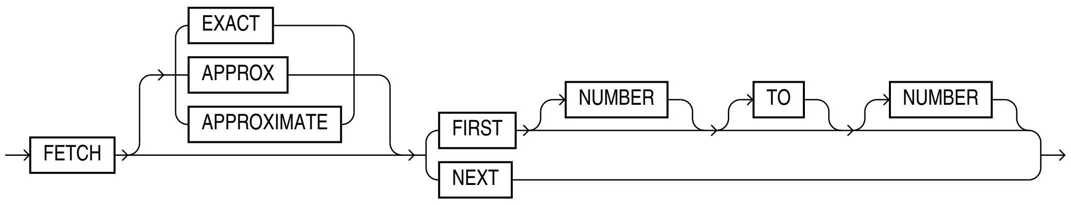
query_partition_clause::=, *order_by_clause*::=, *windowing_clause*

***row_limiting_clause*::=**

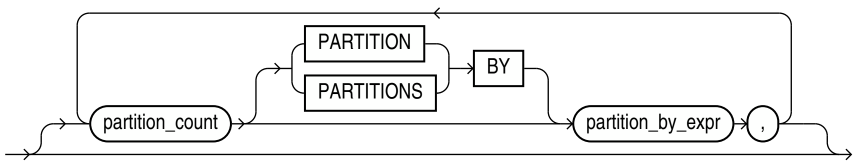


(*fetch_clause*::=, *row_limiting_partition_clause*::=, *row_specification*::=, *accuracy_clause*::=)

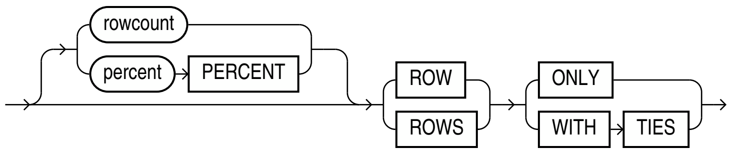
***fetch_clause*::=**



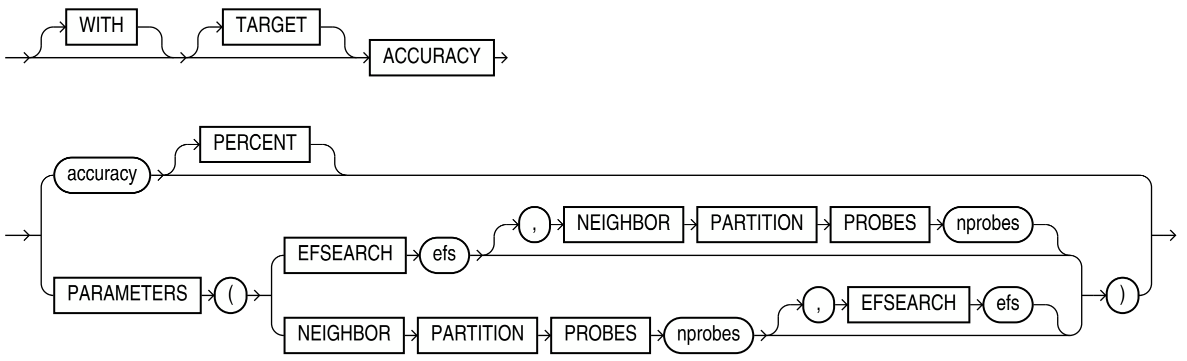
***row_limiting_partition_clause*::=**



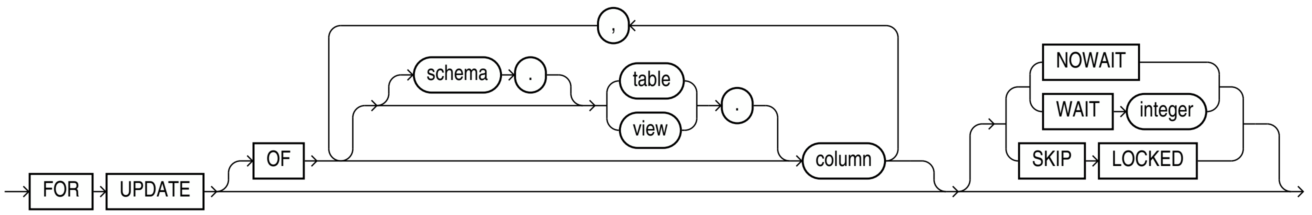
***row_specification*::=**



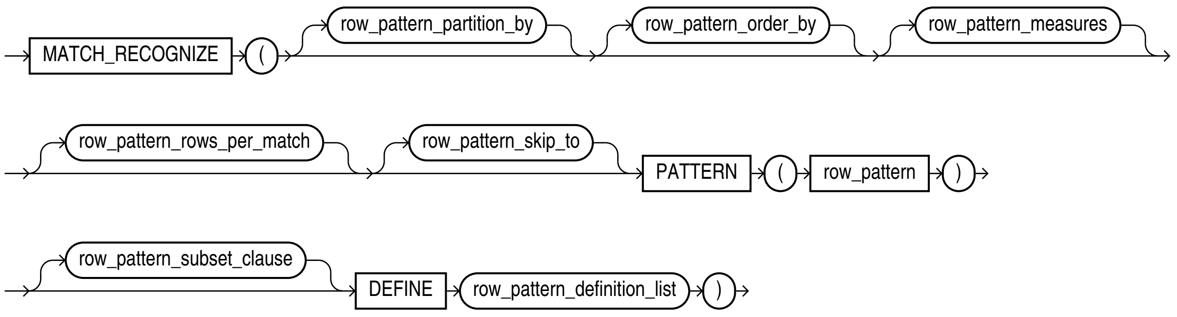
***accuracy_clause*::=**



***for_update_clause*::=**

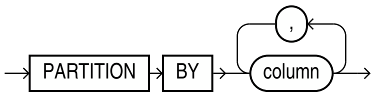


***row_pattern_clause*::=**

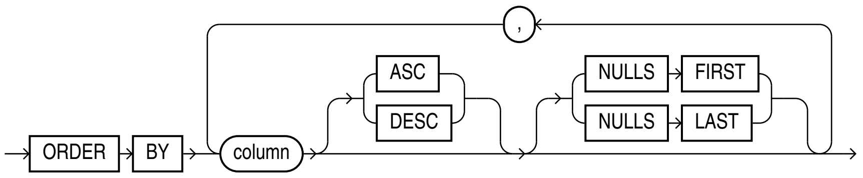


(*row_pattern_partition_by*::=, *row_pattern_order_by*::=, *row_pattern_measures*::=, *row_pattern_rows_per_match*::=, *row_pattern_skip_to*::=, *row_pattern*::=, *row_pattern_subset_clause*::=, *row_pattern_definition_list*::=)

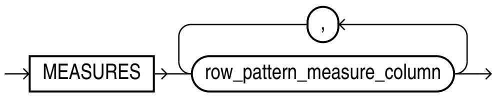
***row_pattern_partition_by*::=**



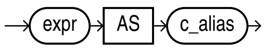
***row_pattern_order_by*::=**



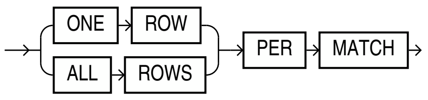
***row_pattern_measures*::=**



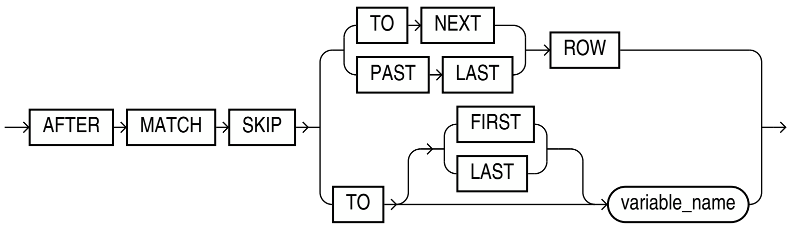
***row_pattern_measure_column*::=**



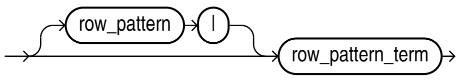
***row_pattern_rows_per_match*::=**



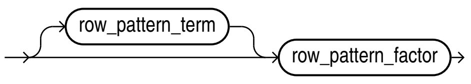
***row_pattern_skip_to*::=**



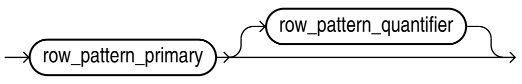
***row_pattern*::=**



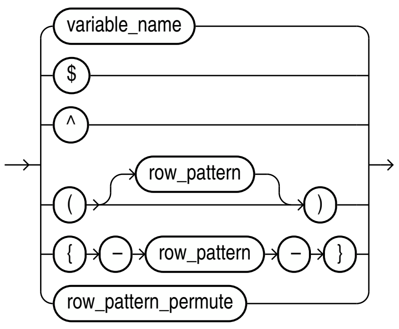
***row_pattern_term*::=**



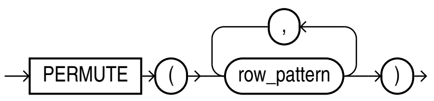
***row_pattern_factor*::=**



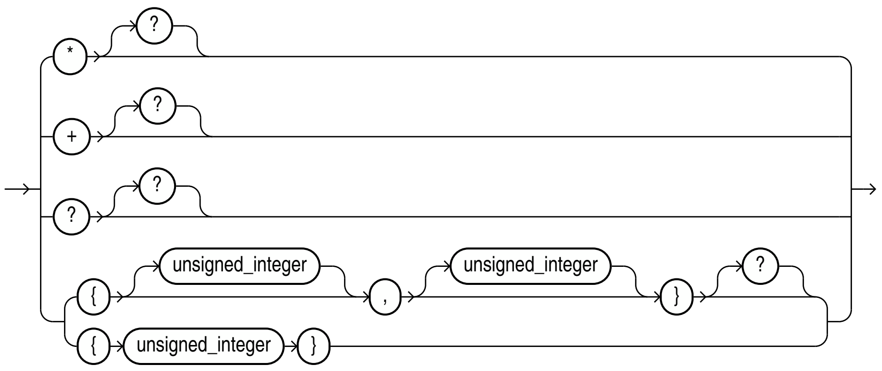
***row_pattern_primary*::=**



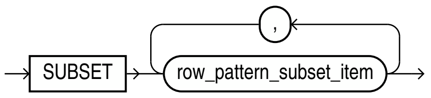
***row_pattern_permute*::=**



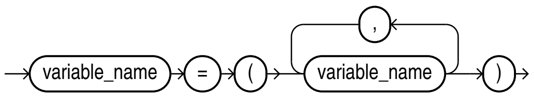
***row_pattern_quantifier*::=**



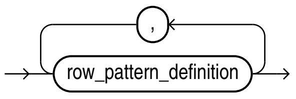
row_pattern_subset_clause::=



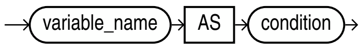
row_pattern_subset_item::=



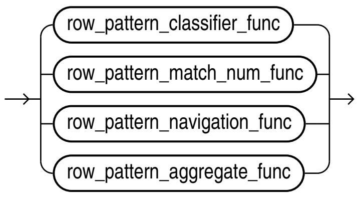
row_pattern_definition_list::=



row_pattern_definition::=

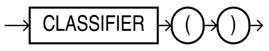


row_pattern_rec_func::=

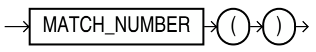

row_pattern_classifier_func
row_pattern_match_num_func
row_pattern_navigation_func
row_pattern_aggregate_func

(*row_pattern_classifier_func*::=, *row_pattern_match_num_func*::=, *row_pattern_navigation_func*::=, *row_pattern_aggregate_func*::=)

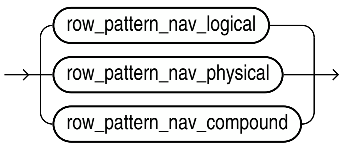
row_pattern_classifier_func::=



row_pattern_match_num_func::=

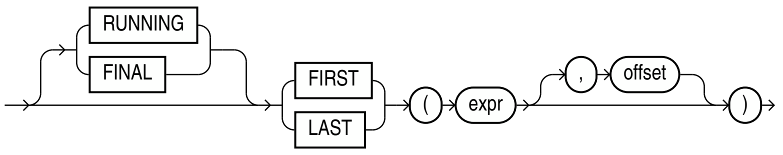


row_pattern_navigation_func::=

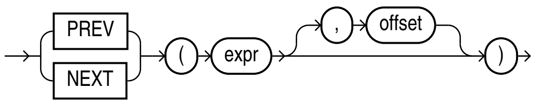


(*row_pattern_nav_logical*::=, *row_pattern_nav_physical*::=, *row_pattern_nav_compound*::=)

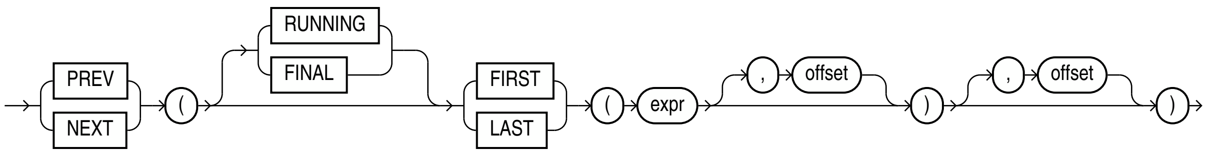
row_pattern_nav_logical::=



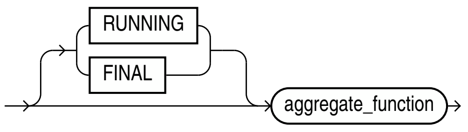
row_pattern_nav_physical::=



row_pattern_nav_compound::=



row_pattern_aggregate_func::=



Semantics

with_clause

Use the *with_clause* to define the following:

- PL/SQL procedures and functions (using the *plsql_declarations* clause)
- Subquery blocks (using *subquery_factoring_clause* or *subav_factoring_clause*, or both)

plsql_declarations

The *plsql_declarations* clause lets you declare and define PL/SQL functions and procedures. You can then reference the PL/SQL functions in the query in which you specify this clause, as well as its subqueries, if any. For the purposes of name resolution, these function names have precedence over schema-level stored functions.

If the query in which you specify this clause is not a top-level `SELECT` statement, then the following rules apply to the top-level SQL statement that contains the query:

- If the top-level statement is a `SELECT` statement, then it must have either a `WITH` *plsql_declarations* clause or the `WITH_PLSQL` hint.
- If the top-level statement is a `DELETE`, `MERGE`, `INSERT`, or `UPDATE` statement, then it must have the `WITH_PLSQL` hint.

The `WITH_PLSQL` hint only enables you to specify the `WITH` *plsql_declarations* clause within the statement. It is not an optimizer hint.

See Also:

- *Oracle Database PL/SQL Language Reference* for syntax and restrictions for *function_declaration* and *procedure_declaration*.
- "Using a PL/SQL Function in the WITH Clause: Examples"

subquery_factoring_clause

The *subquery_factoring_clause* lets you assign a name (*query_name*) to a subquery block. You can then reference the subquery block multiple places in the query by specifying *query_name*. Oracle Database optimizes the query by treating the *query_name* as either an inline view or as a temporary table. The *query_name* is subject to the same naming conventions and restrictions as database schema objects. Refer to "Database Object Naming Rules " for information on database object names.

The column aliases following the *query_name* and the set operators separating multiple subqueries in the `AS` clause are valid and required for recursive subquery factoring. The *search_clause* and *cycle_clause* are valid only for recursive subquery factoring but are not required. See "Recursive Subquery Factoring".

You can specify this clause in any top-level `SELECT` statement and in most types of subqueries. The query name is visible to the main query and to all subsequent subqueries. For recursive subquery factoring, the query name is even visible to the subquery that defines the query name itself.

Recursive Subquery Factoring

If a *subquery_factoring_clause* refers to its own *query_name* in the subquery that defines it, then the *subquery_factoring_clause* is said to be **recursive**. A recursive *subquery_factoring_clause* must contain two query blocks: the first is the **anchor member** and the second is the **recursive member**. The anchor member must appear before the recursive member, and it cannot reference *query_name*. The anchor member can be composed of one or more query blocks combined by the set operators: `UNION ALL`, `UNION`, `INTERSECT` or `MINUS`. The recursive member must follow the anchor member and must reference *query_name* exactly once. You must combine the recursive member with the anchor member using the `UNION ALL` set operator.

The number of column aliases following `WITH` *query_name* and the number of columns in the `SELECT` lists of the anchor and recursive query blocks must be the same.

The recursive member cannot contain any of the following elements:

- The `DISTINCT` keyword or a `GROUP BY` clause
- The *model_clause*
- An aggregate function. However, analytic functions are permitted in the select list.
- Subqueries that refer to *query_name*.
- Outer joins that refer to *query_name* as the right table.

In previous releases of Oracle Database, the recursive member of a recursive `WITH` clause ran serially regardless of the parallelism of the entire query (also known as the top-level `SELECT` statement). Beginning with Oracle Database 12*c* Release 2 (12.2), the recursive member runs in parallel if the optimizer determines that the top-level `SELECT` statement can be executed in parallel.

search_clause

Use the `SEARCH` clause to specify an ordering for the rows.

- Specify `BREADTH FIRST BY` if you want sibling rows returned before any child rows are returned.
- Specify `DEPTH FIRST BY` if you want child rows returned before any siblings rows are returned.
- Sibling rows are ordered by the columns listed after the `BY` keyword.
- The *c_alias* list following the `SEARCH` keyword must contain column names from the column alias list for *query_name*.
- The *ordering_column* is automatically added to the column list for the query name. The query that selects from *query_name* can include an `ORDER BY` on *ordering_column* to return the rows in the order that was specified by the `SEARCH` clause.

cycle_clause

Use the `CYCLE` clause to mark cycles in the recursion.

- The *c_alias* list following the `CYCLE` keyword must contain column names from the column alias list for *query_name*. Oracle Database uses these columns to detect a cycle.
- *cycle_value* and *no_cycle_value* should be character strings of length 1.

- If a cycle is detected, then the cycle mark column specified by *cycle_mark_c_alias* for the row causing the cycle is set to the value specified for *cycle_value*. The recursion will then stop for this row. That is, it will not look for child rows for the offending row, but it will continue for other noncyclic rows.
- If no cycles are found, then the cycle mark column is set to the default value specified for *no_cycle_value*.
- The cycle mark column is automatically added to the column list for the *query_name*.
- A row is considered to form a cycle if one of its ancestor rows has the same values for the cycle columns.

If you omit the `CYCLE` clause, then the recursive `WITH` clause returns an error if cycles are discovered. In this case, a row forms a cycle if one of its ancestor rows has the same values for all the columns in the column alias list for *query_name* that are referenced in the `WHERE` clause of the recursive member.

Restrictions on Subquery Factoring

This clause is subject to the following restrictions:

- You can specify only one *subquery_factoring_clause* in a single SQL statement. Any *query_name* defined in the *subquery_factoring_clause* can be used in any subsequent named query block in the *subquery_factoring_clause*.
- In a compound query with set operators, you cannot use the *query_name* for any of the component queries, but you can use the *query_name* in the `FROM` clause of any of the component queries.
- You cannot specify duplicate names in the column alias list for *query_name*.
- The name used for the *ordering_column* has to be different from the name used for *cycle_mark_c_alias*.
- The *ordering_column* and cycle mark column names cannot already be in the column alias list for *query_name*.

See Also:

- *Oracle Database Concepts* for information about inline views
- "Subquery Factoring: Example"
- "Recursive Subquery Factoring: Examples"

subav_factoring_clause

With the *subav_factoring_clause*, you can define a transitory analytic view that filters fact data prior to aggregation or adds calculated measures to a query of an analytic view. The *subav_name* argument assigns a name to the transitory analytic view. You can then reference the transitory analytic view multiple places in the query by specifying *subav_name*. The *subav_name* is subject to the same naming conventions and restrictions as database schema objects. Refer to "Database Object Naming Rules " for information on database object names.

You can specify this clause in any top-level `SELECT` statement and in most types of subqueries. The query name is visible to the main query and to all subsequent subqueries.

The *sub_av_clause* argument defines a transitory analytic view.

sub_av_clause

With the `USING` keyword, specify the name of an analytic view, which may be a transitory analytic view previously defined in the `WITH` clause or it may be a persistent analytic view. A persistent analytic view is defined in a `CREATE ANALYTIC VIEW` statement. If the analytic view is a persistent one, then the current user must have select access on it.

See Also:

Analytic Views: Examples

hierarchies_clause

The *`hierarchies_clause`* specifies the hierarchies of the base analytic view that the results of the transitory analytic view are dimensioned by. With the `HIERARCHIES` keyword, specify the alias of one or more hierarchies of the base analytic view.

If you do not specify a `HIERARCHIES` clause, then the default hierarchies of the base analytic view are used.

filter_clauses

You may specify a given *`hier_alias`* in at most one *`filter_clause`*.

filter_clause

The filter clause applies the specified predicate condition to the fact table, which reduces the number of rows returned from the table before aggregation of the measure values. The predicate may contain any SQL row function or operation. The predicate may refer to any attribute of the specified hierarchy or it may refer to a measure of the analytic view if you specify the `MEASURES` keyword.

For example, the following clause restricts the aggregation of measure values to those for the first and second quarters of every year of a time hierarchy.

```
FILTER FACT (time_hier TO quarter_of_year IN (1,2))
```

If you then select from the transitory analytic view the sales for the years 2000 and 2001, the values returned are the aggregated values of the first and second quarters only.

An example of specifying a predicate for a measure in the filter clause is the following.

```
FILTER FACT (MEASURES TO sales BETWEEN 100 AND 200)
```

attr_dim_alias

The alias of an attribute dimension in the base analytic view. The `USER_ANALYTIC_VIEW_DIMENSIONS` view contains the aliases of the attribute dimensions in an analytic view.

hier_alias

The alias of a hierarchy in the base analytic view. The `USER_ANALYTIC_VIEW_HIERS` view contains the aliases of the hierarchies in an analytic view.

add_meas_clause

With the `ADD MEASURES` keywords, you may add calculated measures to the transitory analytic view.

calc_meas_clause

Specify a name for the calculated measure and an analytic view expression that specifies values for the calculated measure. The analytic view expression can be any valid *`calc_meas_expression`* as described in Analytic View Expressions. For example, the following adds a calculated measure named “share_sales.”

```
ADD MEASURES (share_sales AS (SHARE_OF(sales HIERARCHY time_hier PARENT)))
```

hint

Specify a comment that passes instructions to the optimizer on choosing an execution plan for the statement.

> **See Also:**
>
> "Hints " for the syntax and description of hints

DISTINCT | UNIQUE

Specify `DISTINCT` or `UNIQUE` if you want the database to return only one copy of each set of duplicate rows selected. These two keywords are synonymous. Duplicate rows are those with matching values for each expression in the select list.

Restrictions on DISTINCT and UNIQUE Queries

These types of queries are subject to the following restrictions:

- When you specify `DISTINCT` or `UNIQUE`, the total number of bytes in all select list expressions is limited to the size of a data block minus some overhead. This size is specified by the initialization parameter `DB_BLOCK_SIZE`.
- You cannot specify `DISTINCT` if the *`select_list`* contains LOB columns.

ALL

Specify `ALL` if you want the database to return all rows selected, including all copies of duplicates. The default is `ALL`.

select_list

The *`select_list`* lets you specify the columns you want to retrieve from the database.

* (all-column wildcard)

Specify the all-column wildcard (asterisk) to select all columns, excluding pseudocolumns and `INVISIBLE` columns, from all tables, views, or materialized views listed in the `FROM` clause. The columns are returned in the order indicated by the `COLUMN_ID` column of the `*_TAB_COLUMNS` data dictionary view for the table, view, or materialized view.

If you are selecting from a table rather than from a view or a materialized view, then columns that have been marked as `UNUSED` by the `ALTER TABLE SET UNUSED` statement are not selected.

> **See Also:**
>
> ALTER TABLE, "Simple Query Examples", and "Selecting from the DUAL Table: Example"

query_name*.

Specify *query_name* followed by a period and the asterisk to select all columns from the specified subquery block. For *query_name*, specify a subquery block name already specified in the *subquery_factoring_clause*. You must have specified the *subquery_factoring_clause* in order to specify *query_name* in the *select_list*. If you specify *query_name* in the *select_list*, then you also must specify *query_name* in the *query_table_expression* (`FROM` clause).

table*.* | *view*.* | *materialized view*.

Specify the object name followed by a period and the asterisk to select all columns from the specified table, view, or materialized view. Oracle Database returns a set of columns in the order in which the columns were specified when the object was created. A query that selects rows from two or more tables, views, or materialized views is a join.

You can use the schema qualifier to select from a table, view, or materialized view in a schema other than your own. If you omit *schema*, then the database assumes the table, view, or materialized view is in your own schema.

> **See Also:**
>
> "Joins "

t_alias* .

Specify a correlation name (alias) followed by a period and the asterisk to select all columns from the object with that correlation name specified in the `FROM` clause of the same subquery. The object can be a table, view, materialized view, or subquery. Oracle Database returns a set of columns in the order in which the columns were specified when the object was created. A query that selects rows from two or more objects is a join.

expr

Specify an expression representing the information you want to select. A column name in this list can be qualified with *schema* only if the table, view, or materialized view containing the column is qualified with *schema* in the `FROM` clause. If you specify a member method of an object type, then you must follow the method name with parentheses even if the method takes no arguments.

The expression can also hold a scalar value that can be return values of PL/SQL functions, subqueries that return a single value per row, and SQL macros.

c_alias

Specify an alias for the column expression. Oracle Database will use this alias in the column heading of the result set. The `AS` keyword is optional. The alias effectively renames the select list item for the duration of the query. The alias can be used in the *`order_by_clause`* but not other clauses in the query.

From Release 23 you can use *`c_alias`* in *`group_by_clause`* .

See Also:

- *Oracle Database Data Warehousing Guide* for information on using the *`expr`* `AS` *`c_alias`* syntax with the `UNION ALL` operator in queries of multiple materialized views
- "About SQL Expressions " for the syntax of *`expr`*

Restrictions on the Select List

The select list is subject to the following restrictions:

- If you also specify a *group_by_clause* in this statement, then this select list can contain only the following types of expressions:
  – Constants
  – Aggregate functions and the functions `USER`, `UID`, and `SYSDATE`
  – Expressions identical to those in the *`group_by_clause`*. If the *`group_by_clause`* is in a subquery, then all columns in the select list of the subquery must match the `GROUP BY` columns in the subquery. If the select list and `GROUP BY` columns of a top-level query or of a subquery do not match, then the statement results in `ORA-00979`.

    From Release 23 you can group by *`position`* and *`alias`*.
  – Expressions involving the preceding expressions that evaluate to the same value for all rows in a group
- You can select a rowid from a join view only if the join has one and only one key-preserved table. The rowid of that table becomes the rowid of the view.

  See Also:

  Oracle Database Administrator's Guide for information on key-preserved tables
- If two or more tables have some column names in common, and if you are specifying a join in the `FROM` clause, then you must qualify column names with names of tables or table aliases.

FROM Clause

Use the optional `FROM` clause to specify the objects from which data is selected.

You can invoke a polymorphic table function (PTF) in the query block of the `FROM` clause like other existing table functions. A PTF is a table function whose operands can have more than one type.

With Oracle Database 21c, you can write SQL table macros and use them inside the `FROM` clause, where it would be legal to call a PL/SQL function. SQL table macros are expressions, typically used in a `FROM` clause, to act as a sort of polymorphic (parameterized) views. You must define these macro functions in PL/SQL and call them from SQL for them to function as macros.

With Oracle Database Release 23, you can use the `GRAPH_TABLE` operator as a table expression in the `FROM` clause.

See Also:

- GRAPH_TABLE Operator
- *PL/SQL Optimization and Tuning*
- Defining SQL Macros

ONLY

The `ONLY` clause applies only to views. Specify `ONLY` if the view in the `FROM` clause is a view belonging to a hierarchy and you do not want to include rows from any of its subviews.

query_table_expression

Use the *`query_table_expression`* clause to identify a subquery block, table, view, materialized view, analytic view, hierarchy, partition, or subpartition, or to specify a subquery that identifies the objects. In order to specify a subquery block, you must have specified the subquery block name (*`query_name`* in the *`subquery_factoring_clause`* or *`subav_name`* in the *`subav_factoring_clause`*).

The analytic view in the expression may be a transitory analytic view defined in the *`with_clause`* or a persistent analytic view.

See Also:

"Using Subqueries: Examples"

LATERAL

Specify `LATERAL` to designate *`subquery`* as a lateral inline view. Within a lateral inline view, you can specify tables that appear to the left of the lateral inline view in the `FROM` clause of a query. You can specify this left correlation anywhere within *`subquery`* (such as the `SELECT`, `FROM`, and `WHERE` clauses) and at any nesting level.

Restrictions on LATERAL

Lateral inline views are subject to the following restrictions:

- If you specify `LATERAL`, then you cannot specify the *`pivot_clause`*, the *`unpivot_clause`*, or a pattern in the *`table_reference`* clause.
- If a lateral inline view contains the *`query_partition_clause`*, and it is the right side of a join clause, then it cannot contain a left correlation to the left table in the join clause.

However, it can contain a left correlation to a table to its left in the `FROM` clause that is not the left table.

- A lateral inline view cannot contain a left correlation to the first table in a right outer join or full outer join.

> **See Also:**
>
> "Using Lateral Inline Views: Example"

inline_external_table

Specify this clause to inline an external table in a query. You must specify the table columns and properties for the external table that will be inlined in the query.

inline_external_table_properties

This clause extends the `external_table_data_props` with the `REJECT LIMIT` and `access_driver_type` options. Use this clause to specify the properties of the external table.

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

modified_external_table

You can use this clause to override some external table properties specified by the `CREATE TABLE` or `ALTER TABLE` statements from within a query.

You can override external table parameters at runtime.

Restrictions

- You must specify the key words `EXTERNAL MODIFY` in the query. If you do not specify the keywords, you will see a `Missing or invalid option` error.
- You must reference an external table in the query. If you do not, you will see an error.
- You must specify at least *one* property in the query. One of `DEFAULT DIRECTORY`, `LOCATION`, `ACCESS PARAMETERS`, or `REJECT LIMIT`.
- If you specify more than one external table properties, they must be listed in order. First the `DEFAULT DIRECTORY` must be specified, followed by the `ACCESS PARAMETERS`, `LOCATION` and `REJECT LIMIT`. Otherwise an error will be raised.
- In the `DEFAULT DIRECTORY` clause, you must specify only one proper default directory. Otherwise a `Missing DEFAULT keyword` error will occur.
- You must enclose a filename in the `LOCATION` clause within quotes. Otherwise a `Missing keyword` error will occur. Note that the access driver will decide whether or not to allow a `LOCATION` clause in the query. If the clause is disallowed for a particular access driver, an error will be raised.
- For `ORACLE_LOADER` and `ORACLE_DATAPUMP` access drivers, the external file location in the `LOCATION` clause must be specified in the following format: directory: location, i.e, the directory and location must be separated by a colon. Multiple locations in the clause must be separated by a comma. Otherwise, a `Missing keyword` error will occur.

- Note that LOCATION will be made optional in CREATE TABLE, and must be specified either when creating or querying the external table. Otherwise an error will be raised in the access driver.
- When populating external data using ORACLE DATAPUMP via CTAS, the external file location must be specified. This will be the only case where LOCATION clause is mandatory in CREATE TABLE.
- When overriding access parameters, a proper access parameter list must be provided in the ACCESS PARAMETERS clause, with enclosing parentheses.

  Note that the syntax and allowable values for the access parameters in the *modified_external_table* clause are the same as for the external table DDL for each access driver. For more see *Oracle Database Utilities* for additional details regarding syntax and permissible values.
- If you specify the REJECT LIMIT, then it must either be UNLIMITED or some valid value that is within range. Otherwise a Reject limit out of range error will be raised.

modify_external_table_properties

You can specify the external table properties that you want to modify at run time using this clause. The parameters that you can modify are DEFAULT DIRECTORY, LOCATION, ACCESS PARAMETERS (BADFILE, LOGFILE, DISCARDFILE) and REJECT LIMIT.

Example: Overriding External Table Parameters in a Query

```
SELECT * FROM
sales_external EXTERNAL MODIFY (LOCATION 'sales_9.csv’ REJECT LIMIT UNLIMITED);
```

flashback_query_clause

Use the *flashback_query_clause* to retrieve data from a table, view, or materialized view based on time dimensions associated with the data.

This clause implements SQL-driven Flashback, which lets you specify the following:

- A different system change number or timestamp for each object in the select list, using the clauses VERSIONS BETWEEN { SCN | TIMESTAMP } or VERSIONS AS OF { SCN | TIMESTAMP }. You can also implement session-level Flashback using the DBMS_FLASHBACK package.
- A valid time period for each object in the select list, using the clauses VERSIONS PERIOD FOR or AS OF PERIOD FOR. You can also implement valid-time session-level Flashback using the DBMS_FLASHBACK_ARCHIVE package.

A Flashback Query lets you retrieve a history of changes made to a row. You can retrieve the corresponding identifier of the transaction that made the change using the VERSIONS_XID pseudocolumn. You can also retrieve information about the transaction that resulted in a particular row version by issuing an Oracle Flashback Transaction Query. You do this by querying the FLASHBACK_TRANSACTION_QUERY data dictionary view for a particular transaction ID.

VERSIONS BETWEEN { SCN | TIMESTAMP }

Specify VERSIONS BETWEEN to retrieve multiple versions of the rows returned by the query. Oracle Database returns all committed versions of the rows that existed between two SCNs or between two timestamp values. The first specified SCN or timestamp must be earlier than the second specified SCN or timestamp. The rows returned include deleted and subsequently reinserted versions of the rows.

- Specify `VERSIONS BETWEEN SCN` ... to retrieve the versions of the row that existed between two SCNs. Both expressions must evaluate to a number and cannot evaluate to NULL. `MINVALUE` and `MAXVALUE` resolve to the SCN of the oldest and most recent data available, respectively.
- Specify `VERSIONS BETWEEN TIMESTAMP` ... to retrieve the versions of the row that existed between two timestamps. Both expressions must evaluate to a timestamp value and cannot evaluate to NULL. `MINVALUE` and `MAXVALUE` resolve to the timestamp of the oldest and most recent data available, respectively.

AS OF { SCN | TIMESTAMP }

Specify `AS OF` to retrieve the single version of the rows returned by the query at a particular change number (SCN) or timestamp. If you specify `SCN`, then *`expr`* must evaluate to a number. If you specify `TIMESTAMP`, then *`expr`* must evaluate to a timestamp value. In either case, *`expr`* cannot evaluate to NULL. Oracle Database returns rows as they existed at the specified system change number or time.

Oracle Database provides a group of version query pseudocolumns that let you retrieve additional information about the various row versions. Refer to "Version Query Pseudocolumns
" for more information.

When both clauses are used together, the `AS OF` clause determines the SCN or moment in time from which the database issues the query. The `VERSIONS` clause determines the versions of the rows as seen from the `AS OF` point. The database returns null for a row version if the transaction started before the first `BETWEEN` value or ended after the `AS OF` point.

VERSIONS PERIOD FOR

Specify `VERSIONS PERIOD FOR` to retrieve rows from *`table`* based on whether they are considered valid during the specified time period. In order to use this clause, *`table`* must support Temporal Validity.

- For *`valid_time_column`*, specify the name of the valid time dimension column for *`table`*.
- Use the `BETWEEN` clause to specify the time period during which rows are considered valid. Both expressions must evaluate to a timestamp value and cannot evaluate to NULL. `MINVALUE` resolves to the earliest date or timestamp in the start time column of *`table`*. `MAXVALUE` resolves to latest date or timestamp in the end time column of *`table`*.

AS OF PERIOD FOR

Specify `AS OF PERIOD FOR` to retrieve rows from *`table`* based on whether they are considered valid as of the specified time. In order to use this clause, *`table`* must support Temporal Validity.

- For *`valid_time_column`*, specify the name of the valid time dimension column for *`table`*.
- Use *`expr`* to specify the time as of which rows are considered valid. The expression must evaluate to a timestamp value and cannot evaluate to NULL.

See Also:

- *Oracle Database Development Guide* for more information on Temporal Validity
- `CREATE TABLE` *period_definition* to learn how to configure a table to support Temporal Validity and for information about the *`valid_time_column`*, start time column, and end time column

Note on Flashback Queries

When performing a flashback query, Oracle Database might not use query optimizations that it would use for other types of queries, which could have a negative impact on performance. In particular, this occurs when you specify multiple flashback queries in a hierarchical query.

Restrictions on Flashback Queries

These queries are subject to the following restrictions:

- You cannot specify a column expression or a subquery in the expression of the `AS OF` clause.
- You cannot specify the `AS OF` clause if you have specified the *`for_update_clause`*.
- You cannot use the `AS OF` clause in the defining query of a materialized view.
- You cannot use the `VERSIONS` clause in flashback queries to temporary or external tables, or tables that are part of a cluster.
- You cannot use the `VERSIONS` clause in flashback queries to views. However, you can use the `VERSIONS` syntax in the defining query of a view.
- You cannot specify the *`flashback_query_clause`* if you have specified *`query_name`* in the *`query_table_expression`*.

See Also:

- *Oracle Database Development Guide* for more information on Oracle Flashback Query
- "Using Flashback Queries: Example"
- *Oracle Database Development Guide* and *Oracle Database PL/SQL Packages and Types Reference* for information about session-level Flashback using the `DBMS_FLASHBACK` package
- *Oracle Database Administrator's Guide* and to the description of `FLASHBACK_TRANSACTION_QUERY` in the *Oracle Database Reference* for more information about transaction history

partition_extension_clause

For `PARTITION` or `SUBPARTITION`, specify the name or key value of the partition or subpartition within *`table`* from which you want to retrieve data.

For range- and list-partitioned data, as an alternative to this clause, you can specify a condition in the `WHERE` clause that restricts the retrieval to one or more partitions of *`table`*. Oracle Database will interpret the condition and fetch data from only those partitions. It is not possible to formulate such a `WHERE` condition for hash-partitioned data.

See Also:

"References to Partitioned Tables and Indexes " and "Selecting from a Partition: Example"

dblink

For *dblink*, specify the complete or partial name for a database link to a remote database where the table, view, or materialized view is located. This database need not be an Oracle Database.

See Also:

- "References to Objects in Remote Databases " for more information on referring to database links
- "Distributed Queries " for more information about distributed queries and "Using Distributed Queries: Example"

If you omit *dblink*, then the database assumes that the table, view, or materialized view is on the local database.

Restrictions on Database Links

Database links are subject to the following restrictions:

- You cannot query a user-defined type or an object `REF` on a remote table.
- You cannot query columns of type `ANYTYPE`, `ANYDATA`, or `ANYDATASET` from remote tables.

table* | *view* | *materialized_view* | *analytic_view* | *hierarchy

Specify the name of a table, view, materialized view, analytic view, or hierarchy from which data is selected.

analytic_view

A persistent analytic view defined with the `CREATE ANALYTIC VIEW` statement or a transitory analytic view defined in a `WITH` clause.

See Also:

Analytic Views: Examples

hierarchy

A hierarchy defined with the `CREATE HIERARCHY` statement.

sample_clause

The *sample_clause* lets you instruct the database to select from a random sample of data from the table, rather than from the entire table.

See Also:

"Selecting a Sample: Examples"

BLOCK

`BLOCK` instructs the database to attempt to perform random block sampling instead of random row sampling.

Block sampling is possible only during full table scans or index fast full scans. If a more efficient execution path exists, then Oracle Database does not perform block sampling. If you want to guarantee block sampling for a particular table or index, then use the `FULL` or `INDEX_FFS` hint.

Beginning with Oracle Database 12*c* Release 2 (12.2.), you can specify block sampling for external tables. In earlier releases, specifying block sampling for external tables had no effect; row sampling was performed.

sample_percent

For *`sample_percent`*, specify the percentage of the total row or block count to be included in the sample. The value must be in the range .000001 to, but not including, 100. This percentage indicates the probability of each row, or each cluster of rows in the case of block sampling, being selected as part of the sample. It does not mean that the database will retrieve exactly *`sample_percent`* of the rows of *`table`*.

WARNING:

The use of statistically incorrect assumptions when using this feature can lead to incorrect or undesirable results.

SEED *seed_value*

Specify this clause to instruct the database to attempt to return the same sample from one execution to the next. The *`seed_value`* must be an integer between 0 and 4294967295. If you omit this clause, then the resulting sample will change from one execution to the next.

Restrictions on *sample_clause*

The following restrictions apply to the `SAMPLE` clause:

- You cannot specify the `SAMPLE` clause in a subquery in a DML statement.
- You can specify the `SAMPLE` clause in a query on a base table, a container table of a materialized view, or a view that is key preserving. You cannot specify this clause on a view that is not key preserving.

subquery_restriction_clause

The *`subquery_restriction_clause`* lets you restrict the subquery in one of the following ways:

WITH READ ONLY

Specify `WITH READ ONLY` to indicate that the table or view cannot be updated.

WITH CHECK OPTION

Specify `WITH CHECK OPTION` to indicate that Oracle Database prohibits any changes to the table or view that would produce rows that are not included in the subquery. When used in the subquery of a DML statement, you can specify this clause in a subquery in the `FROM` clause but not in subquery in the `WHERE` clause.

CONSTRAINT *constraint*

Specify the name of the `CHECK OPTION` constraint. If you omit this identifier, then Oracle automatically assigns the constraint a name of the form `SYS_C`*`n`*, where n is an integer that makes the constraint name unique within the database.

See Also:

"Using the WITH CHECK OPTION Clause: Example"

table_collection_expression

The *`table_collection_expression`* lets you inform Oracle that the value of *`collection_expression`* should be treated as a table for purposes of query and DML operations. The *`collection_expression`* can be a subquery, a column, a function, or a collection constructor. Regardless of its form, it must return a collection value—that is, a value whose type is nested table or varray. This process of extracting the elements of a collection is called **collection unnesting**.

The optional plus (+) is relevant if you are joining the `TABLE` collection expression with the parent table. The + creates an outer join of the two, so that the query returns rows from the outer table even if the collection expression is null.

Note:

In earlier releases of Oracle, when *`collection_expression`* was a subquery, *`table_collection_expression`* was expressed as `THE` *`subquery`*. That usage is now deprecated.

The *`collection_expression`* can reference columns of tables defined to its left in the `FROM` clause. This is called **left correlation**. Left correlation can occur only in *`table_collection_expression`*. Other subqueries cannot contains references to columns defined outside the subquery.

The optional `(+)` lets you specify that *`table_collection_expression`* should return a row with all fields set to null if the collection is null or empty. The `(+)` is valid only if *`collection_expression`* uses left correlation. The result is similar to that of an outer join.

When you use the `(+)` syntax in the `WHERE` clause of a subquery in an `UPDATE` or `DELETE` operation, you must specify two tables in the `FROM` clause of the subquery. Oracle Database ignores the outer join syntax unless there is a join in the subquery itself.

See Also:

- "Outer Joins "
- "Table Collections: Examples" and "Collection Unnesting: Examples"

t_alias

Specify a **correlation name**, which is an alias for the table, view, materialized view, or subquery for evaluating the query. This alias is required if the select list references any object type attributes or object type methods. Correlation names are most often used in a correlated query. Other references to the table, view, or materialized view throughout the query must refer to this alias.

> **See Also:**
>
> "Using Correlated Subqueries: Examples"

pivot_clause

The `pivot_clause` lets you write cross-tabulation queries that rotate rows into columns, aggregating data in the process of the rotation. The output of a pivot operation typically includes more columns and fewer rows than the starting data set. The `pivot_clause` performs the following steps:

1. The `pivot_clause` computes the aggregation functions specified at the beginning of the clause. Aggregation functions must specify a `GROUP BY` clause to return multiple values, yet the `pivot_clause` does not contain an explicit `GROUP BY` clause. Instead, the `pivot_clause` performs an implicit `GROUP BY`. The implicit grouping is based on all the columns not referred to in the `pivot_clause`, along with the set of values specified in the `pivot_in_clause`.). If you specify more than one aggregation function, then you must provide aliases for at least all but one of the aggregation functions.
2. The grouping columns and aggregated values calculated in Step 1 are configured to produce the following cross-tabular output:
   a. All the implicit grouping columns not referred to in the `pivot_clause`, followed by
   b. New columns corresponding to values in the `pivot_in_clause`. Each aggregated value is transposed to the appropriate new column in the cross-tabulation. If you specify the `XML` keyword, then the result is a single new column that expresses the data as an XML string. The database generates a name for each new column. If you do not provide an alias for an aggregation function, then the database uses each pivot column value as the name for each new column to which that aggregated value is transposed. If you provide an alias for an aggregation function, then the database generates a name for each new column to which that aggregated value is transposed by concatenating the pivot column name, the underscore character (_), and the aggregation function alias. If a generated column name exceeds the maximum length of a column name, then an ORA-00918 error is returned. To avoid this issue, specify a shorter alias for the pivot column heading, the aggregation function, or both.

The subclauses of the `pivot_clause` have the following semantics:

XML

The optional `XML` keyword generates XML output for the query. The `XML` keyword permits the `pivot_in_clause` to contain either a subquery or the wildcard keyword `ANY`. Subqueries and `ANY` wildcards are useful when the `pivot_in_clause` values are not known in advance. With XML output, the values of the pivot column are evaluated at execution time. You cannot specify `XML` when you specify explicit pivot values using expressions in the `pivot_in_clause`.

When XML output is generated, the aggregate function is applied to each distinct pivot value, and the database returns a column of `XMLType` containing an XML string for all value and measure pairs.

expr

For *`expr`*, specify an expression that evaluates to a constant value of a pivot column. You can optionally provide an alias for each pivot column value. If there is no alias, the column heading becomes a quoted identifier.

subquery

A subquery is used only in conjunction with the `XML` keyword. When you specify a subquery, all values found by the subquery are used for pivoting. The output is not the same cross-tabular format returned by non-XML pivot queries. Instead of multiple columns specified in the *`pivot_in_clause`*, the subquery produces a single XML string column. The XML string for each row holds aggregated data corresponding to the implicit `GROUP BY` value of that row. The XML string for each output row includes all pivot values found by the subquery, even if there are no corresponding rows in the input data.

The subquery must return a list of unique values at the execution time of the pivot query. If the subquery does not return a unique value, then Oracle Database raises a run-time error. Use the `DISTINCT` keyword in the subquery if you are not sure the query will return unique values.

ANY

The `ANY` keyword is used only in conjunction with the `XML` keyword. The `ANY` keyword acts as a wildcard and is similar in effect to *`subquery`*. The output is not the same cross-tabular format returned by non-XML pivot queries. Instead of multiple columns specified in the *`pivot_in_clause`*, the `ANY` keyword produces a single XML string column. The XML string for each row holds aggregated data corresponding to the implicit `GROUP BY` value of that row. However, in contrast to the behavior when you specify *`subquery`*, the `ANY` wildcard produces an XML string for each output row that includes only the pivot values found in the input data corresponding to that row.

See Also:

Oracle Database Data Warehousing Guide for more information about `PIVOT` and `UNPIVOT` and "Using PIVOT and UNPIVOT: Examples"

unpivot_clause

The *`unpivot_clause`* rotates columns into rows.

- The `INCLUDE` | `EXCLUDE NULLS` clause gives you the option of including or excluding null-valued rows. `INCLUDE NULLS` causes the unpivot operation to include null-valued rows; `EXCLUDE NULLS` eliminates null-values rows from the return set. If you omit this clause, then the unpivot operation excludes nulls.
- For *`column`*, specify a name for each output column that will hold measure values, such as `sales_quantity`.
- In the *`pivot_for_clause`*, specify a name for each output column that will hold descriptor values, such as quarter or product.
- In the *`unpivot_in_clause`*, specify the input data columns whose names will become values in the output columns of the *`pivot_for_clause`*. These input data columns have

names specifying a category value, such as Q1, Q2, Q3, Q4. The optional `AS` clause lets you map the input data column names to the specified *`literal`* values in the output columns.

The unpivot operation turns a set of value columns into one column. Therefore, the data types of all the value columns must be in the same data type group, such as numeric or character.

- If all the value columns are `CHAR`, then the unpivoted column is `CHAR`. If any value column is `VARCHAR2`, then the unpivoted column is `VARCHAR2`.
- If all the value columns are `NUMBER`, then the unpivoted column is `NUMBER`. If any value column is `BINARY_DOUBLE`, then the unpivoted column is `BINARY_DOUBLE`. If no value column is `BINARY_DOUBLE` but any value column is `BINARY_FLOAT`, then the unpivoted column is `BINARY_FLOAT`.

containers_clause

The `CONTAINERS` clause is useful in a multitenant container database (CDB). This clause lets you query data in the specified table or view across all containers in a CDB.

- To query data in a CDB, you must be a common user connected to the CDB root, and the table or view must exist in the root and all PDBs. The query returns all rows from the table or view in the CDB root and in all open PDBs.
- To query data in an application container, you must be a common user connected to the application root, and the table or view must exist in the application root and all PDBs in the application container. The query returns all rows from the table or view in the application root and in all open PDBs in the application container.

The table or view must be in your own schema. It is not necessary to specify *`schema`*, but if you do then you must specify your own schema.

The query returns all rows from the table or view in the root and in all open PDBs, except PDBs that are open in `RESTRICTED` mode. If the queried table or view does not already contain a `CON_ID` column, then the query adds a `CON_ID` column to the query result, which identifies the container whose data a given row represents.

See Also:

- CONTAINERS Hint
- *Oracle Database Administrator's Guide* for more information on the `CONTAINERS` clause

shards_clause

Use the *`shards_clause`* to query Oracle supplied objects such as `V$`, `DBA/USER/ALL` views, and dictionary tables across shards. You can execute a query with the *`shards_clause`* only on the shard catalog database.

This feature enables easier centralized management by providing the ability to execute queries across all shards from a central shard catalog.

values_clause

You can use the *`values_clause`* in the `FROM` and *`with_clause`* of `SELECT` as a table value constructor (TVC).

Each table value constructor contains a set of row value expressions (RVE). The elements in each row expression should be homogeneous in number and their type must be compatible.

The `c_alias` or column alias is the name of the column corresponding to each expression in an RVE.

TVCs in the `FROM` clause of select statements can be used as table expressions.

Example: Using the Values Constructor in the FROM Clause of SELECT

```
SELECT *
            FROM ( VALUES (1,'SCOTT'),
                          (2,'SMITH'),
                          (3,'JOHN' )
                 ) t1 (employee_id, first_name);
```

The example above creates an in-line table `t1` with two columns `employee_id` and `first_name` and three rows.

If you use the *`values_clause`* with the *with_clause*::=, you must specify the column alias. Each column alias must correspond to the column produced by the TVC. In this case, the TVC replaces the subquery.

Example: Using the Values Constructor in the With_Clause of SELECT:

```
WITH X(foo, bar, baz) AS (
         VALUES (0, 1, 2), (3, 4, 5), (6, 7, 8) ) SELECT * FROM X;
```

The table and column aliases (`t_alias` and `c_alias`) are required unless you use `values_clause` with `with_clause` in `SELECT`.

Restrictions

- If multiple RVEs are specified, then each RVE should have the same cardinality. This means that each RVE must have the same number of elements.
- Each element of the RVE can be a valid SQL expression that includes a column name, scalar valued subquery, bind variable, or any other expression that evaluates to a single value.
- The type of the expression or a constant at the corresponding positions of RVE in a TVC should be implicitly convertible to the most general type following normal SQL type conversion rules. The type of expression that will be inferred will be the most general type of expression at the same position in all RVEs that constitute the TVC.
- If a scalar valued subquery is used to compute the value of an element in a RVE then the select list of scalar valued subquery can contain exactly one expression.
- If RVE is used in an `UPDATE`, or `MERGE` statement, then the keyword `DEFAULT` can be specified in a RVE for each position to indicate to the SQL engine that the default column value should be used for this column.
- The execution plan will have a new section that appears only when the TVC has RVEs consisting of constant values.
- If the types of the corresponding elements in a RVE in a TVC have different constraints, then the type of the column will be the union of all the constraints or the most relaxed constraint.

- An error will be thrown if a TVC, that consists of more than one RVE, is used in a place where a scalar valued subquery is expected.
- The parallel behavior will be similar to union all queries on `DUAL`. TVC will not impact parallel behavior.
- RVEs cannot be nested, that is, a RVE cannot contain another RVE.
- The maximum number of columns produced by the *`with_clause`* will be the same as the maximum number of columns in a database table.
- NDV and other statistics that are computed by the optimizer will be similar to a union of all queries on `DUAL`.
- The TVC clause will not have any restriction on number of RVEs other than the restriction imposed by available memory.
- The elimination of `UNION ALL` branches on a predicate will be similar to `UNION ALL` queries with `DUAL`.

join_clause

Use the appropriate *`join_clause`* syntax to identify tables that are part of a join from which to select data. The *`inner_cross_join_clause`* lets you specify an inner or cross join. The *`outer_join_clause`* lets you specify an outer join. The *`cross_outer_apply_clause`* lets you specify a variation of an ANSI `CROSS JOIN` or an ANSI `LEFT OUTER JOIN` with left correlation support.

When you join more than two row sources, you can use parentheses to override default precedence. For example, the following syntax:

```
SELECT ... FROM a JOIN (b JOIN c) ...
```

results in a join of `b` and `c`, and then a join of that result set with `a`.

See Also:

"Joins " for more information on joins, "Using Join Queries: Examples", "Using Self Joins: Example", and "Using Outer Joins: Examples"

inner_cross_join_clause

Inner joins return only those rows that satisfy the join condition.

INNER

Specify `INNER` to explicitly specify an inner join.

JOIN

The `JOIN` keyword explicitly states that a join is being performed. You can use this syntax to replace the comma-delimited table expressions used in `WHERE` clause joins with `FROM` clause join syntax.

ON *condition*

Use the `ON` clause to specify a join condition. Doing so lets you specify join conditions separate from any search or filter conditions in the `WHERE` clause.

USING (*column*)

When you are specifying an equijoin of columns that have the same name in both tables, the `USING` *`column`* clause indicates the columns to be used. You can use this clause only if the join columns in both tables have the same name. Within this clause, do not qualify the column name with a table name or table alias.

CROSS

The `CROSS` keyword indicates that a cross join is being performed. A cross join produces the cross-product of two relations and is essentially the same as the comma-delimited Oracle Database notation.

NATURAL

The `NATURAL` keyword indicates that a natural join is being performed. Refer to NATURAL for the full semantics of this clause.

outer_join_clause

Outer joins return all rows that satisfy the join condition and also return some or all of those rows from one table for which no rows from the other satisfy the join condition. You can specify two types of outer joins: a conventional outer join using the *`table_reference`* syntax on both sides of the join, or a partitioned outer join using the *`query_partition_clause`* on one side or the other. A partitioned outer join is similar to a conventional outer join except that the join takes place between the outer table and each partition of the inner table. This type of join lets you selectively make sparse data more dense along the dimensions of interest. This process is called **data densification**.

query_partition_clause

The *`query_partition_clause`* lets you define a **partitioned outer join**. Such a join extends the conventional outer join syntax by applying the outer join to partitions returned by the query. Oracle Database creates a partition of rows for each expression you specify in the `PARTITION BY` clause. The rows in each query partition have same value for the `PARTITION BY` expression.

The *`query_partition_clause`* can be on either side of the outer join. The result of a partitioned outer join is a `UNION` of the outer joins of each of the partitions in the partitioned result set and the table on the other side of the join. This type of result is useful for filling gaps in sparse data, which simplifies analytic calculations.

If you omit this clause, then the database treats the entire table expression—everything specified in *`table_reference`*—as a single partition, resulting in a conventional outer join.

To use the *`query_partition_clause`* in an analytic function, use the upper branch of the syntax (without parentheses). To use this clause in a model query (in the *`model_column_clauses`*) or a partitioned outer join (in the *`outer_join_clause`*), use the lower branch of the syntax (with parentheses).

Restrictions on Partitioned Outer Joins

Partitioned outer joins are subject to the following restrictions:

- You can specify the *`query_partition_clause`* on either the right or left side of the join, but not both.
- You cannot specify a `FULL` partitioned outer join.
- If you specify the *`query_partition_clause`* in an outer join with an `ON` clause, then you cannot specify a subquery in the `ON` condition.

See Also:

"Using Partitioned Outer Joins: Examples"

NATURAL

The `NATURAL` keyword indicates that a natural join is being performed. A natural join is based on all columns in the two tables that have the same name. It selects rows from the two tables that have equal values in the relevant columns. If two columns with the same name do not have compatible data types, then an error is raised. When specifying columns that are involved in the natural join, do not qualify the column name with a table name or table alias.

On occasion, the table pairings in natural or cross joins may be ambiguous. For example, consider the following join syntax:

```
a NATURAL LEFT JOIN b LEFT JOIN c ON b.c1 = c.c1
```

This example can be interpreted in either of the following ways:

```
a NATURAL LEFT JOIN (b LEFT JOIN c ON b.c1 = c.c1)
(a NATURAL LEFT JOIN b) LEFT JOIN c ON b.c1 = c.c1
```

To avoid this ambiguity, you can use parentheses to specify the pairings of joined tables. In the absence of such parentheses, the database uses left associativity, pairing the tables from left to right.

Restriction on Natural Joins

You cannot specify a LOB column, columns of `ANYTYPE`, `ANYDATA`, or `ANYDATASET`, or a collection column as part of a natural join.

outer_join_type

The *`outer_join_type`* indicates the kind of outer join being performed:

- Specify `RIGHT` to indicate a right outer join.
- Specify `LEFT` to indicate a left outer join.
- Specify `FULL` to indicate a full or two-sided outer join. In addition to the inner join, rows from both tables that have not been returned in the result of the inner join will be preserved and extended with nulls.
- You can specify the optional `OUTER` keyword following `RIGHT`, `LEFT`, or `FULL` to explicitly clarify that an outer join is being performed.

ON *condition*

Use the `ON` clause to specify a join condition. Doing so lets you specify join conditions separate from any search or filter conditions in the `WHERE` clause.

Restriction on the ON *condition* Clause

You cannot specify this clause with a `NATURAL` outer join.

USING *column*

In an outer join with the `USING` clause, the query returns a single column that coalesces the two matching columns in the join. The coalesce function is as follows:

```
COALESCE (a, b) = a if a NOT NULL, else b.
```

Therefore:

- A left outer join returns all the common column values from the left table in the `FROM` clause.
- A right outer join returns all the common column values from the right table in the `FROM` clause.
- A full outer join returns all the common column values from both joined tables.

Restriction on the USING *column* Clause

The `USING` *`column`* clause is subject to the following restrictions:

- Within this clause, do not qualify the column name with a table name or table alias.
- You cannot specify a LOB column or a collection column in the `USING` *`column`* clause.
- You cannot specify this clause with a `NATURAL` outer join.

See Also:

- "Outer Joins " for additional rules and restrictions pertaining to outer joins
- *Oracle Database Data Warehousing Guide* for a complete discussion of partitioned outer joins and data densification
- "Using Outer Joins: Examples"

cross_outer_apply_clause

This clause allows you to perform a variation of an ANSI `CROSS JOIN` or an ANSI `LEFT OUTER JOIN` with left correlation support. You can specify a *`table_reference`* or *`collection_expression`* to the right of the `APPLY` keyword. The *`table_reference`* can be a table, inline view, or `TABLE` collection expression. The *`collection_expression`* can be a subquery, a column, a function, or a collection constructor. Regardless of its form, it must return a collection value—that is, a value whose type is nested table or varray. The *`table_reference`* or *`collection_expression`* can reference columns of tables defined in the `FROM` clause to the left of the `APPLY` keyword. This is called left correlation.

- Specify `CROSS APPLY` to perform a variation of an ANSI `CROSS JOIN`. Only rows from the table on the left side of the join that produce a result set from *`table_reference`* or *`collection_expression`* are returned.
- Specify `OUTER APPLY` to perform a variation of an ANSI `LEFT OUTER JOIN`. All rows from the table on the left side of the join are returned. Rows that do not produce a result set from *`table_reference`* or *`collection_expression`* have the NULL value in the corresponding column(s).

Restriction on the *cross_outer_apply_clause*

The *`table_reference`* cannot be a lateral inline view.

See Also:

Using CROSS APPLY and OUTER APPLY Joins: Examples

inline_analytic_view

An inline analytic view is a transitory analytic view that is specified in the `FROM` clause. To create an inline analytic view, use the `ANALYTIC VIEW` keyword and specify a *`sub_av_clause`* that defines the analytic view. Optionally, you may specify an *`inline_av_alias`*, which is an alias for the inline analytic view. The rules for the *`inline_av_alias`* are the same as the rules for an inline view alias.

See Also:

Analytic Views: Examples

where_clause

The `WHERE` condition lets you restrict the rows selected to those that satisfy one or more conditions. For *`condition`*, specify any valid SQL condition.

If you omit this clause, then the database returns all rows from the tables, views, or materialized views in the `FROM` clause.

Note:

If this clause refers to a `DATE` column of a partitioned table or index, then the database performs partition pruning only if:

- You created the table or index partitions by fully specifying the year using the `TO_DATE` function with a 4-digit format mask, *and*
- You specify the date in the *`where_clause`* of the query using the `TO_DATE` function and either a 2- or 4-digit format mask.

With Oracle Database 21c you can write macros for scalar expressions and use them inside the *`where_clause`* , where it would be legal to call a PLSQL function.

You must define these macro functions in PL/SQL and call them from SQL for them to function as macros.

See Also:

- Conditions for the syntax description of *`condition`*
- "Selecting from a Partition: Example"
- Defining SQL Macros

hierarchical_query_clause

The `hierarchical_query_clause` lets you select rows in a hierarchical order.

`SELECT` statements that contain hierarchical queries can contain the `LEVEL` pseudocolumn in the select list. `LEVEL` returns the value 1 for a root node, 2 for a child node of a root node, 3 for a grandchild, and so on. The number of levels returned by a hierarchical query may be limited by available user memory.

Oracle processes hierarchical queries as follows:

- A join, if present, is evaluated first, whether the join is specified in the `FROM` clause or with `WHERE` clause predicates.
- The `CONNECT BY` condition is evaluated.
- Any remaining `WHERE` clause predicates are evaluated.

If you specify this clause, then do not specify either `ORDER BY` or `GROUP BY`, because they will destroy the hierarchical order of the `CONNECT BY` results. If you want to order rows of siblings of the same parent, then use the `ORDER SIBLINGS BY` clause.

See Also:

"Hierarchical Queries " for a discussion of hierarchical queries and "Using the LEVEL Pseudocolumn: Examples"

START WITH Clause

Specify a condition that identifies the row(s) to be used as the root(s) of a hierarchical query. The `condition` can be any condition as described in Conditions. Oracle Database uses as root(s) all rows that satisfy this condition. If you omit this clause, then the database uses all rows in the table as root rows.

CONNECT BY Clause

Specify a condition that identifies the relationship between parent rows and child rows of the hierarchy. The `condition` can be any condition as described in Conditions. However, it must use the `PRIOR` operator to refer to the parent row.

See Also:

- Pseudocolumns for more information on `LEVEL`
- "Hierarchical Queries " for general information on hierarchical queries
- "Hierarchical Query: Examples"

group_by_clause

Specify the `GROUP BY` clause if you want the database to group the selected rows based on the value of `expr`(s) for each row and return a single row of summary information for each group. If this clause contains `CUBE` or `ROLLUP` extensions, then the database produces superaggregate groupings in addition to the regular groupings.

Expressions in the `GROUP BY` clause can contain any columns of the tables, views, or materialized views in the `FROM` clause, regardless of whether the columns appear in the select list.

The `GROUP BY` clause groups rows but does not guarantee the order of the result set. To order the groupings, use the `ORDER BY` clause.

If a column name in the source tables and column alias in the `SELECT` list are the same, `GROUP BY` will interpret the identifier as the column name, not the alias.

See Also:

- *Oracle Database Data Warehousing Guide* for an expanded discussion and examples of using SQL grouping syntax for data aggregation
- the GROUP_ID , GROUPING , and GROUPING_ID functions for examples
- "Using the GROUP BY Clause: Examples"
- Restrictions for Linguistic Collations for information on implications of how `GROUP BY` character values are compared linguistically
- Appendix C in *Oracle Database Globalization Support Guide* for the collation determination rules for the expressions in the `GROUP BY` clause

ROLLUP

The `ROLLUP` operation in the *`simple_grouping_clause`* groups the selected rows based on the values of the first n, n-1, n-2, ... 0 expressions in the `GROUP BY` specification, and returns a single row of summary for each group. You can use the `ROLLUP` operation to produce **subtotal values** by using it with the `SUM` function. When used with `SUM`, `ROLLUP` generates subtotals from the most detailed level to the grand total. Aggregate functions such as `COUNT` can be used to produce other kinds of superaggregates.

For example, given three expressions (n=3) in the `ROLLUP` clause of the *`simple_grouping_clause`*, the operation results in n+1 = 3+1 = 4 groupings.

Rows grouped on the values of the first *n* expressions are called **regular rows**, and the others are called **superaggregate rows**.

See Also:

Oracle Database Data Warehousing Guide for information on using `ROLLUP` with materialized views

CUBE

The `CUBE` operation in the *`simple_grouping_clause`* groups the selected rows based on the values of all possible combinations of expressions in the specification. It returns a single row of summary information for each group. You can use the `CUBE` operation to produce **cross-tabulation values**.

For example, given three expressions (n=3) in the `CUBE` clause of the *`simple_grouping_clause`*, the operation results in $2^n = 2^3 = 8$ groupings. Rows grouped on the values of *`n`* expressions are called **regular rows**, and the rest are called **superaggregate rows**.

See Also:

- *Oracle Database Data Warehousing Guide* for information on using `CUBE` with materialized views
- "Using the GROUP BY CUBE Clause: Example"

GROUPING SETS

`GROUPING SETS` are a further extension of the `GROUP BY` clause that let you specify multiple groupings of data. Doing so facilitates efficient aggregation by pruning the aggregates you do not need. You specify just the desired groups, and the database does not need to perform the full set of aggregations generated by `CUBE` or `ROLLUP`. Oracle Database computes all groupings specified in the `GROUPING SETS` clause and combines the results of individual groupings with a `UNION ALL` operation. The `UNION ALL` means that the result set can include duplicate rows.

Within the `GROUP BY` clause, you can combine expressions in various ways:

- To specify **composite columns**, group columns within parentheses so that the database treats them as a unit while computing `ROLLUP` or `CUBE` operations.
- To specify **concatenated grouping sets**, separate multiple grouping sets, `ROLLUP`, and `CUBE` operations with commas so that the database combines them into a single `GROUP BY` clause. The result is a cross-product of groupings from each grouping set.

See Also:

"Using the GROUPING SETS Clause: Example"

HAVING Clause

Use the `HAVING` clause to restrict the groups of returned rows to those groups for which the specified *`condition`* is `TRUE`. If you omit this clause, then the database returns summary rows for all groups.

Specify `GROUP BY` and `HAVING` after the *`where_clause`* and *`hierarchical_query_clause`*. If you specify both `GROUP BY` and `HAVING`, then they can appear in either order.

With Oracle Database 21c you can write macros for scalar expressions and use them inside the `HAVING` clause, where it would be legal to call a PL/SQL function.

You must define these macro functions in PL/SQL and call them from SQL for them to function as macros.

See Also:

- "Using the HAVING Condition: Example"
- Defining SQL Macros

Restrictions on the GROUP BY Clause

This clause is subject to the following restrictions:

- You cannot specify LOB columns, nested tables, or varrays as part of *expr*.
- The expressions can be of any form except scalar subquery expressions.
- If the *group_by_clause* references any object type columns, then the query will not be parallelized.
- To group by position, the parameter group_by_position_enabled must be set to true, this is false by default

model_clause

The *model_clause* lets you view selected rows as a multidimensional array and randomly access cells within that array. Using the *model_clause*, you can specify a series of cell assignments, referred to as **rules**, that invoke calculations on individual cells and ranges of cells. These rules operate on the results of a query and do not update any database tables.

When using the *model_clause* in a query, the SELECT and ORDER BY clauses must refer only to those columns defined in the *model_column_clauses*.

See Also:

- The syntax description of *expr* in "About SQL Expressions " and the syntax description of *condition* in Conditions
- *Oracle Database Data Warehousing Guide* for an expanded discussion and examples
- "The MODEL clause: Examples"

main_model

The *main_model* clause defines how the selected rows will be viewed in a multidimensional array and what rules will operate on which cells in that array.

model_column_clauses

The *model_column_clauses* define and classify the columns of a query into three groups: partition columns, dimension columns, and measure columns. For *expr*, you can specify a column, constant, host variable, single-row function, aggregate function, or any expression involving them. If *expr* is a column, then the column alias (*c_alias*) is optional. If *expr* is not a column, then the column alias is required. If you specify a column alias, then you must use the alias to refer to the column in the *model_rules_clause*, SELECT list, and the query ORDER BY clauses.

PARTITION BY

The PARTITION BY clause specifies the columns that will be used to divide the selected rows into partitions based on the values of the specified columns.

DIMENSION BY

The DIMENSION BY clause specifies the columns that will identify a row within a partition. The values of the dimension columns, along with those of the partition columns, serve as array indexes to the measure columns within a row.

MEASURES

The MEASURES clause identifies the columns on which the calculations can be performed. Measure columns in individual rows are treated like cells that you can reference, by specifying the values for the partition and dimension columns, and update.

cell_reference_options

Use the *cell_reference_options* clause to specify how null and absent values are treated in rules and how column uniqueness is constrained.

IGNORE NAV

When you specify IGNORE NAV, the database returns the following values for the null and absent values of the data type specified:

- Zero for numeric data types
- 01-JAN-2000 for datetime data types
- An empty string for character data types
- Null for all other data types

KEEP NAV

When you specify KEEP NAV, the database returns null for both null and absent cell values. KEEP NAV is the default.

UNIQUE SINGLE REFERENCE

When you specify UNIQUE SINGLE REFERENCE, the database checks only single-cell references on the right-hand side of the rule for uniqueness, not the entire query result set.

UNIQUE DIMENSION

When you specify UNIQUE DIMENSION, the database checks that the PARTITION BY and DIMENSION BY columns form a unique key to the query. UNIQUE DIMENSION is the default.

model_rules_clause

Use the *model_rules_clause* to specify the cells to be updated, the rules for updating those cells, and optionally, how the rules are to be applied and processed.

Each rule represents an assignment and consists of a left-hand side and right-hand side. The left-hand side of the rule identifies the cells to be updated by the right-hand side of the rule. The right-hand side of the rule evaluates to the values to be assigned to the cells specified on the left-hand side of the rule.

UPSERT ALL

`UPSERT ALL` allows `UPSERT` behavior for a rule with both positional and symbolic references on the left-hand side of the rule. When evaluating an `UPSERT ALL` rule, Oracle performs the following steps to create a list of cell references to be upserted:

1. Find the existing cells that satisfy all the symbolic predicates of the cell reference.
2. Using just the dimensions that have symbolic references, find the distinct dimension value combinations of these cells.
3. Perform a cross product of these value combinations with the dimension values specified by way of positional references.

Refer to *Oracle Database Data Warehousing Guide* for more information on the semantics of `UPSERT ALL`.

UPSERT

When you specify `UPSERT`, the database applies the rules to those cells referenced on the left-hand side of the rule that exist in the multidimensional array, and inserts new rows for those that do not exist. `UPSERT` behavior applies only when positional referencing is used on the left-hand side and a single cell is referenced. `UPSERT` is the default. Refer to cell_assignment for more information on positional referencing and single-cell references.

`UPDATE` and `UPSERT` can be specified for individual rules as well. When either `UPDATE` or `UPSERT` is specified for a specific rule, it takes precedence over the option specified in the `RULES` clause.

Note:

If an `UPSERT ALL`, `UPSERT`, or `UPDATE` rule does not contain the appropriate predicates, then the database may implicitly convert it to a different type of rule:

- If an `UPSERT` rule contains an existential predicate, then the rule is treated as an `UPDATE` rule.
- An `UPSERT ALL` rule must have at least one existential predicate and one qualified predicate on its left side. If it has no existential predicate, then it is treated as an `UPSERT` rule. If it has no qualified predicate, then it is treated as an `UPDATE` rule

UPDATE

When you specify `UPDATE`, the database applies the rules to those cells referenced on the left-hand side of the rule that exist in the multidimensional array. If the cells do not exist, then the assignment is ignored.

AUTOMATIC ORDER

When you specify `AUTOMATIC ORDER`, the database evaluates the rules based on their dependency order. In this case, a cell can be assigned a value once only.

SEQUENTIAL ORDER

When you specify `SEQUENTIAL ORDER`, the database evaluates the rules in the order they appear. In this case, a cell can be assigned a value more than once. `SEQUENTIAL ORDER` is the default.

ITERATE ... [UNTIL]

Use `ITERATE` ... [`UNTIL`] to specify the number of times to cycle through the rules and, optionally, an early termination condition. The parentheses around the `UNTIL` condition are optional.

When you specify `ITERATE` ... [`UNTIL`], rules are evaluated in the order in which they appear. Oracle Database returns an error if both `AUTOMATIC ORDER` and `ITERATE` ... `[UNTIL]` are specified in the *`model_rules_clause`*.

cell_assignment

The *`cell_assignment`* clause, which is the left-hand side of the rule, specifies one or more cells to be updated. When a *`cell_assignment`* references a single cell, it is called a **single-cell reference**. When more than one cell is referenced, it is called a **multiple-cell reference**.

All dimension columns defined in the *`model_clause`* must be qualified in the *`cell_assignment`* clause. A dimension can be qualified using either symbolic or positional referencing.

A **symbolic reference** qualifies a single dimension column using a Boolean condition like *`dimension_column=constant`*. A **positional reference** is one where the dimension column is implied by its position in the `DIMENSION BY` clause. The only difference between symbolic references and positional references is in the treatment of nulls.

Using a single-cell symbolic reference such as `a[x=null,y=2000]`, no cells qualify because `x=null` evaluates to `FALSE`. However, using a single-cell positional reference such as `a[null,2000]`, a cell where `x` is null and `y` is 2000 qualifies because null = null evaluates to `TRUE`. With single-cell positional referencing, you can reference, update, and insert cells where dimension columns are null.

You can specify a condition or an expression representing a dimension column value using either symbolic or positional referencing. *`condition`* cannot contain aggregate functions or the `CV` function, and *`condition`* must reference a single dimension column. *`expr`* cannot contain a subquery. Refer to "Model Expressions" for information on model expressions.

single_column_for_loop

The *`single_column_for_loop`* clause lets you specify a range of cells to be updated within a single dimension column.

The `IN` clause lets you specify the values of the dimension column as either a list of values or as a subquery. When using *`subquery`*, it cannot:

- Be a correlated query
- Return more than 10,000 rows
- Be a query defined in the `WITH` clause

The `FROM` clause lets you specify a range of values for a dimension column with discrete increments within the range. The `FROM` clause can only be used for those columns with a data type for which addition and subtraction is supported. The `INCREMENT` and `DECREMENT` values must be positive.

Optionally, you can specify the `LIKE` clause within the `FROM` clause. In the `LIKE` clause, *`pattern`* is a character string containing a single pattern-matching character `%`. This character is replaced during execution with the current incremented or decremented value in the `FROM` clause.

If all dimensions other than those used by a `FOR` loop involve a single-cell reference, then the expressions can insert new rows. The number of dimension value combinations generated by `FOR` loops is counted as part of the 10,000 row limit of the `MODEL` clause.

multi_column_for_loop

The *`multi_column_for_loop`* clause lets you specify a range of cells to be updated across multiple dimension columns. The `IN` clause lets you specify the values of the dimension columns as either multiple lists of values or as a subquery. When using *`subquery`*, it cannot:

- Be a correlated query
- Return more than 10,000 rows
- Be a query defined in the `WITH` clause

If all dimensions other than those used by a `FOR` loop involve a single-cell reference, then the expressions can insert new rows. The number of dimension value combinations generated by `FOR` loops is counted as part of the 10,000 row limit of the `MODEL` clause.

See Also:

Oracle Database Data Warehousing Guide for more information about using `FOR` loops in the `MODEL` clause

order_by_clause

Use the `ORDER BY` clause to specify the order in which cells on the left-hand side of the rule are to be evaluated. The *`expr`* must resolve to a dimension or measure column. If the `ORDER BY` clause is not specified, then the order defaults to the order of the columns as specified in the `DIMENSION BY` clause. See *order_by_clause* for more information.

Restrictions on the *order_by_clause*

Use of the `ORDER BY` clause in the model rule is subject to the following restrictions:

- You cannot specify `SIBLINGS`, *`position`*, or *`c_alias`* in the *`order_by_clause`* of the *`model_clause`*.
- You cannot specify this clause on the left-hand side of the model rule and also specify a `FOR` loop on the right-hand side of the rule.

expr

Specify an expression representing the value or values of the cell or cells specified on the right-hand side of the rule. *`expr`* cannot contain a subquery. Refer to "Model Expressions" for information on model expressions.

return_rows_clause

The *`return_rows_clause`* lets you specify whether to return all rows selected or only those rows updated by the model rules. `ALL` is the default.

reference_model

Use the *`reference_model`* clause when you need to access multiple arrays from inside the *`model_clause`*. This clause defines a read-only multidimensional array based on the results of a query.

The subclauses of the *`reference_model`* clause have the same semantics as for the *`main_model`* clause. Refer to model_column_clauses and cell_reference_options.

Restrictions on the *reference_model* Clause

This clause is subject to the following restrictions:

- `PARTITION BY` columns cannot be specified for reference models.
- The subquery of the reference model cannot refer to columns in an outer subquery.

Set Operators: (UNION, INTERSECT, MINUS, EXCEPT) ALL

The set operators combine the rows returned by two `SELECT` statements into a single result. The number and data types of the columns selected by each component query must be the same, but the column lengths can be different. The names of the columns in the result set are the names of the expressions in the select list preceding the set operator.

If you combine more than two queries with set operators, then the database evaluates adjacent queries from left to right. The parentheses around the subquery are optional. You can use them to specify a different order of evaluation.

Refer to "The Set Operators" for information on these operators, including restrictions on their use.

order_by_clause

Use the `ORDER BY` clause to order rows returned by the statement. Without an *`order_by_clause`*, no guarantee exists that the same query executed more than once will retrieve rows in the same order.

SIBLINGS

The `SIBLINGS` keyword is valid only if you also specify the *`hierarchical_query_clause`* (`CONNECT BY`). `ORDER SIBLINGS BY` preserves any ordering specified in the hierarchical query clause and then applies the *`order_by_clause`* to the siblings of the hierarchy.

expr

`expr` orders rows based on their value for *`expr`*. The expression is based on columns in the select list or columns in the tables, views, or materialized views in the `FROM` clause.

position

Specify *`position`* to order rows based on their value for the expression in this position of the select list. The *`position`* value must be an integer.

You can specify multiple expressions in the *`order_by_clause`*. Oracle Database first sorts rows based on their values for the first expression. Rows with the same value for the first expression are then sorted based on their values for the second expression, and so on. The database sorts nulls following all others in ascending order and preceding all others in descending order. Refer to "Sorting Query Results " for a discussion of ordering query results.

ASC | DESC

Specify whether the ordering sequence is ascending or descending. `ASC` is the default.

NULLS FIRST | NULLS LAST

Specify whether returned rows containing null values should appear first or last in the ordering sequence.

`NULLS LAST` is the default for ascending order, and `NULLS FIRST` is the default for descending order.

Restrictions on the ORDER BY Clause

The following restrictions apply to the ORDER BY clause:

- If you have specified the DISTINCT operator in this statement, then this clause cannot refer to columns unless they appear in the select list.
- An *order_by_clause* can contain no more than 255 expressions.
- You cannot order by a LOB, LONG, or LONG RAW column, nested table, or varray.
- If you specify a *group_by_clause* in the same statement, then this *order_by_clause* is restricted to the following expressions:
  - Constants
  - Aggregate functions
  - Analytic functions
  - The functions USER, UID, and SYSDATE
  - Expressions identical to those in the *group_by_clause*
  - Expressions comprising the preceding expressions that evaluate to the same value for all rows in a group

See Also:

- "Using the ORDER BY Clause: Examples"
- Restrictions for Linguistic Collations for information on implications of how ORDER BY character values are compared linguistically
- Appendix C in *Oracle Database Globalization Support Guide* for the collation determination rules for the expressions in the ORDER BY clause

window_clause

Oracle Database Release 21c supports the *window_clause* in the *query_block* clause.

Rules

- If you use a new *window_specification* to specify an *existing_window_name* then
  - *existing_window_name* must refer to an earlier entry in the *window_name* list
  - You cannot use *existing_window_name* with *windowing_clause*
  - You cannot define a new window with the *query_partition_clause*. If *existing_window_name* has *order_by_clause*, then the new window definition cannot have *order_by_clause*.
- Note that OVER *window_name* is not equivalent to OVER (*window_name* …). OVER (*window_name* …) implies copying and modifying the window specification, and will be rejected if the referenced window specification includes a *windowing_clause*.

Example

The following query shows the usage of *window_clause* specified as part of table expression and window functions specified using the window name as defined in window clause.

```
SELECT
      ename, mgr,
      FIRST_VALUE(sal) OVER w AS "first",
      LAST_VALUE(sal) OVER w AS "last",
      NTH_VALUE(sal, 2) OVER w AS "second",
      NTH_VALUE(sal, 4) OVER w AS "fourth"
   FROM emp
   WINDOW w AS (PARTITION BY deptno ORDER BY sal ROWS UNBOUNDED PRECEDING);
```

row_limiting_clause

The `row_limiting_clause` allows you to limit the rows returned by the query. You can specify an offset, and the number of rows or percentage of rows to return. You can use this clause to implement top-N reporting. For consistent results, specify the `order_by_clause` to ensure a deterministic sort order.

OFFSET

Use this clause to specify the number of rows to skip before row limiting begins. `offset` must be a number or an expression that evaluates to a numeric value. If you specify a negative number, then `offset` is treated as 0. If you specify NULL, or a number greater than or equal to the number of rows returned by the query, then 0 rows are returned. If `offset` includes a fraction, then the fractional portion is truncated. If you do not specify this clause, then `offset` is 0 and row limiting begins with the first row.

Restrictions

This clause is subject to the following restrictions:

- You cannot specify this clause with the `for_update_clause`.
- If you specify this clause, then the select list cannot contain the sequence pseudocolumns `CURRVAL` or `NEXTVAL`.
- Materialized views are not eligible for an incremental refresh if the defining query contains the `row_limiting_clause`.
- If the select list contains columns with identical names and you specify the `row_limiting_clause`, then an `ORA-00918` error occurs. This error occurs whether the identically named columns are in the same table or in different tables. You can work around this issue by specifying unique column aliases for the identically named columns.

fetch_clause

Use this clause to specify the number of rows or percentage of rows to return. If you do not specify this clause, then all rows are returned, beginning at row `offset` + 1.

APPROX | APPROXIMATE | EXACT

Specify `EXACT` to limit results as specified exactly.

Specify `APPROX` or `APPROXIMATE` to perform approximate vector search.

The two keywords `APPROX` and `APPROXIMATE` are synonyms. If you specify neither of them, the default is `APPROXIMATE`. However, approximate vector search can only be performed when all syntax and semantic rules are satisfied, the corresponding vector index is available, and the query optimizer determines to perform it. If any of these conditions are unmet, then an approximate search is not performed. In this case the query returns exact results.

Syntax and Semantic Rules for an Approximate Vector Search

- `row_limiting_partition_clause` must not be specified.
- `OFFSET` must not be specified.

- *percent* PERCENT (of *row_specification*, not *accuracy* PERCENT of *accuracy_clause*) must not be specified.
- WITH TIES must not be specified .
- The approximate row limiting clause must be associated with an ORDER BY clause.
- The first key of the ORDER BY must be a distance function (VECTOR_DISTANCE or variant), which must have one and only one vector column operand.
- There may be additional ORDER BY expressions after the distance function, but not before.

FIRST | NEXT

These keywords can be used interchangeably and are provided for semantic clarity.

row_limiting_partition_clause

You can specify one or more levels of partitions in *partition_count* to apply row limiting within each partition or each combination of all levels of partitions.

You cannot use this clause with OFFSET, *percent* PERCENT, or WITH TIES.

You may specify unlimited levels of partitions. For each partition level, the following rules apply:

- *partition_countX* must be a number or an expression that evaluates to a numeric value. It can be given as a constant literal, a bind, a non-scalar subquery, or a correlated variable. Otherwise an error is raised.
- If a negative number is specified, then it is treated as 0.
- If *partition_countX* is greater than the number of partitions available in this level, then certain rows from all available partitions in this level are returned.
- If *partition_countX* includes a fraction, then the fractional portion is truncated.
- If *partition_countX* in any level is NULL, then 0 rows are returned.
- *partition_by_exprX* must be constants, columns, nonanalytic functions, function expressions, or expressions involving any of these.

Given that the query result may be sorted in certain order, partitioned row limiting clause filters out records so that only records that meet the following conditions are returned:

- the record has *partition_by_expr1* being one of the top *partition_count1* values of *partition_by_expr1*
- within the same *partition_by_expr1*, the record has *partition_by_expr2* being one of the top *partition_count2* values of *partition_by_expr2*
- within the same *partition_by_expr1* and *partition_by_expr2*, the record has *partition_by_expr3* being one of the top *partition_count3* values of *partition_by_expr3*
- the same logic applies to all levels of partitions
- within the nested partition of *partition_by_expr1*, ..., *partition_by_exprN*, the record is the top *rowcount* rows.

The keywords PARTITION BY or PARTITIONS BY are optional as long as there is no semantic ambiguity when they are missing.

row_specification

***rowcount* | *percent* PERCENT**

Use *rowcount* to specify the number of rows to return. *rowcount* must be a number or an expression that evaluates to a numeric value. If you specify a negative number, then *rowcount* is treated as 0. If *rowcount* is greater than the number of rows available beginning at row *offset* + 1, then all available rows are returned. If *rowcount* includes a fraction, then the fractional portion is truncated. If *rowcount* is NULL, then 0 rows are returned.

Use *percent* PERCENT to specify the percentage of the total number of selected rows to return. *percent* must be a number or an expression that evaluates to a numeric value. If you specify a negative number, then *percent* is treated as 0. If *percent* is NULL, then 0 rows are returned.

If you do not specify *rowcount* or *percent* PERCENT, then 1 row is returned.

ROW | ROWS

Specify one of ROW or ROWS. These keywords can be used interchangeably and are provided for semantic clarity.

If any of these conditions are not met, an exact search will be performed even though the APPROXIMATE syntax is used. In addition, even if all the conditions are met, the optimizer may employ other cost-based decisions and choose not to use the index and perform exact search .

Example: Vector Search Query

```
SELECT docID FROM vec_table
ORDER BY VECTOR_DISTANCE(data, :query_vec)
FETCH APPROX FIRST 20 ROWS ONLY;
```

You can use this clause in vector and non-vector contexts. See examples Partitioned Row Limiting in Non-Vector Context: Example and Partitioned Row Limiting in a Multi-Vector Search: Example .

ONLY | WITH TIES

Specify ONLY to return exactly the specified number of rows or percentage of rows.

Specify WITH TIES to return additional rows with the same sort key as the last row fetched. WITH TIES must be specified with *order_by_clause* . If you do not specify the *order_by_clause*, then no additional rows will be returned.

You cannot use WITH TIES for approximate vector search and partition row limit. If you specify it, approximate search will not happen, or if there are partitions, the statement will fail.

See Also:

"Row Limiting: Examples"

accuracy_clause

Specify a value or certain parameters to tune the accuracy of the approximate vector search. If approximate vector search is not performed for any reason, this clause is ignored.

Rules

- Keywords WITH, TARGET, and PERCENT are optional and used for semantic clarity. There is no impact on the query's semantic if you choose not to specify these keywords.

- `accuracy` must be a number or an expression that evaluates to a numeric value between 1 and 100.
- In the case where a vector index is used, the accuracy, if specified, overwrites the index specification, otherwise it inherits the index specification. In the case where no vector index is used, exact results are returned, and the accuracy is meaningless.
- `PARAMETERS` *`efs`* and *`nprobes`* must be a number or an expression that evaluates to a numeric value.

for_update_clause

The `FOR UPDATE` clause lets you lock the selected rows so that other users cannot lock or update the rows until you end your transaction. You can specify this clause only in a top-level `SELECT` statement, not in subqueries.

> **Note:**
>
> Prior to updating a LOB value, you must lock the row containing the LOB. One way to lock the row is with an embedded `SELECT` ... `FOR UPDATE` statement. You can do this using one of the programmatic languages or `DBMS_LOB` package. For more information on lock rows before writing to a LOB, see *Oracle Database SecureFiles and Large Objects Developer's Guide*.

Nested table rows are not locked as a result of locking the parent table rows. If you want the nested table rows to be locked, then you must lock them explicitly.

Restrictions on the FOR UPDATE Clause

This clause is subject to the following restrictions:

- You cannot specify this clause with the following other constructs: the `DISTINCT` operator, `CURSOR` expression, set operators, *`group_by_clause`*, or aggregate functions.
- The tables locked by this clause must all be located on the same database and on the same database as any `LONG` columns and sequences referenced in the same statement.

> **See Also:**
>
> "Using the FOR UPDATE Clause: Examples"

Using the FOR UPDATE Clause on Views

In general, this clause is not supported on views. However, in some cases, a `SELECT` ... `FOR UPDATE` query on a view can succeed without any errors. This occurs when the view has been merged to its containing query block internally by the query optimizer, and `SELECT` ... `FOR UPDATE` succeeds on the internally transformed query. The examples in this section illustrate when using the `FOR UPDATE` clause on a view can succeed or fail.

- Using the `FOR UPDATE` clause on merged views

  An error can occur when you use the `FOR UPDATE` clause on a merged view if both of the following conditions apply:

  - The underlying column of the view is an expression
  - The FOR UPDATE clause applies to a column list

  The following statement succeeds because the underlying column of the view is not an expression:

```
SELECT employee_id FROM (SELECT * FROM employees)
   FOR UPDATE OF employee_id;
```

  The following statement succeeds because, while the underlying column of the view is an expression, the FOR UPDATE clause does not apply to a column list:

```
SELECT employee_id FROM (SELECT employee_id+1 AS employee_id FROM employees)
   FOR UPDATE;
```

  The following statement fails because the underlying column of the view is an expression and the FOR UPDATE clause applies to a column list:

```
SELECT employee_id FROM (SELECT employee_id+1 AS employee_id FROM employees)
   FOR UPDATE OF employee_id;
                 *
Error at line 2:
ORA-01733: virtual column not allowed here
```

- Using the FOR UPDATE clause on non-merged views

  Since the FOR UPDATE clause is not supported on views, anything that prevents view merging, such as the NO_MERGE hint, parameters that disallow view merging, or something in the query structure that prevents view merging, will result in an ORA-02014 error.

  In the following example, the GROUP BY statement prevents view merging, which causes an error:

```
SELECT avgsal
   FROM (SELECT AVG(salary) AS avgsal FROM employees GROUP BY job_id)
   FOR UPDATE;
FROM (SELECT AVG(salary) AS avgsal FROM employees GROUP BY job_id)
     *
ERROR at line 2:
ORA-02014: cannot select FOR UPDATE from view with DISTINCT, GROUP BY, etc.
```

Note:

Due to the complexity of the view merging mechanism, Oracle recommends against using the FOR UPDATE clause on views.

OF ... *column*

Use the OF ... *column* clause to lock the select rows only for a particular table or view in a join. The columns in the OF clause only indicate which table or view rows are locked. The specific columns that you specify are not significant. However, you must specify an actual column name, not a column alias. If you omit this clause, then the database locks the selected rows from all the tables in the query.

NOWAIT | WAIT

The NOWAIT and WAIT clauses let you tell the database how to proceed if the SELECT statement attempts to lock a row that is locked by another user.

- Specify `NOWAIT` to return control to you immediately if a lock exists.
- Specify `WAIT` to instruct the database to wait *`integer`* seconds for the row to become available and then return control to you.

If you specify neither `WAIT` nor `NOWAIT`, then the database waits until the row is available and then returns the results of the `SELECT` statement.

SKIP LOCKED

`SKIP LOCKED` is an alternative way to handle a contending transaction that is locking some rows of interest. Specify `SKIP LOCKED` to instruct the database to attempt to lock the rows specified by the `WHERE` clause and to skip any rows that are found to be already locked by another transaction. This feature is designed for use in multiconsumer queue environments. It enables queue consumers to skip rows that are locked by other consumers and obtain unlocked rows without waiting for the other consumers to finish. Refer to *Oracle Database Advanced Queuing User's Guide* for more information.

Note on the WAIT and SKIP LOCKED Clauses

If you specify `WAIT` or `SKIP LOCKED` and the table is locked in exclusive mode, then the database will not return the results of the `SELECT` statement until the lock on the table is released. In the case of `WAIT`, the `SELECT FOR UPDATE` clause is blocked regardless of the wait time specified.

row_pattern_clause

The `MATCH_RECOGNIZE` clause lets you perform pattern matching. Use this clause to recognize patterns in a sequence of rows in *`table`*, which is called the row pattern input table. The result of a query that uses the `MATCH_RECOGNIZE` clause is called the row pattern output table.

The `MATCH_RECOGNIZE` enables you to do the following tasks:

- Logically partition and order the data with the `PARTITION BY` and `ORDER BY` clauses.
- Define measures, which are expressions usable in other parts of the SQL query, in the `MEASURES` clause.
- Define patterns of rows to seek using the `PATTERN` clause. These patterns use regular expression syntax, a powerful and expressive feature, applied to the pattern variables you define.
- Specify the logical conditions required to map a row to a row pattern variable in the `DEFINE` clause.

See Also:

- *Oracle Database Data Warehousing Guide* for more information on pattern matching
- "Row Pattern Matching: Example"

row_pattern_partition_by

Specify `PARTITION BY` to divide the rows in the row pattern input table into logical groups called row pattern partitions. Use *`column`* to specify one or more partitioning columns. Each partition consists of the set of rows in the row pattern input table that have the same value(s) on the partitioning column(s).

If you specify this clause, then matches are found within partitions and do not cross partition boundaries. If you do not specify this clause, then all rows of the row input table constitute a single row pattern partition.

row_pattern_order_by

Specify ORDER BY to order rows within each row pattern partition. Use *column* to specify one or more ordering columns. If you specify multiple columns, then Oracle Database first sorts rows based on their values for the first column. Rows with the same value for the first column are then sorted based on their values for the second column, and so on. Oracle Database sorts nulls following all others in ascending order.

If you do not specify this clause, then the result of the *row_pattern_clause* is nondeterministic and you may get inconsistent results each time you run the query.

row_pattern_measures

Use the MEASURES clause to define one or more row pattern measure columns. These columns are included in the row pattern output table and contain values that are useful for analyzing data.

When you define a row pattern measure column, using the *row_pattern_measure_column* clause, you specify its pattern measure expression. The values in the column are calculated by evaluating the pattern measure expression whenever a match is found.

row_pattern_measure_column

Use this clause to define a row pattern measure column.

- For *expr*, specify the pattern measure expression. A pattern measure expression is an expression as described in Expressions that can contain only the following elements:
  - Constants: Text literals and numeric literals
  - References to any column of the row pattern input table
  - The CLASSIFIER function, which returns the name of the primary row pattern variable to which the row is mapped. Refer to row_pattern_classifier_func for more information.
  - The MATCH_NUMBER function, which returns the sequential number of a row pattern match within the row pattern partition. Refer to row_pattern_match_num_func for more information.
  - Row pattern navigation functions: PREV, NEXT, FIRST, and LAST. Refer to row_pattern_navigation_func for more information.
  - Row pattern aggregate functions: AVG , COUNT , MAX , MIN , or SUM . Refer to row_pattern_aggregate_func for more information.
- For *c_alias*, specify the alias for the pattern measure expression. Oracle Database uses this alias in the column heading of the row pattern output table. The AS keyword is optional. The alias can be used in other parts of the query, such as the SELECT ... ORDER BY clause.

row_pattern_rows_per_match

This clause lets you specify whether the row pattern output table includes summary or detailed data about each match.

- If you specify ONE ROW PER MATCH, then each match produces one summary row. This is the default.

- If you specify `ALL ROWS PER MATCH`, then each match that spans multiple rows will produce one output row for each row in the match.

row_pattern_skip_to

This clause lets you specify the point to resume row pattern matching after a non-empty match is found.

- Specify `AFTER MATCH SKIP TO NEXT ROW` to resume pattern matching at the row after the first row of the current match.
- Specify `AFTER MATCH SKIP PAST LAST ROW` to resume pattern matching at the next row after the last row of the current match. This is the default.
- Specify `AFTER MATCH SKIP TO FIRST` *`variable_name`* to resume pattern matching at the first row that is mapped to pattern variable *`variable_name`*. The *`variable_name`* must be defined in the `DEFINE` clause.
- Specify `AFTER MATCH SKIP TO LAST` *`variable_name`* to resume pattern matching at the last row that is mapped to pattern variable *`variable_name`*. The *`variable_name`* must be defined in the `DEFINE` clause.
- `AFTER MATCH SKIP TO` *`variable_name`* has the same behavior as `AFTER MATCH SKIP TO LAST` *`variable_name`*.

See Also:

Oracle Database Data Warehousing Guide for more information on the `AFTER MATCH SKIP` clauses

PATTERN

Use the `PATTERN` clause to define which pattern variables must be matched, the sequence in which they must be matched, and the quantity of rows that must be matched for each pattern variable.

A row pattern match consists of a set of contiguous rows in a row pattern partition. Each row of the match is mapped to a pattern variable. The mapping of rows to pattern variables must conform to the regular expression specified in the *`row_pattern`* clause, and all conditions in the `DEFINE` clause must be true.

Note:

It is outside the scope of this document to explain regular expression concepts and details. If you are not familiar with regular expressions, then you are encouraged to familiarize yourself with the topic using other sources.

The precedence of the elements that you specify in the regular expression of the `PATTERNS` clause, in decreasing order, is as follows:

- Row pattern elements (specified in the *`row_pattern_primary`* clause)
- Row pattern quantifiers (specified in the *`row_pattern_quantifier`* clause)

- Concatenation (specified in the *`row_pattern_term`* clause)
- Alternation (specified in the *`row_pattern`* clause)

> **See Also:**
>
> *Oracle Database Data Warehousing Guide* for more information on the `PATTERN` clause

row_pattern

Use this clause to specify the row pattern. A row pattern is a regular expression that can take one of the following forms:

- A single row pattern term

  For example: `PATTERN (A)`
- A row pattern, a vertical bar, and a row pattern term

  For example: `PATTERN (A|B)`
- A recursively built row pattern, a vertical bar, and a row pattern term

  For example: `PATTERN (A|B|C)`

The vertical bar in this clause represents **alternation**. Alternation matches a single regular expression from a list of several possible regular expressions. Alternatives are preferred in the order they are specified. For example, if you specify `PATTERN (A|B|C)`, then Oracle Database attempts to match `A` first. If `A` is not matched, then it attempts to match `B`. If `B` is not matched, then it attempts to match `C`.

row_pattern_term

This clause lets you specify a row pattern term. A row pattern term can take one of the following forms:

- A single row pattern factor

  For example: `PATTERN (A)`
- A row pattern term followed by a row pattern factor.

  For example: `PATTERN (A B)`
- A recursively built row pattern term followed by a row pattern factor

  For example: `PATTERN (A B C)`

The syntax used in the second and third examples represents **concatenation**. Concatenation is used to list two or more items in a pattern to be matched and the order in which they are to be matched. For example, if you specify `PATTERN (A B C)`, then Oracle Database first matches `A`, then uses the resulting matched rows to match `B`, then uses the resulting matched rows to match `C`. Only rows that match `A`, `B`, and `C`, are included in the row pattern match.

row_pattern_factor

This clause lets you specify a row pattern factor. A row pattern factor consists of a row pattern element, specified using the *`row_pattern_primary`* clause, and an optional row pattern quantifier, specified using the *`row_pattern_quantifier`* clause.

row_pattern_primary

Use this clause to specify the row pattern element. Table 19-1 lists the valid row pattern elements and their descriptions.

Table 19-1 Row Pattern Elements

Row Pattern Element	Description
variable_name	Specify a primary pattern variable name that is defined in the *row_pattern_definition* clause. You cannot specify a union pattern variable that is defined in the *row_pattern_subset_item* clause.
`$`	`$` matches the position after the last row in the partition. This element is an anchor. Anchors work in terms of positions rather than rows.
`^`	`^` matches the position before the first row in the partition. This element is an anchor. Anchors work in terms of positions rather than rows
`( [`*row_pattern*`] )`	Use *row_pattern* to specify the row pattern to be matched. An empty pattern `()` matches an empty set of rows.
`{-` *row_pattern* `-}`	Exclusion syntax. Use *row_pattern* to specify parts of the pattern to be excluded from the output of `ALL ROWS PER MATCH`.
row_pattern_permute	Use *row_pattern_permute* to specify a pattern that is a permutation of row pattern elements. Refer to row_pattern_permute for the full semantics of this clause.

row_pattern_permute

Use the `PERMUTE` clause to express a pattern that is a permutation of the specified row pattern elements. For example, `PATTERN (PERMUTE (A, B, C))` is equivalent to an alternation of all permutations of the three row pattern elements `A`, `B`, and `C`, similar to the following:

```
PATTERN (A B C | A C B | B A C | B C A | C A B | C B A)
```

Note that the row pattern elements are expanded lexicographically and that each element to permute must be separated by a comma from the other elements.

See Also:

Oracle Database Data Warehousing Guide for more information on permutations

row_pattern_quantifier

Use this clause to specify the row pattern quantifier, which is a postfix operator that defines the number of iterations accepted for a match.

Row pattern quantifiers are referred to as greedy; they will attempt to match as many instances of the regular expression on which they are applied as possible. The exception is row pattern quantifiers that have a question mark (`?`) as a suffix, which are referred to as reluctant. They will attempt to match as few instances as possible of the regular expression on which they are applied.

Table 19-2 lists the valid row pattern quantifiers and the number of iterations they accept for a match. In this table, *n* and *m* represent unsigned integers.

Table 19-2 Row Pattern Quantifiers

Row Pattern Quantifier	Number of Iterations Accepted for a Match
`*`	0 or more iterations (greedy)
`*?`	0 or more iterations (reluctant)
`+`	1 or more iterations (greedy)
`+?`	1 or more iterations (reluctant)
`?`	0 or 1 iterations (greedy)
`??`	0 or 1 iterations (reluctant)
`{n,}`	*n* or more iterations, (*n* >= 0) (greedy)
`{n,}?`	*n* or more iterations, (*n* >= 0) (reluctant)
`{n,m}`	Between *n* and *m* iterations, inclusive, (0 <= *n* <= *m*, 0 < *m*) (greedy)
`{n,m}?`	Between *n* and *m* iterations, inclusive, (0 <= *n* <= *m*, 0 < *m*) (reluctant)
`{,m}`	Between 0 and *m* iterations, inclusive (*m* > 0) (greedy)
`{,m}?`	Between 0 and *m* iterations, inclusive (*m* > 0) (reluctant)
`{n}?`	*n* iterations, (*n* > 0)

See Also:

Oracle Database Data Warehousing Guide for more information on row pattern quantifiers

row_pattern_subset_clause

The `SUBSET` clause lets you specify one or more union row pattern variables. Use the *`row_pattern_subset_item`* clause to declare each union row pattern variable.

You can specify union row pattern variables in the following clauses:

- `MEASURES` clause: In the expression for a row pattern measure column. That is, in expression *`expr`* of the *`row_pattern_measure_column`* clause.
- `DEFINE` clause: In the condition that defines a primary pattern variable. That is, in *`condition`* of the *`row_pattern_definition`* clause

row_pattern_subset_item

This clause lets you create a grouping of multiple pattern variables that can be referred to with a variable name of its own. The variable name that refers to this grouping is called a union row pattern variable.

- For *`variable_name`* on the left side of the equal sign, specify the name of the union row pattern variable.
- On the right side of the equal sign, specify a comma-separated list of distinct primary row pattern variables within parentheses. This list cannot include any union row pattern variables.

See Also:

Oracle Database Data Warehousing Guide for more information on defining union row pattern variables

DEFINE

Use the `DEFINE` clause to specify one or more row pattern definitions. A row pattern definition specifies the conditions that a row must meet in order to be mapped to a specific pattern variable.

The `DEFINE` clause only supports running semantics.

See Also:

- *Oracle Database Data Warehousing Guide* for more information on the `DEFINE` clause
- *Oracle Database Data Warehousing Guide* for more information on running and final semantics

row_pattern_definition_list

This clause lets you specify one or more row pattern definitions.

row_pattern_definition

This clause lets you specify a row pattern definition, which contains the conditions that a row must meet in order to be mapped to the specified pattern variable.

- For *`variable_name`*, specify the name of the pattern variable.
- For *`condition`*, specify a condition as described in Conditions, with the following extension: *`condition`* can contain any of the functions described by *row_pattern_navigation_func*::= and *row_pattern_aggregate_func*::=.

row_pattern_rec_func

This clause comprises the following clauses, which let you specify row pattern recognition functions:

- *`row_pattern_classifier_func`*: Use this clause to specify the `CLASSIFIER` function, which returns a character string whose value is the name of the variable to which the row is mapped.
- *`row_pattern_match_num_func`*: Use this clause to specify the `MATCH_NUMBER` function, which returns a numeric value with scale 0 (zero) whose value is the sequential number of the match within the row pattern partition.
- *`row_pattern_navigation_func`*: Use this clause to specify functions that perform row pattern navigation operations.
- *`row_pattern_aggregate_func`*: Use this clause to specify an aggregate function in the expression for a row pattern measure column or in the condition that defines a primary pattern variable.

You can specify row pattern recognition functions in the following clauses:

- `MEASURES` clause: In the expression for a row pattern measure column. That is, in expression *`expr`* of the *`row_pattern_measure_column`* clause.
- `DEFINE` clause: In the condition that defines a primary pattern variable. That is, in *`condition`* of the *`row_pattern_definition`* clause

A row pattern recognition function may behave differently depending whether you specify it in the `MEASURES` or `DEFINE` clause. These details are explained in the semantics for each clause.

row_pattern_classifier_func

The `CLASSIFIER` function returns a character string whose value is the name of the variable to which the row is mapped.

- In the `MEASURES` clause:
  - If you specify `ONE ROW PER MATCH`, then the query uses the last row of the match when processing the `MEASURES` clause, so the `CLASSIFIER` function returns the name of the pattern variable to which the last row of the match is mapped.
  - If you specify `ALL ROWS PER MATCH`, then for each row of the match found, the `CLASSIFIER` function returns the name of the pattern variable to which the row is mapped.

  For empty matches—that is, matches that contain no rows, the `CLASSIFER` function returns NULL.
- In the `DEFINE` clause, the `CLASSIFIER` function returns the name of the primary pattern variable to which the current row is mapped.

row_pattern_match_num_func

The `MATCH_NUMBER` function returns a numeric value with scale 0 (zero) whose value is the sequential number of the match within the row pattern partition.

Matches within a row pattern partition are numbered sequentially starting with 1 in the order in which they are found. If multiple rows satisfy a match, then they are all assigned the same match number. Note that match numbering starts over again at 1 in each row pattern partition, because there is no inherent ordering between row pattern partitions.

- In the `MEASURES` clause: You can use `MATCH_NUMBER` to obtain the sequential number of the match within the row pattern.
- In the `DEFINE` clause: You can use `MATCH_NUMBER` to define conditions that depend upon the match number.

row_pattern_navigation_func

This clause lets you perform the following row pattern navigation operations:

- Navigate among the group of rows mapped to a pattern variable using the `FIRST` and `LAST` functions of the *`row_pattern_nav_logical`* clause.
- Navigate among all rows in a row pattern partition using the `PREV` and `NEXT` functions of the *`row_pattern_nav_physical`* clause
- Nest the `FIRST` or `LAST` function within the `PREV` or `NEXT` function using the *`row_pattern_nav_compound`* clause.

row_pattern_nav_logical

This clause lets you use the `FIRST` and `LAST` functions to navigate among the group of rows mapped to a pattern variable using an optional logical offset.

- The `FIRST` function returns the value of expression *`expr`* when evaluated in the first row of the group of rows mapped to the pattern variable that is specified in *`expr`*. If no rows are mapped to the pattern variable, then the `FIRST` function returns NULL.
- The `LAST` function returns the value of expression *`expr`* when evaluated in the last row of the group of rows mapped to the pattern variable that is specified in *`expr`*. If no rows are mapped to the pattern variable, then the `LAST` function returns NULL.
- Use *`expr`* to specify the expression to be evaluated. It must contain at least one row pattern column reference. If it contains more than one row pattern column reference, then all must refer to the same pattern variable.
- Use the optional *`offset`* to specify the logical offset within the set of rows mapped to the pattern variable. When specified with the `FIRST` function, the offset is the number of rows from the first row, in ascending order. When specified with the `LAST` function, the offset is the number of rows from the last row in descending order. The default offset is 0.

  For *`offset`*, specify a non-negative integer. It must be a runtime constant (literal, bind variable, or expressions involving them), but not a column or subquery.

  If you specify an *`offset`* that is greater than or equal to the number of rows mapped to the pattern variable minus 1, then the function returns NULL.

You can specify running or final semantics for the `FIRST` and `LAST` functions as follows:

- The `MEASURES` clause supports running and final semantics. Specify `RUNNING` for running semantics. Specify `FINAL` for final semantics. The default is `RUNNING`.
- The `DEFINE` clause supports only running semantics. Therefore, running semantics will be used whether you specify or omit `RUNNING`. You cannot specify `FINAL`.

See Also:

– *Oracle Database Data Warehousing Guide* for more information on the `FIRST` and `LAST` functions

– *Oracle Database Data Warehousing Guide* for more information on running and final semantics

row_pattern_nav_physical

This clause lets you use the `PREV` and `NEXT` functions to navigate all rows in a row pattern partition using an optional physical offset.

- The `PREV` function returns the value of expression *`expr`* when evaluated in the previous row in the partition. If there is no previous row in the partition, then the `PREV` function returns NULL.
- The `NEXT` function returns the value of expression *`expr`* when evaluated in the next row in the partition. If there is no next row in the partition, then the NEXT function returns NULL.
- Use *`expr`* to specify the expression to be evaluated. It must contain at least one row pattern column reference. If it contains more than one row pattern column reference, then all must refer to the same pattern variable.

- Use the optional *offset* to specify the physical offset within the partition. When specified with the PREV function, it is the number of rows before the current row. When specified with the NEXT function, it is the number of rows after the current row. The default is 1. If you specify an offset of 0, then the current row is evaluated.

  For *offset*, specify a non-negative integer. It must be a runtime constant (literal, bind variable, or expressions involving them), but not a column or subquery.

The PREV and NEXT functions always use running semantics. Therefore, you cannot specify the RUNNING or FINAL keywords with this clause.

See Also:

- *Oracle Database Data Warehousing Guide* for more information on the PREV and NEXT functions
- *Oracle Database Data Warehousing Guide* for more information on running and final semantics

row_pattern_nav_compound

This clause lets you nest the *row_pattern_nav_logical* clause within the *row_pattern_nav_physical* clause. That is, it lets you nest the FIRST or LAST function within the PREV or NEXT function. The *row_pattern_nav_logical* clause is evaluated first and then the result is supplied to the *row_pattern_nav_physical* clause.

Refer to row_pattern_nav_logical and row_pattern_nav_physical for the full semantics of these clauses.

See Also:

Oracle Database Data Warehousing Guide for more information on nesting the FIRST and LAST functions within the PREV and NEXT functions

row_pattern_aggregate_func

This clause lets you use an aggregate function in the expression for a row pattern measure column or in the condition that defines a primary pattern variable.

For *aggregate_function*, specify any one of the AVG , COUNT , MAX , MIN , or SUM functions. The DISTINCT keyword is not supported.

You can specify running or final semantics for aggregate functions as follows:

- The MEASURES clause supports running and final semantics. Specify RUNNING for running semantics. Specify FINAL for final semantics. The default is RUNNING.
- The DEFINE clause supports only running semantics. Therefore, running semantics will be used whether you specify or omit RUNNING. You cannot specify FINAL.

See Also:

- *Oracle Database Data Warehousing Guide* for more information on aggregate functions
- *Oracle Database Data Warehousing Guide* for more information on running and final semantics

Examples

SQL Macros - Scalar Valued Macros: Examples

Print Hello <name>

A PL/SQL function `greet` is defined as a scalar SQL Macro that returns the string 'Hello, <name>! ' when called from a SQL `SELECT` statement.

```
create or replace function greet(name varchar2 default 'World')
                  return varchar2 SQL_MACRO(Scalar) is
begin
  return q'{ 'Hello, ' || name || '!' }';
end;
/
```

You can call `greet` in two ways:

Option 1: Without passing an explicit argument . In this case the default argument is used and 'Hello World' is returned.

```
SELECT greet ('World') from dual;
----------------
Hello, World!
```

Option 2: Passing an explicit argument . In this case the argument passed is used and 'Hello Bob' is returned.

```
SELECT greet ('Bob') from dual;
----------------
Hello, Bob!
```

Split String Based on Delimiter

The PL/SQL function `split_part` splits a string on the specified delimiter and returns the part at the specified position.

```
create or replace function split_part(string    varchar2,
                                      delimiter varchar2,
                                      position  pls_integer)
          return varchar2 SQL_MACRO(Scalar) is
begin
  return q'{
    regexp_substr(replace(string, delimiter||delimiter, delimiter||' '||delimiter),
                  '[^'||delimiter||']+', 1, position, 'imx')
  }';
end;
/
SELECT split_part( sysdate, '-', 2) month from dual;
    --------------
    MONTH
    -----
    OCT
```

SQL Macros - Table Valued Macros: Examples

The macro function `budget` computes the amount of each department's budget for a given job. It returns the number of employees in each department with the specified job title.

```
create or replace function budget(job varchar2) return varchar2 SQL_MACRO is
begin
  return q'{
     select deptno, sum(sal) budget
     from emp
     where job = budget.job
     group by deptno
  }';
end;
/


SELECT * FROM budget ('MANAGER');
   DEPTNO     BUDGET
-----------  --------
     20       2975
     30       2850
     10       2450
```

Using a PL/SQL Function in the WITH Clause: Examples

The following example declares and defines a PL/SQL function `get_domain` in the `WITH` clause. The `get_domain` function returns the domain name from a URL string, assuming that the URL string has the "`www`" prefix immediately preceding the domain name, and the domain name is separated by dots on the left and right. The `SELECT` statement uses `get_domain` to find distinct catalog domain names from the `orders` table in the `oe` schema.

```
WITH
 FUNCTION get_domain(url VARCHAR2) RETURN VARCHAR2 IS
   pos BINARY_INTEGER;
   len BINARY_INTEGER;
 BEGIN
   pos := INSTR(url, 'www.');
   len := INSTR(SUBSTR(url, pos + 4), '.') - 1;
   RETURN SUBSTR(url, pos + 4, len);
 END;
SELECT DISTINCT get_domain(catalog_url)
  FROM product_information;
/
```

Subquery Factoring: Example

The following statement creates the query names `dept_costs` and `avg_cost` for the initial query block containing a join, and then uses the query names in the body of the main query.

```
WITH
   dept_costs AS (
      SELECT department_name, SUM(salary) dept_total
         FROM employees e, departments d
         WHERE e.department_id = d.department_id
      GROUP BY department_name),
   avg_cost AS (
      SELECT SUM(dept_total)/COUNT(*) avg
      FROM dept_costs)
```

```
SELECT * FROM dept_costs
   WHERE dept_total >
      (SELECT avg FROM avg_cost)
      ORDER BY department_name;

DEPARTMENT_NAME                DEPT_TOTAL
------------------------------ ----------
Sales                              304500
Shipping                           156400
```

Recursive Subquery Factoring: Examples

The following statement shows the employees who directly or indirectly report to employee 101 and their reporting level.

```
WITH
  reports_to_101 (eid, emp_last, mgr_id, reportLevel) AS
  (
     SELECT employee_id, last_name, manager_id, 0 reportLevel
     FROM employees
     WHERE employee_id = 101
   UNION ALL
     SELECT e.employee_id, e.last_name, e.manager_id, reportLevel+1
     FROM reports_to_101 r, employees e
     WHERE r.eid = e.manager_id
  )
SELECT eid, emp_last, mgr_id, reportLevel
FROM reports_to_101
ORDER BY reportLevel, eid;

       EID EMP_LAST                      MGR_ID REPORTLEVEL
---------- ------------------------- ---------- -----------
       101 Kochhar                          100           0
       108 Greenberg                        101           1
       200 Whalen                           101           1
       203 Mavris                           101           1
       204 Baer                             101           1
       205 Higgins                          101           1
       109 Faviet                           108           2
       110 Chen                             108           2
       111 Sciarra                          108           2
       112 Urman                            108           2
       113 Popp                             108           2
       206 Gietz                            205           2
```

The following statement shows employees who directly or indirectly report to employee 101, their reporting level, and their management chain.

```
WITH
  reports_to_101 (eid, emp_last, mgr_id, reportLevel, mgr_list) AS
  (
     SELECT employee_id, last_name, manager_id, 0 reportLevel,
            CAST(manager_id AS VARCHAR2(2000))
     FROM employees
     WHERE employee_id = 101
  UNION ALL
     SELECT e.employee_id, e.last_name, e.manager_id, reportLevel+1,
            CAST(mgr_list || ',' || manager_id AS VARCHAR2(2000))
     FROM reports_to_101 r, employees e
     WHERE r.eid = e.manager_id
  )
SELECT eid, emp_last, mgr_id, reportLevel, mgr_list
FROM reports_to_101
```

```
ORDER BY reportLevel, eid;

        EID EMP_LAST                         MGR_ID REPORTLEVEL MGR_LIST
 ---------- ------------------------- ---------- ----------- --------
       101 Kochhar                          100           0  100
       108 Greenberg                        101           1  100,101
       200 Whalen                           101           1  100,101
       203 Mavris                           101           1  100,101
       204 Baer                             101           1  100,101
       205 Higgins                          101           1  100,101
       109 Faviet                           108           2  100,101,108
       110 Chen                             108           2  100,101,108
       111 Sciarra                          108           2  100,101,108
       112 Urman                            108           2  100,101,108
       113 Popp                             108           2  100,101,108
       206 Gietz                            205           2  100,101,205
```

The following statement shows the employees who directly or indirectly report to employee 101 and their reporting level. It stops at reporting level 1.

```
WITH
  reports_to_101 (eid, emp_last, mgr_id, reportLevel) AS
  (
    SELECT employee_id, last_name, manager_id, 0 reportLevel
    FROM employees
    WHERE employee_id = 101
  UNION ALL
    SELECT e.employee_id, e.last_name, e.manager_id, reportLevel+1
    FROM reports_to_101 r, employees e
    WHERE r.eid = e.manager_id
  )
SELECT eid, emp_last, mgr_id, reportLevel
FROM reports_to_101
WHERE reportLevel <= 1
ORDER BY reportLevel, eid;

       EID EMP_LAST                          MGR_ID REPORTLEVEL
---------- ------------------------- ---------- -----------
       101 Kochhar                          100           0
       108 Greenberg                        101           1
       200 Whalen                           101           1
       203 Mavris                           101           1
       204 Baer                             101           1
       205 Higgins                          101           1
```

The following statement shows the entire organization, indenting for each level of management.

```
WITH
  org_chart (eid, emp_last, mgr_id, reportLevel, salary, job_id) AS
  (
    SELECT employee_id, last_name, manager_id, 0 reportLevel, salary, job_id
    FROM employees
    WHERE manager_id is null
  UNION ALL
    SELECT e.employee_id, e.last_name, e.manager_id,
           r.reportLevel+1 reportLevel, e.salary, e.job_id
    FROM org_chart r, employees e
    WHERE r.eid = e.manager_id
  )
  SEARCH DEPTH FIRST BY emp_last SET order1
SELECT lpad(' ',2*reportLevel)||emp_last emp_name, eid, mgr_id, salary, job_id
```

```
FROM org_chart
ORDER BY order1;

EMP_NAME                    EID     MGR_ID     SALARY JOB_ID
-------------------- ---------- ---------- ---------- ----------
King                        100                 24000 AD_PRES
  Cambrault                 148        100      11000 SA_MAN
    Bates                   172        148       7300 SA_REP
    Bloom                   169        148      10000 SA_REP
    Fox                     170        148       9600 SA_REP
    Kumar                   173        148       6100 SA_REP
    Ozer                    168        148      11500 SA_REP
    Smith                   171        148       7400 SA_REP
  De Haan                   102        100      17000 AD_VP
    Hunold                  103        102       9000 IT_PROG
      Austin                105        103       4800 IT_PROG
      Ernst                 104        103       6000 IT_PROG
      Lorentz               107        103       4200 IT_PROG
      Pataballa             106        103       4800 IT_PROG
  Errazuriz                 147        100      12000 SA_MAN
    Ande                    166        147       6400 SA_REP
. . .
```

The following statement shows the entire organization, indenting for each level of management, with each level ordered by `hire_date`. The value of `is_cycle` is set to `Y` for any employee who has the same `hire_date` as any manager above him in the management chain.

```
WITH
  dup_hiredate (eid, emp_last, mgr_id, reportLevel, hire_date, job_id) AS
  (
    SELECT employee_id, last_name, manager_id, 0 reportLevel, hire_date, job_id
    FROM employees
    WHERE manager_id is null
  UNION ALL
    SELECT e.employee_id, e.last_name, e.manager_id,
           r.reportLevel+1 reportLevel, e.hire_date, e.job_id
    FROM dup_hiredate r, employees e
    WHERE r.eid = e.manager_id
  )
  SEARCH DEPTH FIRST BY hire_date SET order1
  CYCLE hire_date SET is_cycle TO 'Y' DEFAULT 'N'
SELECT lpad(' ',2*reportLevel)||emp_last emp_name, eid, mgr_id,
       hire_date, job_id, is_cycle
FROM dup_hiredate
ORDER BY order1;

EMP_NAME                    EID     MGR_ID HIRE_DATE JOB_ID     IS_CYCLE
-------------------- ---------- ---------- --------- ---------- --------
King                        100            17-JUN-03 AD_PRES           N
  De Haan                   102        100 13-JAN-01 AD_VP             N
    Hunold                  103        102 03-JAN-06 IT_PROG           N
      Austin                105        103 25-JUN-05 IT_PROG           N
. . .
  Kochhar                   101        100 21-SEP-05 AD_VP             N
    Mavris                  203        101 07-JUN-02 HR_REP            N
    Baer                    204        101 07-JUN-02 PR_REP            N
    Higgins                 205        101 07-JUN-02 AC_MGR            N
      Gietz                 206        205 07-JUN-02 AC_ACCOUNT        Y
    Greenberg               108        101 17-AUG-02 FI_MGR            N
      Faviet                109        108 16-AUG-02 FI_ACCOUNT        N
      Chen                  110        108 28-SEP-05 FI_ACCOUNT        N
. . .
```

The following statement counts the number of employees under each manager.

```
WITH
  emp_count (eid, emp_last, mgr_id, mgrLevel, salary, cnt_employees) AS
  (
    SELECT employee_id, last_name, manager_id, 0 mgrLevel, salary, 0 cnt_employees
    FROM employees
  UNION ALL
    SELECT e.employee_id, e.last_name, e.manager_id,
           r.mgrLevel+1 mgrLevel, e.salary, 1 cnt_employees
    FROM emp_count r, employees e
    WHERE e.employee_id = r.mgr_id
  )
  SEARCH DEPTH FIRST BY emp_last SET order1
SELECT emp_last, eid, mgr_id, salary, sum(cnt_employees), max(mgrLevel) mgrLevel
FROM emp_count
GROUP BY emp_last, eid, mgr_id, salary
HAVING max(mgrLevel) > 0
ORDER BY mgr_id NULLS FIRST, emp_last;

EMP_LAST                         EID     MGR_ID     SALARY SUM(CNT_EMPLOYEES)   MGRLEVEL
------------------ ---------- ---------- ---------- ------------------ ----------
King                          100                 24000                106          3
Cambrault                     148        100      11000                  7          2
De Haan                       102        100      17000                  5          2
Errazuriz                     147        100      12000                  6          1
Fripp                         121        100       8200                  8          1
Hartstein                     201        100      13000                  1          1
Kaufling                      122        100       7900                  8          1
. . .
```

Analytic Views: Examples

The following statement uses the persistent analytic view sales_av. The query selects the `member_name` hierarchical attribute of time_hier, which is the alias of a hierarchy of the same name, and values from the sales and units measures of the analytic view that are dimensioned by the time attribute dimension used by the time_hier hierarchy.. The results of the selection are filtered to those for the YEAR level of the hierarchy. The results are returned in hierarchical order.

```
SELECT time_hier.member_name as TIME,
 sales,
 units
FROM
 sales_av HIERARCHIES(time_hier)
WHERE time_hier.level_name = 'YEAR'
ORDER BY time_hier.hier_order;
```

The results of the query are the following:

```
TIME    SALES            UNITS
------  -------------  ---------
CY2011  6755115980.73  24462444
CY2012  6901682398.95  24400619
CY2013  7240938717.57  24407259
CY2014  7579746352.89  24402666
CY2015  7941102885.15  24475206
```

Transitory Analytic View Examples

The following statement defines the transitory analytic view my_av in the `WITH` clause. The transitory analytic view is based on the persistent analytic view sales_av. The lag_sales calculated measure is a `LAG` calculation that is used at query time.

```
WITH
  my_av ANALYTIC VIEW AS (
    USING sales_av HIERARCHIES (time_hier)
    ADD MEASURES (
      lag_sales AS (LAG(sales) OVER (HIERARCHY time_hier OFFSET 1))
    )
  )
SELECT time_hier.member_name time, sales, lag_sales
FROM my_av HIERARCHIES (time_hier)
WHERE time_hier.level_name = 'YEAR'
ORDER BY time_hier.hier_order;
```

The results of the query are the following:

```
TIME          SALES   LAG_SALES
------  ----------  ----------
CY2011  6755115981      (null)
CY2012  6901682399  6755115981
CY2013  7240938718  6901682399
CY2014  7579746353  7240938718
CY2015  7941102885  7579746353
```

The following statement defines a transitory analytic view that uses a filter clause.

```
WITH
  my_av ANALYTIC VIEW AS (
    USING sales_av HIERARCHIES (time_hier)
    FILTER FACT (
      time_hier TO quarter_of_year IN (1, 2)
        AND year_name IN ('CY2011', 'CY2012')
    )
  )
SELECT time_hier.member_name time, sales
  FROM my_av HIERARCHIES (time_hier)
  WHERE time_hier.level_name IN ('YEAR', 'QUARTER')
  ORDER BY time_hier.hier_order;
```

The results of the query are the following:

```
TIME            SALES
--------  ----------
CY2011    3340459835
Q1CY2011  1625299627
Q2CY2011  1715160208
CY2012    3397271965
Q1CY2012  1644857783
Q2CY2012  1752414182
```

Inline Analytic View Example

The following statement defines an inline analytic view in the `FROM` clause. The transitory analytic view is based on the persistent analytic view sales_av. The lag_sales calculated measure is a `LAG` calculation that is used at query time.

```
SELECT time_hier.member_name time, sales, lag_sales
FROM
  ANALYTIC VIEW (
    USING sales_av HIERARCHIES (time_hier)
    ADD MEASURES (
      lag_sales AS (LAG(sales) OVER (HIERARCHY time_hier OFFSET 1))
    )
  )
WHERE time_hier.level_name = 'YEAR'
ORDER BY time_hier.hier_order;
```

The results of the query are the following:

```
TIME         SALES   LAG_SALES
------  ----------  ----------
CY2011  6755115981      (null)
CY2012  6901682399  6755115981
CY2013  7240938718  6901682399
CY2014  7579746353  7240938718
CY2015  7941102885  7579746353
```

Simple Query Examples

The following statement selects rows from the `employees` table with the department number of 30:

```
SELECT *
   FROM employees
   WHERE department_id = 30
   ORDER BY last_name;
```

The following statement selects the name, job, salary and department number of all employees except purchasing clerks from department number 30:

```
SELECT last_name, job_id, salary, department_id
   FROM employees
   WHERE NOT (job_id = 'PU_CLERK' AND department_id = 30)
   ORDER BY last_name;
```

The following statement selects from subqueries in the `FROM` clause and for each department returns the total employees and salaries as a decimal value of all the departments:

```
SELECT a.department_id "Department",
   a.num_emp/b.total_count "%_Employees",
   a.sal_sum/b.total_sal "%_Salary"
FROM
(SELECT department_id, COUNT(*) num_emp, SUM(salary) sal_sum
   FROM employees
   GROUP BY department_id) a,
(SELECT COUNT(*) total_count, SUM(salary) total_sal
   FROM employees) b
ORDER BY a.department_id;
```

Selecting from a Partition: Example

You can select rows from a single partition of a partitioned table by specifying the keyword `PARTITION` in the `FROM` clause. This SQL statement assigns an alias for and retrieves rows from the `sales_q2_2000` partition of the sample table `sh.sales`:

```
SELECT * FROM sales PARTITION (sales_q2_2000) s
   WHERE s.amount_sold > 1500
   ORDER BY cust_id, time_id, channel_id;
```

The following example selects rows from the `oe.orders` table for orders earlier than a specified date:

```
SELECT * FROM orders
   WHERE order_date < TO_DATE('2006-06-15', 'YYYY-MM-DD');
```

Selecting a Sample: Examples

The following query estimates the number of orders in the `oe.orders` table:

```
SELECT COUNT(*) * 10 FROM orders SAMPLE (10);

COUNT(*)*10
-----------
         70
```

Because the query returns an estimate, the actual return value may differ from one query to the next.

```
SELECT COUNT(*) * 10 FROM orders SAMPLE (10);

COUNT(*)*10
-----------
         80
```

The following query adds a seed value to the preceding query. Oracle Database always returns the same estimate given the same seed value:

```
SELECT COUNT(*) * 10 FROM orders SAMPLE(10) SEED (1);

COUNT(*)*10
-----------
        130

SELECT COUNT(*) * 10 FROM orders SAMPLE(10) SEED(4);

COUNT(*)*10
-----------
        120

SELECT COUNT(*) * 10 FROM orders SAMPLE(10) SEED (1);

COUNT(*)*10
-----------
        130
```

Using Flashback Queries: Example

The following statements show a current value from the sample table `hr.employees` and then change the value. The intervals used in these examples are very short for demonstration purposes. Time intervals in your own environment are likely to be larger.

```
SELECT salary FROM employees
   WHERE last_name = 'Chung';

    SALARY
----------
      3800

UPDATE employees SET salary = 4000
   WHERE last_name = 'Chung';
1 row updated.

SELECT salary FROM employees
   WHERE last_name = 'Chung';

    SALARY
----------
      4000
```

To learn what the value was before the update, you can use the following Flashback Query:

```
SELECT salary FROM employees
   AS OF TIMESTAMP (SYSTIMESTAMP - INTERVAL '1' MINUTE)
   WHERE last_name = 'Chung';

    SALARY
----------
      3800
```

To learn what the values were during a particular time period, you can use a version Flashback Query:

```
SELECT salary FROM employees
  VERSIONS BETWEEN TIMESTAMP
    SYSTIMESTAMP - INTERVAL '10' MINUTE AND
    SYSTIMESTAMP - INTERVAL '1' MINUTE
  WHERE last_name = 'Chung';
```

To revert to the earlier value, use the Flashback Query as the subquery of another `UPDATE` statement:

```
UPDATE employees SET salary =
   (SELECT salary FROM employees
   AS OF TIMESTAMP (SYSTIMESTAMP - INTERVAL '2' MINUTE)
   WHERE last_name = 'Chung')
   WHERE last_name = 'Chung';
1 row updated.

SELECT salary FROM employees
   WHERE last_name = 'Chung';

    SALARY
----------
      3800
```

Using the GROUP BY Clause: Examples

To return the minimum and maximum salaries for each department in the `employees` table, issue the following statement:

```
SELECT department_id, MIN(salary), MAX (salary)
     FROM employees
     GROUP BY department_id
   ORDER BY department_id;
```

To return the minimum and maximum salaries for the clerks in each department, issue the following statement:

```
SELECT department_id, MIN(salary), MAX (salary)
     FROM employees
     WHERE job_id = 'PU_CLERK'
     GROUP BY department_id
   ORDER BY department_id;
```

The following example counts how many employees were hired each year. The `GROUP BY` clause uses the column alias `YEAR_HIRED`, so this groups using the expression `TRUNC(hire_date, 'YYYY')`

```
SELECT TRUNC(hire_date, 'YYYY') year_hired, COUNT(*)
FROM employees
GROUP BY year_hired
ORDER BY year_hired;

YEAR_HIRED COUNT(*)
----------- ----------
01-JAN-2011 1
01-JAN-2012 7
...
01-JAN-2017 19
01-JAN-2018 11
```

The following example counts how many employees were hired each day. The query groups by `HIRE_DATE`, which is the name of a column in `EMPLOYEES` and a `SELECT` list alias. The column name takes priority, so the query groups by the column, not the alias.

```
SELECT TRUNC(hire_date, 'YYYY') hire_date, COUNT(*)
FROM employees
GROUP BY hire_date
ORDER BY hire_date;

HIRE_DATE COUNT(*)
----------- ----------
01-JAN-2011 1
01-JAN-2012 4
01-JAN-2012 1
...
01-JAN-2018 1
01-JAN-2018 1
```

Using the GROUP BY CUBE Clause: Example

To return the number of employees and their average yearly salary across all possible combinations of department and job category, issue the following query on the sample tables `hr.employees` and `hr.departments`:

```
SELECT DECODE(GROUPING(department_name), 1, 'All Departments',
      department_name) AS department_name,
   DECODE(GROUPING(job_id), 1, 'All Jobs', job_id) AS job_id,
   COUNT(*) "Total Empl", AVG(salary) * 12 "Average Sal"
   FROM employees e, departments d
   WHERE d.department_id = e.department_id
   GROUP BY CUBE (department_name, job_id)
   ORDER BY department_name, job_id;

DEPARTMENT_NAME                JOB_ID     Total Empl Average Sal
```

```
------------------------------ ---------- ---------- -----------
Accounting                     AC_ACCOUNT          1       99600
Accounting                     AC_MGR              1      144000
Accounting                     All Jobs            2      121800
Administration                 AD_ASST             1       52800
. . .
Shipping                       ST_CLERK           20       33420
Shipping                       ST_MAN              5       87360
```

Using the GROUPING SETS Clause: Example

The following example finds the sum of sales aggregated for three precisely specified groups:

- `(channel_desc, calendar_month_desc, country_id)`
- `(channel_desc, country_id)`
- `(calendar_month_desc, country_id)`

Without the `GROUPING SETS` syntax, you would have to write less efficient queries with more complicated SQL. For example, you could run three separate queries and `UNION` them, or run a query with a `CUBE(channel_desc, calendar_month_desc, country_id)` operation and filter out five of the eight groups it would generate.

```
SELECT channel_desc, calendar_month_desc, co.country_id,
      TO_CHAR(sum(amount_sold) , '9,999,999,999') SALES$
   FROM sales, customers, times, channels, countries co
   WHERE sales.time_id=times.time_id
      AND sales.cust_id=customers.cust_id
      AND sales.channel_id= channels.channel_id
      AND customers.country_id = co.country_id
      AND channels.channel_desc IN ('Direct Sales', 'Internet')
      AND times.calendar_month_desc IN ('2000-09', '2000-10')
      AND co.country_iso_code IN ('UK', 'US')
  GROUP BY GROUPING SETS(
      (channel_desc, calendar_month_desc, co.country_id),
      (channel_desc, co.country_id),
      (calendar_month_desc, co.country_id) );

CHANNEL_DESC         CALENDAR COUNTRY_ID     SALES$
-------------------- -------- ----------     ----------
Internet             2000-09       52790        124,224
Direct Sales         2000-09       52790        638,201
Internet             2000-10       52790        137,054
Direct Sales         2000-10       52790        682,297
                     2000-09       52790        762,425
                     2000-10       52790        819,351
Internet                           52790        261,278
Direct Sales                       52790      1,320,497
```

See Also:

The functions GROUP_ID , GROUPING , and GROUPING_ID for more information on those functions

Hierarchical Query: Examples

The following query with a `CONNECT BY` clause defines a hierarchical relationship in which the `employee_id` value of the parent row is equal to the `manager_id` value of the child row:

```
SELECT last_name, employee_id, manager_id FROM employees
   CONNECT BY employee_id = manager_id
   ORDER BY last_name;
```

In the following `CONNECT BY` clause, the `PRIOR` operator applies only to the `employee_id` value. To evaluate this condition, the database evaluates `employee_id` values for the parent row and `manager_id`, `salary`, and `commission_pct` values for the child row:

```
SELECT last_name, employee_id, manager_id FROM employees
   CONNECT BY PRIOR employee_id = manager_id
   AND salary > commission_pct
   ORDER BY last_name;
```

To qualify as a child row, a row must have a `manager_id` value equal to the `employee_id` value of the parent row and it must have a `salary` value greater than its `commission_pct` value.

Using the HAVING Condition: Example

To return the minimum and maximum salaries for the employees in each department whose lowest salary is less than $5,000, issue the next statement:

```
SELECT department_id, MIN(salary), MAX (salary)
   FROM employees
   GROUP BY department_id
   HAVING MIN(salary) < 5000
   ORDER BY department_id;

DEPARTMENT_ID MIN(SALARY) MAX(SALARY)
------------- ----------- -----------
           10        4400        4400
           30        2500       11000
           50        2100        8200
           60        4200        9000
```

The following example uses a correlated subquery in a `HAVING` clause that eliminates from the result set any departments without managers and managers without departments:

```
SELECT department_id, manager_id
   FROM employees
   GROUP BY department_id, manager_id HAVING (department_id, manager_id) IN
   (SELECT department_id, manager_id FROM employees x
      WHERE x.department_id = employees.department_id)
   ORDER BY department_id;
```

Using the ORDER BY Clause: Examples

To select all purchasing clerk records from `employees` and order the results by salary in descending order, issue the following statement:

```
SELECT *
   FROM employees
   WHERE job_id = 'PU_CLERK'
   ORDER BY salary DESC;
```

To select information from `employees` ordered first by ascending department number and then by descending salary, issue the following statement:

```
SELECT last_name, department_id, salary
   FROM employees
   ORDER BY department_id ASC, salary DESC, last_name;
```

To select the same information as the previous `SELECT` and use the positional `ORDER BY` notation, issue the following statement, which orders by ascending `department_id`, then descending `salary`, and finally alphabetically by `last_name`:

```
SELECT last_name, department_id, salary
   FROM employees
   ORDER BY 2 ASC, 3 DESC, 1;
```

The MODEL clause: Examples

The view created below is based on the sample `sh` schema and is used by the example that follows.

```
CREATE OR REPLACE VIEW sales_view_ref AS
  SELECT country_name country,
         prod_name prod,
         calendar_year year,
         SUM(amount_sold) sale,
         COUNT(amount_sold) cnt
    FROM sales,times,customers,countries,products
    WHERE sales.time_id = times.time_id
      AND sales.prod_id = products.prod_id
      AND sales.cust_id = customers.cust_id
      AND customers.country_id = countries.country_id
      AND ( customers.country_id = 52779
            OR customers.country_id = 52776 )
      AND ( prod_name = 'Standard Mouse'
            OR prod_name = 'Mouse Pad' )
    GROUP BY country_name,prod_name,calendar_year;

SELECT country, prod, year, sale
  FROM sales_view_ref
  ORDER BY country, prod, year;

COUNTRY       PROD                                    YEAR       SALE
----------    -----------------------------------  --------  ---------
France        Mouse Pad                               1998    2509.42
France        Mouse Pad                               1999    3678.69
France        Mouse Pad                               2000    3000.72
France        Mouse Pad                               2001    3269.09
France        Standard Mouse                          1998    2390.83
France        Standard Mouse                          1999    2280.45
France        Standard Mouse                          2000    1274.31
France        Standard Mouse                          2001    2164.54
Germany       Mouse Pad                               1998    5827.87
Germany       Mouse Pad                               1999    8346.44
Germany       Mouse Pad                               2000    7375.46
Germany       Mouse Pad                               2001    9535.08
Germany       Standard Mouse                          1998    7116.11
Germany       Standard Mouse                          1999    6263.14
Germany       Standard Mouse                          2000    2637.31
Germany       Standard Mouse                          2001    6456.13

16 rows selected.
```

The next example creates a multidimensional array from `sales_view_ref` with columns containing country, product, year, and sales. It also:

- Assigns the sum of the sales of the Mouse Pad for years 1999 and 2000 to the sales of the Mouse Pad for year 2001, if a row containing sales of the Mouse Pad for year 2001 exists.

- Assigns the value of sales of the Standard Mouse for year 2001 to sales of the Standard Mouse for year 2002, creating a new row if a row containing sales of the Standard Mouse for year 2002 does not exist.

```
SELECT country,prod,year,s
  FROM sales_view_ref
  MODEL
    PARTITION BY (country)
    DIMENSION BY (prod, year)
    MEASURES (sale s)
    IGNORE NAV
    UNIQUE DIMENSION
    RULES UPSERT SEQUENTIAL ORDER
    (
      s[prod='Mouse Pad', year=2001] =
        s['Mouse Pad', 1999] + s['Mouse Pad', 2000],
      s['Standard Mouse', 2002] = s['Standard Mouse', 2001]
    )
  ORDER BY country, prod, year;

COUNTRY        PROD                                      YEAR       SALE
----------     -----------------------------------   --------  ---------
France         Mouse Pad                                 1998    2509.42
France         Mouse Pad                                 1999    3678.69
France         Mouse Pad                                 2000    3000.72
France         Mouse Pad                                 2001    6679.41
France         Standard Mouse                            1998    2390.83
France         Standard Mouse                            1999    2280.45
France         Standard Mouse                            2000    1274.31
France         Standard Mouse                            2001    2164.54
France         Standard Mouse                            2002    2164.54
Germany        Mouse Pad                                 1998    5827.87
Germany        Mouse Pad                                 1999    8346.44
Germany        Mouse Pad                                 2000    7375.46
Germany        Mouse Pad                                 2001    15721.9
Germany        Standard Mouse                            1998    7116.11
Germany        Standard Mouse                            1999    6263.14
Germany        Standard Mouse                            2000    2637.31
Germany        Standard Mouse                            2001    6456.13
Germany        Standard Mouse                            2002    6456.13

18 rows selected.
```

The first rule uses `UPDATE` behavior because symbolic referencing is used on the left-hand side of the rule. The rows represented by the left-hand side of the rule exist, so the measure columns are updated. If the rows did not exist, then no action would have been taken.

The second rule uses `UPSERT` behavior because positional referencing is used on the left-hand side and a single cell is referenced. The rows do not exist, so new rows are inserted and the related measure columns are updated. If the rows did exist, then the measure columns would have been updated.

See Also:

Oracle Database Data Warehousing Guide for an expanded discussion and examples

The next example uses the same `sales_view_ref` view and the analytic function `SUM` to calculate a cumulative sum (`csum`) of sales per country and per year.

```
SELECT country, year, sale, csum
   FROM
   (SELECT country, year, SUM(sale) sale
    FROM sales_view_ref
    GROUP BY country, year
   )
   MODEL DIMENSION BY (country, year)
         MEASURES (sale, 0 csum)
         RULES (csum[any, any]=
                  SUM(sale) OVER (PARTITION BY country
                                  ORDER BY year
                                  ROWS UNBOUNDED PRECEDING)
                )
   ORDER BY country, year;

COUNTRY               YEAR       SALE       CSUM
--------------- ---------- ---------- ----------
France                1998    4900.25    4900.25
France                1999    5959.14   10859.39
France                2000    4275.03   15134.42
France                2001    5433.63   20568.05
Germany               1998   12943.98   12943.98
Germany               1999   14609.58   27553.56
Germany               2000   10012.77   37566.33
Germany               2001   15991.21   53557.54

8 rows selected.
```

Row Limiting: Examples

The following statement returns the 5 employees with the lowest `employee_id` values:

```
SELECT employee_id, last_name
  FROM employees
  ORDER BY employee_id
  FETCH FIRST 5 ROWS ONLY;

EMPLOYEE_ID LAST_NAME
----------- -------------------------
        100 King
        101 Kochhar
        102 De Haan
        103 Hunold
        104 Ernst
```

The following statement returns the next 5 employees with the lowest `employee_id` values:

```
SELECT employee_id, last_name
  FROM employees
  ORDER BY employee_id
  OFFSET 5 ROWS FETCH NEXT 5 ROWS ONLY;

EMPLOYEE_ID LAST_NAME
----------- -------------------------
        105 Austin
        106 Pataballa
        107 Lorentz
        108 Greenberg
        109 Faviet
```

The following statement returns the 5 percent of employees with the lowest salaries:

```
SELECT employee_id, last_name, salary
  FROM employees
  ORDER BY salary
  FETCH FIRST 5 PERCENT ROWS ONLY;

EMPLOYEE_ID LAST_NAME                     SALARY
----------- ------------------------- ----------
        132 Olson                           2100
        128 Markle                          2200
        136 Philtanker                      2200
        127 Landry                          2400
        135 Gee                             2400
        119 Colmenares                      2500
```

Because `WITH TIES` is specified, the following statement returns the 5 percent of employees with the lowest salaries, plus all additional employees with the same salary as the last row fetched in the previous example:

```
SELECT employee_id, last_name, salary
  FROM employees
  ORDER BY salary
  FETCH FIRST 5 PERCENT ROWS WITH TIES;

EMPLOYEE_ID LAST_NAME                     SALARY
----------- ------------------------- ----------
        132 Olson                           2100
        128 Markle                          2200
        136 Philtanker                      2200
        127 Landry                          2400
        135 Gee                             2400
        119 Colmenares                      2500
        131 Marlow                          2500
        140 Patel                           2500
        144 Vargas                          2500
        182 Sullivan                        2500
        191 Perkins                         2500
```

Using the FOR UPDATE Clause: Examples

The following statement locks rows in the `employees` table with purchasing clerks located in Oxford, which has `location_id` 2500, and locks rows in the `departments` table with departments in Oxford that have purchasing clerks:

```
SELECT e.employee_id, e.salary, e.commission_pct
   FROM employees e, departments d
   WHERE job_id = 'SA_REP'
   AND e.department_id = d.department_id
   AND location_id = 2500
   ORDER BY e.employee_id
   FOR UPDATE;
```

The following statement locks only those rows in the `employees` table with purchasing clerks located in Oxford. No rows are locked in the `departments` table:

```
SELECT e.employee_id, e.salary, e.commission_pct
   FROM employees e JOIN departments d
   USING (department_id)
   WHERE job_id = 'SA_REP'
   AND location_id = 2500
   ORDER BY e.employee_id
   FOR UPDATE OF e.salary;
```

Using the WITH CHECK OPTION Clause: Example

The following statement is legal even though the third value inserted violates the condition of the subquery *`where_clause`*:

```
INSERT INTO (SELECT department_id, department_name, location_id
   FROM departments WHERE location_id < 2000)
   VALUES (9999, 'Entertainment', 2500);
```

However, the following statement is illegal because it contains the `WITH CHECK OPTION` clause:

```
INSERT INTO (SELECT department_id, department_name, location_id
   FROM departments WHERE location_id < 2000 WITH CHECK OPTION)
   VALUES (9999, 'Entertainment', 2500);
     *
ERROR at line 2:
ORA-01402: view WITH CHECK OPTION where-clause violation
```

Using PIVOT and UNPIVOT: Examples

The `oe.orders` table contains information about when an order was placed (`order_date`), how it was place (`order_mode`), and the total amount of the order (`order_total`), as well as other information. The following example shows how to use the `PIVOT` clause to pivot `order_mode` values into columns, aggregating `order_total` data in the process, to get yearly totals by order mode:

```
CREATE TABLE pivot_table AS
SELECT * FROM
(SELECT EXTRACT(YEAR FROM order_date) year, order_mode, order_total FROM orders)
PIVOT
(SUM(order_total) FOR order_mode IN ('direct' AS Store, 'online' AS Internet));

SELECT * FROM pivot_table ORDER BY year;

      YEAR      STORE   INTERNET
---------- ---------- ----------
      2004     5546.6
      2006   371895.5   100056.6
      2007  1274078.8  1271019.5
      2008   252108.3   393349.4
```

The `UNPIVOT` clause lets you rotate specified columns so that the input column headings are output as values of one or more descriptor columns, and the input column values are output as values of one or more measures columns. The first query that follows shows that nulls are excluded by default. The second query shows that you can include nulls using the `INCLUDE NULLS` clause.

```
SELECT * FROM pivot_table
  UNPIVOT (yearly_total FOR order_mode IN (store AS 'direct',
           internet AS 'online'))
  ORDER BY year, order_mode;

      YEAR ORDER_ YEARLY_TOTAL
---------- ------ ------------
      2004 direct       5546.6
      2006 direct     371895.5
      2006 online     100056.6
      2007 direct    1274078.8
      2007 online    1271019.5
      2008 direct     252108.3
      2008 online     393349.4

7 rows selected.

SELECT * FROM pivot_table
  UNPIVOT INCLUDE NULLS
    (yearly_total FOR order_mode IN (store AS 'direct', internet AS 'online'))
  ORDER BY year, order_mode;

      YEAR ORDER_ YEARLY_TOTAL
---------- ------ ------------
      2004 direct       5546.6
      2004 online
      2006 direct     371895.5
      2006 online     100056.6
      2007 direct    1274078.8
      2007 online    1271019.5
      2008 direct     252108.3
      2008 online     393349.4

8 rows selected.
```

Using Join Queries: Examples

The following examples show various ways of joining tables in a query. In the first example, an equijoin returns the name and job of each employee and the number and name of the department in which the employee works:

```
SELECT last_name, job_id, departments.department_id, department_name
   FROM employees, departments
   WHERE employees.department_id = departments.department_id
   ORDER BY last_name, job_id;

LAST_NAME           JOB_ID     DEPARTMENT_ID DEPARTMENT_NAME
------------------- ---------- ------------- ----------------------
Abel                 SA_REP               80 Sales
Ande                 SA_REP               80 Sales
Atkinson             ST_CLERK             50 Shipping
Austin               IT_PROG              60 IT
. . .
```

You must use a join to return this data because employee names and jobs are stored in a different table than department names. Oracle Database combines rows of the two tables according to this join condition:

```
employees.department_id = departments.department_id
```

The following equijoin returns the name, job, department number, and department name of all sales managers:

```
SELECT last_name, job_id, departments.department_id, department_name
   FROM employees, departments
   WHERE employees.department_id = departments.department_id
   AND job_id = 'SA_MAN'
   ORDER BY last_name;

LAST_NAME           JOB_ID     DEPARTMENT_ID DEPARTMENT_NAME
------------------- ---------- ------------- -----------------------
Cambrault           SA_MAN                80 Sales
Errazuriz           SA_MAN                80 Sales
Partners            SA_MAN                80 Sales
```

```
Russell              SA_MAN                80 Sales
Zlotkey              SA_MAN                80 Sales
```

This query is identical to the preceding example, except that it uses an additional *where_clause* condition to return only rows with a `job` value of '`SA_MAN`'.

Using Subqueries: Examples

To determine who works in the same department as employee '`Lorentz`', issue the following statement:

```
SELECT last_name, department_id FROM employees
   WHERE department_id =
     (SELECT department_id FROM employees
      WHERE last_name = 'Lorentz')
   ORDER BY last_name, department_id;
```

To give all employees in the `employees` table a 10% raise if they have changed jobs—if they appear in the `job_history` table—issue the following statement:

```
UPDATE employees
    SET salary = salary * 1.1
    WHERE employee_id IN (SELECT employee_id FROM job_history);
```

To create a second version of the `departments` table `new_departments`, with only three of the columns of the original table, issue the following statement:

```
CREATE TABLE new_departments
   (department_id, department_name, location_id)
   AS SELECT department_id, department_name, location_id
   FROM departments;
```

Using Self Joins: Example

The following query uses a self join to return the name of each employee along with the name of the employee's manager. A `WHERE` clause is added to shorten the output.

```
SELECT e1.last_name||' works for '||e2.last_name
   "Employees and Their Managers"
   FROM employees e1, employees e2
   WHERE e1.manager_id = e2.employee_id
      AND e1.last_name LIKE 'R%'
   ORDER BY e1.last_name;

Employees and Their Managers
-------------------------------
Rajs works for Mourgos
Raphaely works for King
Rogers works for Kaufling
Russell works for King
```

The join condition for this query uses the aliases `e1` and `e2` for the sample table `employees`:

```
e1.manager_id = e2.employee_id
```

Using Outer Joins: Examples

The following example shows how a partitioned outer join fills data gaps in rows to facilitate analytic function specification and reliable report formatting. The example first creates a small data table to be used in the join:

```
SELECT d.department_id, e.last_name
   FROM departments d LEFT OUTER JOIN employees e
   ON d.department_id = e.department_id
   ORDER BY d.department_id, e.last_name;
```

Users familiar with the traditional Oracle Database outer joins syntax will recognize the same query in this form:

```
SELECT d.department_id, e.last_name
   FROM departments d, employees e
   WHERE d.department_id = e.department_id(+)
   ORDER BY d.department_id, e.last_name;
```

Oracle strongly recommends that you use the more flexible `FROM` clause join syntax shown in the former example.

The left outer join returns all departments, including those without any employees. The same statement with a right outer join returns all employees, including those not yet assigned to a department:

Note:

The employee Zeuss was added to the employees table for these examples, and is not part of the sample data.

```
SELECT d.department_id, e.last_name
   FROM departments d RIGHT OUTER JOIN employees e
   ON d.department_id = e.department_id
   ORDER BY d.department_id, e.last_name;

DEPARTMENT_ID LAST_NAME
------------- -------------------------
. . .
          110 Gietz
          110 Higgins
              Grant
              Zeuss
```

It is not clear from this result whether employees Grant and Zeuss have `department_id` `NULL`, or whether their `department_id` is not in the `departments` table. To determine this requires a full outer join:

```
SELECT d.department_id as d_dept_id, e.department_id as e_dept_id,
      e.last_name
   FROM departments d FULL OUTER JOIN employees e
   ON d.department_id = e.department_id
   ORDER BY d.department_id, e.last_name;

 D_DEPT_ID  E_DEPT_ID LAST_NAME
---------- ---------- -------------------------
  . . .
       110        110 Gietz
       110        110 Higgins
  . . .
       260
       270
                  999 Zeuss
                      Grant
```

Because the column names in this example are the same in both tables in the join, you can also use the common column feature by specifying the `USING` clause of the join syntax. The output is the same as for the preceding example except that the `USING` clause coalesces the two matching columns `department_id` into a single column output:

```
SELECT department_id AS d_e_dept_id, e.last_name
   FROM departments d FULL OUTER JOIN employees e
   USING (department_id)
   ORDER BY department_id, e.last_name;

D_E_DEPT_ID LAST_NAME
----------- -------------------------
  . . .
        110 Higgins
        110 Gietz
  . . .
        260
        270
        999 Zeuss
            Grant
```

Using Partitioned Outer Joins: Examples

The following example shows how a partitioned outer join fills in gaps in rows to facilitate analytic calculation specification and reliable report formatting. The example first creates and populates a simple table to be used in the join:

```
CREATE TABLE inventory (time_id    DATE,
                        product    VARCHAR2(10),
                        quantity   NUMBER);

INSERT INTO inventory VALUES (TO_DATE('01/04/01', 'DD/MM/YY'), 'bottle', 10);
INSERT INTO inventory VALUES (TO_DATE('06/04/01', 'DD/MM/YY'), 'bottle', 10);
INSERT INTO inventory VALUES (TO_DATE('01/04/01', 'DD/MM/YY'), 'can', 10);
INSERT INTO inventory VALUES (TO_DATE('04/04/01', 'DD/MM/YY'), 'can', 10);

SELECT times.time_id, product, quantity FROM inventory
   PARTITION BY  (product)
   RIGHT OUTER JOIN times ON (times.time_id = inventory.time_id)
   WHERE times.time_id BETWEEN TO_DATE('01/04/01', 'DD/MM/YY')
      AND TO_DATE('06/04/01', 'DD/MM/YY')
   ORDER BY  2,1;

TIME_ID   PRODUCT      QUANTITY
--------- ---------- ----------
01-APR-01 bottle             10
02-APR-01 bottle
03-APR-01 bottle
04-APR-01 bottle
05-APR-01 bottle
06-APR-01 bottle             10
01-APR-01 can                10
02-APR-01 can
03-APR-01 can
04-APR-01 can                10
05-APR-01 can
06-APR-01 can

12 rows selected.
```

The data is now more dense along the time dimension for each partition of the product dimension. However, each of the newly added rows within each partition is null in the quantity column. It is more useful to see the nulls replaced by the preceding non-`NULL` value in time order. You can achieve this by applying the analytic function `LAST_VALUE` on top of the query result:

```
SELECT time_id, product, LAST_VALUE(quantity IGNORE NULLS)
   OVER (PARTITION BY product ORDER BY time_id) quantity
   FROM ( SELECT times.time_id, product, quantity
             FROM inventory PARTITION BY  (product)
                RIGHT OUTER JOIN times ON (times.time_id = inventory.time_id)
   WHERE times.time_id BETWEEN TO_DATE('01/04/01', 'DD/MM/YY')
      AND TO_DATE('06/04/01', 'DD/MM/YY'))
   ORDER BY  2,1;

TIME_ID   PRODUCT      QUANTITY
--------- ---------- ----------
01-APR-01 bottle             10
02-APR-01 bottle             10
03-APR-01 bottle             10
04-APR-01 bottle             10
05-APR-01 bottle             10
06-APR-01 bottle             10
01-APR-01 can                10
02-APR-01 can                10
03-APR-01 can                10
04-APR-01 can                10
05-APR-01 can                10
06-APR-01 can                10

12 rows selected.
```

See Also:

Oracle Database Data Warehousing Guide for an expanded discussion on filling gaps in time series calculations and examples of usage

Using Antijoins: Example

The following example selects a list of departments having no employee making 10000 or more as salary:

```
  SELECT department_name FROM hr.departments d
  WHERE NOT EXISTS (SELECT asdf FROM hr.employees e
                WHERE e.department_id = d.department_id
                AND e.salary >= 10000)
  ORDER BY department_name;
```

Using Semijoins: Example

In the following example, only one row needs to be returned from the `departments` table, even though many rows in the `employees` table might match the subquery. If no index has been defined on the `salary` column in `employees`, then a semijoin can be used to improve query performance.

```
SELECT * FROM departments
   WHERE EXISTS
   (SELECT * FROM employees
       WHERE departments.department_id = employees.department_id
```

```
      AND employees.salary > 2500)
  ORDER BY department_name;
```

Using CROSS APPLY and OUTER APPLY Joins: Examples

The following statement uses the `CROSS APPLY` clause of the *`cross_outer_apply_clause`*. The join returns only rows from the table on the left side of the join (`departments`) that produce a result from the inline view on the right side of the join. That is, the join returns only the departments that have at least one employee. The `WHERE` clause restricts the result set to include only the Marketing, Operations, and Public Relations departments. However, the Operations department is not included in the result set because it has no employees.

```
SELECT d.department_name, v.employee_id, v.last_name
  FROM departments d CROSS APPLY (SELECT * FROM employees e
                                  WHERE e.department_id = d.department_id) v
  WHERE d.department_name IN ('Marketing', 'Operations', 'Public Relations')
  ORDER BY d.department_name, v.employee_id;

DEPARTMENT_NAME                EMPLOYEE_ID LAST_NAME
------------------------------ ----------- -------------------------
Marketing                      201         Hartstein
Marketing                      202         Fay
Public Relations               204         Baer
```

The following statement uses the `OUTER APPLY` clause of the *`cross_outer_apply_clause`*. The join returns all rows from the table on the left side of the join (`departments`) regardless of whether they produce a result from the inline view on the right side of the join. That is, the join returns all departments regardless of whether the departments have any employees. The `WHERE` clause restricts the result set to include only the Marketing, Operations, and Public Relations departments. The Operations department is included in the result set even though it has no employees.

```
SELECT d.department_name, v.employee_id, v.last_name
  FROM departments d OUTER APPLY (SELECT * FROM employees e
                                  WHERE e.department_id = d.department_id) v
  WHERE d.department_name IN ('Marketing', 'Operations', 'Public Relations')
  ORDER by d.department_name, v.employee_id;

DEPARTMENT_NAME                EMPLOYEE_ID LAST_NAME
------------------------------ ----------- -------------------------
Marketing                      201         Hartstein
Marketing                      202         Fay
Operations
Public Relations               204         Baer
```

Using Lateral Inline Views: Example

The following example shows a scalar subquery that finds the highest-paid employee in each department, with `employee_id` as a tie-breaker:

```
  SELECT department_name,
   (SELECT last_name FROM
     (SELECT last_name FROM hr.employees e
       WHERE e.department_id = d.department_id
       ORDER BY e.salary DESC, e.employee_id ASC)
     WHERE ROWNUM = 1) highest_paid
  FROM hr.departments d;
```

If you would like not only to see the highest-paid employee's last name, but also their `first_name, salary`, and `email`, as separate columns, the above approach would require 4 separate scalar subqueries. A `LATERAL` join in this case offers a way to extend a scalar-like subqueries to return any number of columns and other expressions that can then be referenced any number of times anywhere these could be referenced from an ordinary join, the `SELECT` list, the `WHERE` clause, `ORDER BY`, `GROUP BY`, and others, for example:

```
  SELECT d.department_name, e2.last_name, e2.first_name, e2.salary, e2.email
  FROM hr.departments d,
     LATERAL (SELECT * FROM
                      (SELECT * FROM hr.employees e
                       WHERE e.department_id = d.department_id
                       ORDER BY e.salary DESC, e.employee_id ASC)
               WHERE ROWNUM = 1) e2;
```

Table Collections: Examples

You can perform DML operations on nested tables only if they are defined as columns of a table. Therefore, when the *`query_table_expr_clause`* of an `INSERT`, `DELETE`, or `UPDATE` statement is a *`table_collection_expression`*, the collection expression must be a subquery that uses the `TABLE` collection expression to select the nested table column of the table. The examples that follow are based on the following scenario:

Suppose the database contains a table `hr_info` with columns `department_id`, `location_id`, and `manager_id`, and a column of nested table type `people` which has `last_name`, `department_id`, and `salary` columns for all the employees of each respective manager:

```
CREATE TYPE people_typ AS OBJECT (
   last_name      VARCHAR2(25),
   department_id  NUMBER(4),
   salary         NUMBER(8,2));
/
CREATE TYPE people_tab_typ AS TABLE OF people_typ;
/
CREATE TABLE hr_info (
   department_id   NUMBER(4),
   location_id     NUMBER(4),
   manager_id      NUMBER(6),
   people          people_tab_typ)
   NESTED TABLE people STORE AS people_stor_tab;

INSERT INTO hr_info VALUES (280, 1800, 999, people_tab_typ());
```

The following example inserts into the `people` nested table column of the `hr_info` table for department 280:

```
INSERT INTO TABLE(SELECT h.people FROM hr_info h
   WHERE h.department_id = 280)
   VALUES ('Smith', 280, 1750);
```

The next example updates the department 280 `people` nested table:

```
UPDATE TABLE(SELECT h.people FROM hr_info h
   WHERE h.department_id = 280) p
   SET p.salary = p.salary + 100;
```

The next example deletes from the department 280 `people` nested table:

```
DELETE TABLE(SELECT h.people FROM hr_info h
   WHERE h.department_id = 280) p
   WHERE p.salary > 1700;
```

Collection Unnesting: Examples

To select data from a nested table column, use the `TABLE` collection expression to treat the nested table as columns of a table. This process is called **collection unnesting**.

You could get all the rows from `hr_info`, which was created in the preceding example, and all the rows from the `people` nested table column of `hr_info` using the following statement:

```
SELECT t1.department_id, t2.* FROM hr_info t1, TABLE(t1.people) t2
   WHERE t2.department_id = t1.department_id;
```

Now suppose that `people` is not a nested table column of `hr_info`, but is instead a separate table with columns `last_name`, `department_id`, `address`, `hiredate`, and `salary`. You can extract the same rows as in the preceding example with this statement:

```
SELECT t1.department_id, t2.*
   FROM hr_info t1, TABLE(CAST(MULTISET(
      SELECT t3.last_name, t3.department_id, t3.salary
         FROM people t3
      WHERE t3.department_id = t1.department_id)
      AS people_tab_typ)) t2;
```

Finally, suppose that `people` is neither a nested table column of table `hr_info` nor a table itself. Instead, you have created a function `people_func` that extracts from various sources the name, department, and salary of all employees. You can get the same information as in the preceding examples with the following query:

```
SELECT t1.department_id, t2.* FROM hr_info t1, TABLE(CAST
   (people_func( ... ) AS people_tab_typ)) t2;
```

See Also:

Oracle Database Object-Relational Developer's Guide for more examples of collection unnesting.

Using the LEVEL Pseudocolumn: Examples

The following statement returns all employees in hierarchical order. The root row is defined to be the employee whose job is `AD_VP`. The child rows of a parent row are defined to be those who have the employee number of the parent row as their manager number.

```
SELECT LPAD(' ',2*(LEVEL-1)) || last_name org_chart,
        employee_id, manager_id, job_id
    FROM employees
    START WITH job_id = 'AD_VP'
    CONNECT BY PRIOR employee_id = manager_id;

ORG_CHART          EMPLOYEE_ID MANAGER_ID JOB_ID
------------------ ----------- ---------- ----------
Kochhar                    101        100 AD_VP
  Greenberg                108        101 FI_MGR
    Faviet                 109        108 FI_ACCOUNT
    Chen                   110        108 FI_ACCOUNT
    Sciarra                111        108 FI_ACCOUNT
    Urman                  112        108 FI_ACCOUNT
    Popp                   113        108 FI_ACCOUNT
  Whalen                   200        101 AD_ASST
  Mavris                   203        101 HR_REP
  Baer                          204        101 PR_REP
  Higgins                       205        101 AC_MGR
    Gietz                       206        205 AC_ACCOUNT
De Haan                         102        100 AD_VP
  Hunold                        103        102 IT_PROG
    Ernst                       104        103 IT_PROG
    Austin                      105        103 IT_PROG
    Pataballa                   106        103 IT_PROG
    Lorentz                     107        103 IT_PROG
```

The following statement is similar to the previous one, except that it does not select employees with the job `FI_MGR`.

```
SELECT LPAD(' ',2*(LEVEL-1)) || last_name org_chart,
        employee_id, manager_id, job_id
    FROM employees
    WHERE job_id != 'FI_MGR'
    START WITH job_id = 'AD_VP'
    CONNECT BY PRIOR employee_id = manager_id;

ORG_CHART          EMPLOYEE_ID MANAGER_ID JOB_ID
------------------ ----------- ---------- ----------
Kochhar                         101        100 AD_VP
    Faviet                      109        108 FI_ACCOUNT
    Chen                        110        108 FI_ACCOUNT
    Sciarra                     111        108 FI_ACCOUNT
    Urman                       112        108 FI_ACCOUNT
    Popp                        113        108 FI_ACCOUNT
  Whalen                        200        101 AD_ASST
  Mavris                        203        101 HR_REP
  Baer                          204        101 PR_REP
  Higgins                       205        101 AC_MGR
    Gietz                       206        205 AC_ACCOUNT
De Haan                         102        100 AD_VP
  Hunold                        103        102 IT_PROG
    Ernst                       104        103 IT_PROG
    Austin                      105        103 IT_PROG
    Pataballa                   106        103 IT_PROG
    Lorentz                     107        103 IT_PROG
```

Oracle Database does not return the manager `Greenberg`, although it does return employees who are managed by `Greenberg`.

The following statement is similar to the first one, except that it uses the `LEVEL` pseudocolumn to select only the first two levels of the management hierarchy:

```
SELECT LPAD(' ',2*(LEVEL-1)) || last_name org_chart,
employee_id, manager_id, job_id
    FROM employees
    START WITH job_id = 'AD_PRES'
    CONNECT BY PRIOR employee_id = manager_id AND LEVEL <= 2;

ORG_CHART          EMPLOYEE_ID MANAGER_ID JOB_ID
------------------ ----------- ---------- ----------
King                            100            AD_PRES
  Kochhar                       101        100 AD_VP
  De Haan                       102        100 AD_VP
  Raphaely                      114        100 PU_MAN
  Weiss                         120        100 ST_MAN
  Fripp                         121        100 ST_MAN
  Kaufling                      122        100 ST_MAN
```

```
Vollman                    123        100 ST_MAN
Mourgos                    124        100 ST_MAN
Russell                    145        100 SA_MAN
Partners                   146        100 SA_MAN
Errazuriz                  147        100 SA_MAN
Cambrault                  148        100 SA_MAN
Zlotkey                    149        100 SA_MAN
Hartstein                  201        100 MK_MAN
```

Using Distributed Queries: Example

This example shows a query that joins the `departments` table on the local database with the `employees` table on the `remote` database:

```
SELECT last_name, department_name
   FROM employees@remote, departments
   WHERE employees.department_id = departments.department_id;
```

Using Correlated Subqueries: Examples

The following examples show the general syntax of a correlated subquery:

```
SELECT select_list
    FROM table1 t_alias1
    WHERE expr operator
        (SELECT column_list
            FROM table2 t_alias2
            WHERE t_alias1.column
               operator t_alias2.column);

UPDATE table1 t_alias1
    SET column =
        (SELECT expr
            FROM table2 t_alias2
            WHERE t_alias1.column = t_alias2.column);

DELETE FROM table1 t_alias1
    WHERE column operator
        (SELECT expr
            FROM table2 t_alias2
            WHERE t_alias1.column = t_alias2.column);
```

The following statement returns data about employees whose salaries exceed their department average. The following statement assigns an alias to `employees`, the table containing the salary information, and then uses the alias in a correlated subquery:

```
SELECT department_id, last_name, salary
   FROM employees x
   WHERE salary > (SELECT AVG(salary)
      FROM employees
      WHERE x.department_id = department_id)
   ORDER BY department_id;
```

For each row of the `employees` table, the parent query uses the correlated subquery to compute the average salary for members of the same department. The correlated subquery performs the following steps for each row of the `employees` table:

1. The `department_id` of the row is determined.
2. The `department_id` is then used to evaluate the parent query.
3. If the salary in that row is greater than the average salary of the departments of that row, then the row is returned.

The subquery is evaluated once for each row of the `employees` table.

Selecting from the DUAL Table: Example

The following statement returns the current date:

```
SELECT CURRENT_DATE FROM DUAL;
```

You could select `CURRENT_DATE` from the `employees` table, but the database would return 14 rows of the same `CURRENT_DATE`, one for every row of the `employees` table. Selecting from `DUAL` is more convenient.

From Release 23 you can omit the optional `FROM` clause as in the following example:

```
SELECT CURRENT_DATE;
```

Selecting Sequence Values: Examples

The following statement increments the `employees_seq` sequence and returns the new value:

```
SELECT employees_seq.nextval
    FROM DUAL;
```

The following statement selects the current value of `employees_seq`:

```
SELECT employees_seq.currval
    FROM DUAL;
```

Row Pattern Matching: Example

This example uses row pattern matching to query stock price data. The following statements create table `Ticker` and inserts stock price data into the table:

```
CREATE TABLE Ticker (SYMBOL VARCHAR2(10), tstamp DATE, price NUMBER);

INSERT INTO Ticker VALUES('ACME', '01-Apr-11', 12);
INSERT INTO Ticker VALUES('ACME', '02-Apr-11', 17);
INSERT INTO Ticker VALUES('ACME', '03-Apr-11', 19);
INSERT INTO Ticker VALUES('ACME', '04-Apr-11', 21);
INSERT INTO Ticker VALUES('ACME', '05-Apr-11', 25);
INSERT INTO Ticker VALUES('ACME', '06-Apr-11', 12);
INSERT INTO Ticker VALUES('ACME', '07-Apr-11', 15);
INSERT INTO Ticker VALUES('ACME', '08-Apr-11', 20);
INSERT INTO Ticker VALUES('ACME', '09-Apr-11', 24);
INSERT INTO Ticker VALUES('ACME', '10-Apr-11', 25);
INSERT INTO Ticker VALUES('ACME', '11-Apr-11', 19);
INSERT INTO Ticker VALUES('ACME', '12-Apr-11', 15);
INSERT INTO Ticker VALUES('ACME', '13-Apr-11', 25);
INSERT INTO Ticker VALUES('ACME', '14-Apr-11', 25);
INSERT INTO Ticker VALUES('ACME', '15-Apr-11', 14);
INSERT INTO Ticker VALUES('ACME', '16-Apr-11', 12);
INSERT INTO Ticker VALUES('ACME', '17-Apr-11', 14);
INSERT INTO Ticker VALUES('ACME', '18-Apr-11', 24);
INSERT INTO Ticker VALUES('ACME', '19-Apr-11', 23);
INSERT INTO Ticker VALUES('ACME', '20-Apr-11', 22);
```

The following query uses row pattern matching to find all cases where stock prices dipped to a bottom price and then rose. This is generally called a V-shape. The resulting output contains only three rows because the query specifies `ONE ROW PER MATCH`, and three matches were found.

```
SELECT *
FROM Ticker MATCH_RECOGNIZE (
     PARTITION BY symbol
     ORDER BY tstamp
     MEASURES STRT.tstamp AS start_tstamp,
              LAST(DOWN.tstamp) AS bottom_tstamp,
              LAST(UP.tstamp) AS end_tstamp
     ONE ROW PER MATCH
     AFTER MATCH SKIP TO LAST UP
     PATTERN (STRT DOWN+ UP+)
     DEFINE
        DOWN AS DOWN.price < PREV(DOWN.price),
        UP AS UP.price > PREV(UP.price)
     ) MR
ORDER BY MR.symbol, MR.start_tstamp;

SYMBOL     START_TST BOTTOM_TS END_TSTAM
---------- --------- --------- ---------
ACME       05-APR-11 06-APR-11 10-APR-11
ACME       10-APR-11 12-APR-11 13-APR-11
ACME       14-APR-11 16-APR-11 18-APR-11
```

Partitioned Row Limiting in Non-Vector Context: Example

The following example finds the top two departments that people with highest salary work in, and the top three people with the highest salary within each selected department:

```
SELECT deptno, ename FROM emp
ORDER BY sal DESC
FETCH FIRST 2 PARTITIONS BY deptno, 3 ROWS ONLY;
```

Partitioned Row Limiting in a Multi-Vector Search: Example

The following statement creates a table `chunk_table` with three columns: *`doc_id`* and *`chunk_id`* (of type `NUMBER`), and *`data_vec`* (of type `VECTOR`).

`doc_id` refers to the document id, *`chunk_id`* refers to the chunk id, and *`data_vec`* refers to the vector embedding.

```
CREATE TABLE chunk_table (
  doc_id NUMBER,
  chunk_id NUMBER,
  data_vec VECTOR
 );
```

The following query performs a multi-vector search :

```
SELECT doc_id,
  FROM chunk_table
  ORDER BY VECTOR_DISTANCE(data_vec, :query_vec)
  FETCH [APPROX] FIRST 10 PARTITIONS BY docId, 1 ROW ONLY;
```

SET CONSTRAINT[S]

Purpose

Use the `SET CONSTRAINTS` statement to specify, for a particular transaction, whether a deferrable constraint is checked following each DML statement (`IMMEDIATE`) or when the transaction is committed (`DEFERRED`). You can use this statement to set the mode for a list of constraint names or for `ALL` constraints.

The SET CONSTRAINTS mode lasts for the duration of the transaction or until another SET CONSTRAINTS statement resets the mode.

Note:

You can also use an ALTER SESSION statement with the SET CONSTRAINTS clause to set *all* deferrable constraints. This is equivalent to making issuing a SET CONSTRAINTS statement at the start of each transaction in the current session.

You cannot specify this statement inside of a trigger definition.

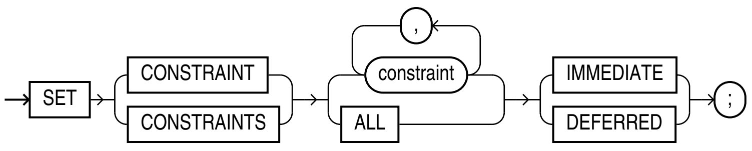
SET CONSTRAINTS can be a distributed statement. Existing database links that have transactions in process are notified when a SET CONSTRAINTS ALL statement is issued, and new links are notified that it was issued as soon as they start a transaction.

Prerequisites

To specify when a deferrable constraint is checked, you must have the READ or SELECT privilege on the table to which the constraint is applied unless the table is in your schema.

Syntax

***set_constraints*::=**



Semantics

constraint

Specify the name of one or more integrity constraints.

ALL

Specify ALL to set all deferrable constraints for this transaction.

IMMEDIATE

Specify IMMEDIATE to cause the specified constraints to be checked immediately on execution of each constrained DML statement. Oracle Database first checks any constraints that were deferred earlier in the transaction and then continues immediately checking constraints of any further statements in that transaction, as long as all the checked constraints are consistent and no other SET CONSTRAINTS statement is issued. If any constraint fails the check, then an error is signaled. At that point, a COMMIT statement causes the whole transaction to undo.

Making constraints immediate at the end of a transaction is a way of checking whether COMMIT can succeed. You can avoid unexpected rollbacks by setting constraints to IMMEDIATE as the last statement in a transaction. If any constraint fails the check, you can then correct the error before committing the transaction.

DEFERRED

Specify `DEFERRED` to indicate that the conditions specified by the deferrable constraint are checked when the transaction is committed.

Note:

You can verify the success of deferrable constraints prior to committing them by issuing a `SET CONSTRAINTS ALL IMMEDIATE` statement.

Examples

Setting Constraints: Examples

The following statement sets all deferrable constraints in this transaction to be checked immediately following each DML statement:

```
SET CONSTRAINTS ALL IMMEDIATE;
```

The following statement checks three deferred constraints when the transaction is committed. This example fails if the constraints were specified to be `NOT DEFERRABLE`.

```
SET CONSTRAINTS emp_job_nn, emp_salary_min,
   hr.jhist_dept_fk@remote DEFERRED;
```

SET ROLE

Purpose

When a user logs on to Oracle Database, the database enables all privileges granted explicitly to the user and all privileges in the user's default roles. During the session, the user or an application can use the `SET ROLE` statement any number of times to enable or disable the roles currently enabled for the session.

You cannot enable more than 148 user-defined roles at one time.

Note:

- For most roles, you cannot enable or disable a role unless it was granted to you either directly or through other roles. However, a secure application role can be granted and enabled by its associated PL/SQL package. See the `CREATE ROLE` semantics for USING *package* and *Oracle Database Security Guide* for information about secure application roles.
- `SET ROLE` succeeds only if there are no definer's rights units on the call stack. If at least one DR unit is on the call stack, then issuing the `SET ROLE` command causes `ORA-06565`. See *Oracle Database PL/SQL Language Reference* for more information about definer's rights units.
- To run the `SET ROLE` command from PL/SQL, you must use dynamic SQL, preferably the `EXECUTE IMMEDIATE` statement. See *Oracle Database PL/SQL Language Reference* for more information about this statement.

You can see which roles are currently enabled by examining the `SESSION_ROLES` data dictionary view.

See Also:

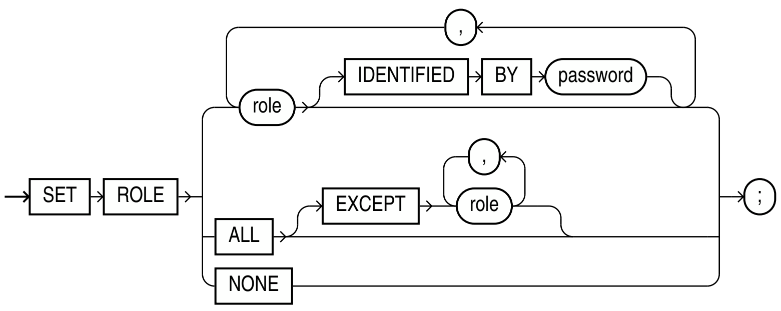
- CREATE ROLE for information on creating roles
- ALTER USER for information on changing a user's default roles
- *Oracle Database Reference* for information on the `SESSION_ROLES` session parameter

Prerequisites

You must already have been granted the roles that you name in the `SET ROLE` statement.

Syntax

***set_role*::=**



Semantics

role

Specify one or more roles to be enabled for the current session. All roles not specified are disabled for the current session or until another `SET ROLE` statement is issued in the current session.

In the `IDENTIFIED BY` *`password`* clause, specify the password for a role. If the role has a password, then you must specify the password to enable the role.

Restriction on Setting Roles

You cannot specify a role identified globally. Global roles are enabled by default at login, and cannot be reenabled later.

IDENTIFIED BY

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

ALL Clause

Specify `ALL` to enable all roles granted to you for the current session except those optionally listed in the `EXCEPT` clause.

Roles listed in the `EXCEPT` clause must be roles granted directly to you. They cannot be roles granted to you through other roles.

If you list a role in the `EXCEPT` clause that has been granted to you both directly and through another role, then the role remains enabled by virtue of the role to which it has been granted.

Restrictions on the ALL Clause

The following restrictions apply to the `ALL` clause:

- You cannot use this clause to enable roles with passwords that have been granted directly to you.
- You cannot use this clause to enable a secure application role, which is a role that can be enabled only by applications using an authorized package. Refer to *Oracle Database Security Guide* for information on creating a secure application role.

NONE

Specify `NONE` to disable all roles for the current session, including the `DEFAULT` role.

Examples

Setting Roles: Examples

To enable the role `dw_manager` identified by a password for your current session, issue the following statement:

```
SET ROLE dw_manager IDENTIFIED BY password;
```

To enable all roles granted to you for the current session, issue the following statement:

```
SET ROLE ALL;
```

To enable all roles granted to you except `dw_manager`, issue the following statement:

```
SET ROLE ALL EXCEPT dw_manager;
```

To disable all roles granted to you for the current session, issue the following statement:

```
SET ROLE NONE;
```

SET TRANSACTION

Purpose

Use the `SET TRANSACTION` statement to establish the current transaction as read-only or read/write, establish its isolation level, assign it to a specified rollback segment, or assign a name to the transaction.

A transaction implicitly begins with any operation that obtains a TX lock:

- When a statement that modifies data is issued
- When a `SELECT` ... `FOR UPDATE` statement is issued
- When a transaction is explicitly started with a `SET TRANSACTION` statement or the `DBMS_TRANSACTION` package

Issuing either a `COMMIT` or `ROLLBACK` statement explicitly ends the current transaction.

The operations performed by a SET TRANSACTION statement affect only your current transaction, not other users or other transactions. Your transaction ends whenever you issue a COMMIT or ROLLBACK statement. Oracle Database implicitly commits the current transaction before and after executing a data definition language (DDL) statement.

See Also:

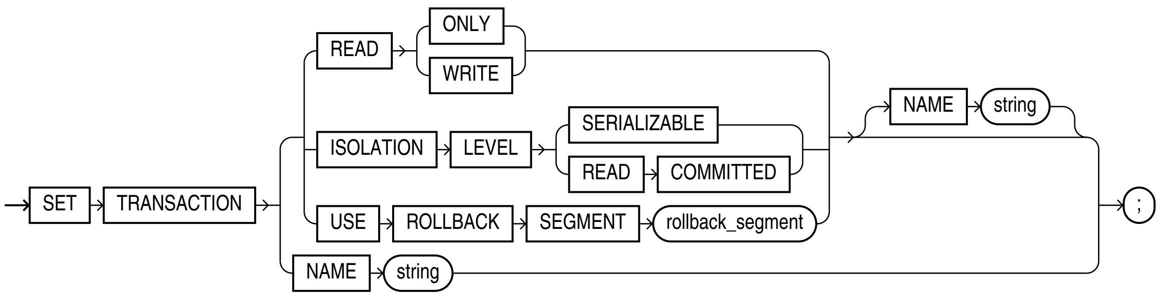
COMMIT and ROLLBACK

Prerequisites

If you use a SET TRANSACTION statement, then it must be the first statement in your transaction. However, a transaction need not have a SET TRANSACTION statement.

Syntax

***set_transaction*::=**



Semantics

READ ONLY

The READ ONLY clause establishes the current transaction as a read-only transaction. This clause established **transaction-level read consistency**.

All subsequent queries in that transaction see only changes that were committed before the transaction began. Read-only transactions are useful for reports that run multiple queries against one or more tables while other users update these same tables.

This clause is not supported for the user SYS. Queries by SYS will return changes made during the transaction even if SYS has set the transaction to be READ ONLY.

Restriction on Read-only Transactions

Only the following statements are permitted in a read-only transaction:

- Subqueries—SELECT statements without the *for_update_clause*
- LOCK TABLE
- SET ROLE
- ALTER SESSION
- ALTER SYSTEM

READ WRITE

Specify `READ WRITE` to establish the current transaction as a read/write transaction. This clause establishes **statement-level read consistency**, which is the default.

Restriction on Read/Write Transactions

You cannot toggle between transaction-level and statement-level read consistency in the same transaction.

ISOLATION LEVEL Clause

- The `SERIALIZABLE` setting specifies serializable transaction isolation mode as defined in the SQL standard. If a serializable transaction contains data manipulation language (DML) that attempts to update any resource that may have been updated in a transaction uncommitted at the start of the serializable transaction, then the DML statement fails.
- The `READ COMMITTED` setting is the default Oracle Database transaction behavior. If the transaction contains DML that requires row locks held by another transaction, then the DML statement waits until the row locks are released.

USE ROLLBACK SEGMENT Clause

Note:

This clause is relevant and valid only if you are using rollback segments for undo. Oracle strongly recommends that you use automatic undo management to handle undo space. If you follow this recommendation and run your database in automatic undo mode, then Oracle Database ignores this clause.

Specify `USE ROLLBACK SEGMENT` to assign the current transaction to the specified rollback segment. This clause also implicitly establishes the transaction as a read/write transaction.

Parallel DML requires more than one rollback segment. Therefore, if your transaction contains parallel DML operations, then the database ignores this clause.

NAME Clause

Use the `NAME` clause to assign a name to the current transaction. This clause is especially useful in distributed database environments when you must identify and resolve in-doubt transactions. The *`string`* value is limited to 255 bytes.

If you specify a name for a distributed transaction, then when the transaction commits, the name becomes the commit comment, overriding any comment specified explicitly in the `COMMIT` statement.

See Also:

Oracle Database Concepts for more information about transaction naming

Examples

Setting Transactions: Examples

The following statements could be run at midnight of the last day of every month to count the products and quantities on hand in the West Coast warehouses in the sample Order Entry (`oe`) schema. This report would not be affected by any other user who might be adding or removing inventory to a different warehouse between the running of the first query and the running of the second query.

```
COMMIT;

SET TRANSACTION READ ONLY NAME 'West Coast';

SELECT product_id, quantity_on_hand, 'San Francisco' location
  FROM inventories
    WHERE warehouse_id = 2
    ORDER BY product_id;

SELECT product_id, quantity_on_hand, 'Seattle' location
  FROM inventories
    WHERE warehouse_id = 4
    ORDER BY product_id;

COMMIT;
```

The first `COMMIT` statement ensures that `SET TRANSACTION` is the first statement in the transaction. The last `COMMIT` statement does not actually make permanent any changes to the database. It simply ends the read-only transaction.

TRUNCATE CLUSTER

Purpose

Note:

You cannot roll back a `TRUNCATE CLUSTER` statement.

Use the `TRUNCATE CLUSTER` statement to remove all rows from a cluster. By default, Oracle Database also performs the following tasks:

- Deallocates all space used by the removed rows except that specified by the `MINEXTENTS` storage parameter
- Sets the `NEXT` storage parameter to the size of the last extent removed from the segment by the truncation process

Removing rows with the `TRUNCATE` statement can be more efficient than dropping and re-creating a cluster. Dropping and re-creating a cluster invalidates dependent objects of the cluster, requires you to regrant object privileges on the cluster, and requires you to re-create the indexes and cluster on the table and respecify its storage parameters. Truncating has none of these effects.

Removing rows with the `TRUNCATE CLUSTER` statement can be faster than removing all rows with the `DELETE` statement, especially if the cluster has numerous indexes and other dependencies.

See Also:

- DELETE and DROP CLUSTER for information on other ways of dropping data from a cluster
- TRUNCATE TABLE for information on truncating a table

Prerequisites

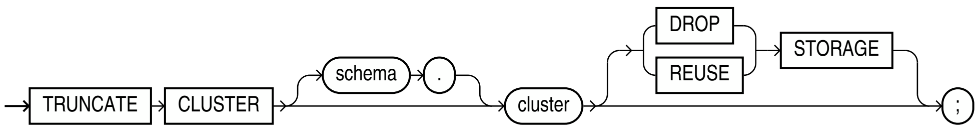
To truncate a cluster, the cluster must be in your schema or you must have `DROP ANY TABLE` system privilege.

See Also:

"Restrictions on Truncating Tables"

Syntax

truncate_cluster::=



Semantics

CLUSTER Clause

Specify the schema and name of the cluster to be truncated. You can truncate only an indexed cluster, not a hash cluster. If you omit *`schema`*, then the database assumes the cluster is in your own schema.

When you truncate a cluster, the database also automatically deletes all data in the indexes of the cluster tables.

STORAGE Clauses

The `STORAGE` clauses let you determine what happens to the space freed by the truncated rows. The `DROP STORAGE` clause and `REUSE STORAGE` clause also apply to the space freed by the data deleted from associated indexes.

DROP STORAGE

Specify `DROP STORAGE` to deallocate all space from the deleted rows from the cluster except the space allocated by the `MINEXTENTS` parameter of the cluster. This space can subsequently be used by other objects in the tablespace. Oracle Database also sets the `NEXT` storage parameter to the size of the last extent removed from the segment in the truncation process. This is the default.

REUSE STORAGE

Specify `REUSE STORAGE` to retain the space from the deleted rows allocated to the cluster. Storage values are not reset to the values when the table or cluster was created. This space can subsequently be used only by new data in the cluster resulting from insert or update operations. This clause leaves storage parameters at their current settings.

If you have specified more than one free list for the object you are truncating, then the `REUSE STORAGE` clause also removes any mapping of free lists to instances and resets the high-water mark to the beginning of the first extent.

Examples

Truncating a Cluster: Example

The following statement removes all rows from all tables in the `personnel` cluster, but leaves the freed space allocated to the tables:

```
TRUNCATE CLUSTER personnel REUSE STORAGE;
```

The preceding statement also removes all data from all indexes on the tables in the `personnel` cluster.

TRUNCATE TABLE

Purpose

> **Note:**
>
> You cannot roll back a `TRUNCATE TABLE` statement, nor can you use a `FLASHBACK TABLE` statement to retrieve the contents of a table that has been truncated.

Use the `TRUNCATE TABLE` statement to remove all rows from a table. By default, Oracle Database also performs the following tasks:

- Deallocates all space used by the removed rows except that specified by the `MINEXTENTS` storage parameter
- Sets the `NEXT` storage parameter to the size of the last extent removed from the segment by the truncation process

Removing rows with the `TRUNCATE TABLE` statement can be more efficient than dropping and re-creating a table. Dropping and re-creating a table invalidates dependent objects of the table, and requires you to repeat the following actions:

- Grant object privileges on the table
- Create the indexes, integrity constraints, and triggers on the table
- Specify the storage parameters of the table

Truncating has none of these effects.

Removing rows with the `TRUNCATE TABLE` statement can be faster than removing all rows with the `DELETE` statement, especially if the table has numerous triggers, indexes, and other dependencies.

See Also:

- DELETE and DROP TABLE for information on other ways of removing data from a table
- TRUNCATE CLUSTER for information on truncating a cluster

Prerequisites

To truncate a table, the table must be in your schema or you must have the `DROP ANY TABLE` system privilege.

To specify the `CASCADE` clause, all affected child tables must be in your schema or you must have the `DROP ANY TABLE` system privilege.

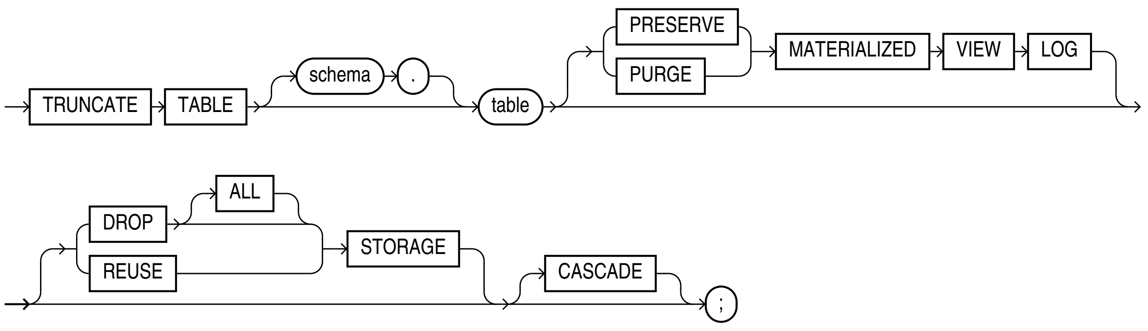
You can truncate a private temporary table with the existing `TRUNCATE TABLE` command. Truncating a private temporary table will not commit and existing transaction. This applies to both transaction-specific and session-specific private temporary tables. Note that a truncated private temporary table will not go into the `RECYCLEBIN`.

See Also:

"Restrictions on Truncating Tables"

Syntax

***truncate_table*::=**



Semantics

TABLE Clause

Specify the schema and name of the table to be truncated. This table cannot be part of a cluster. If you omit *`schema`*, then Oracle Database assumes the table is in your own schema.

- You can truncate index-organized tables and temporary tables. When you truncate a temporary table, only the rows created during the current session are removed.
- Oracle Database changes the `NEXT` storage parameter of *`table`* to be the size of the last extent deleted from the segment in the process of truncation.

- Oracle Database also automatically truncates and resets any existing `UNUSABLE` indicators for the following indexes on *`table`*: range and hash partitions of local indexes and subpartitions of local indexes.
- If *`table`* is not empty, then the database marks `UNUSABLE` all nonpartitioned indexes and all partitions of global partitioned indexes on the table. However, when the table is truncated, the index is also truncated, and a new high water mark is calculated for the index segment. This operation is equivalent to creating a new segment for the index. Therefore, at the end of the truncate operation, the indexes are once again `USABLE`.
- For a domain index, this statement invokes the appropriate truncate routine to truncate the domain index data.

> **See Also:**
>
> *Oracle Database Data Cartridge Developer's Guide* for more information on domain indexes

- If a regular or index-organized table contains LOB columns, then all LOB data and LOB index segments are truncated.
- If *`table`* is partitioned, then all partitions or subpartitions, as well as the LOB data and LOB index segments for each partition or subpartition, are truncated.

> **Note:**
>
> When you truncate a table, Oracle Database automatically removes all data in the table's indexes and any materialized view direct-path `INSERT` information held in association with the table. This information is independent of any materialized view log. If this direct-path `INSERT` information is removed, then an incremental refresh of the materialized view may lose data.

- All cursors are invalidated.

Restrictions on Truncating Tables

This statement is subject to the following restrictions:

- You cannot roll back a `TRUNCATE TABLE` statement.
- You cannot flash back to the state of the table before the truncate operation.
- You cannot individually truncate a table that is part of a cluster. You must either truncate the cluster, delete all rows from the table, or drop and re-create the table.
- You cannot truncate the parent table of an enabled foreign key constraint. You must disable the constraint before truncating the table. An exception is that you can truncate the table if the integrity constraint is self-referential.
- If a domain index is defined on *`table`*, then neither the index nor any index partitions can be marked `IN_PROGRESS`.
- You cannot truncate the parent table of a reference-partitioned table. You must first drop the reference-partitioned child table.
- You cannot truncate a duplicated table.

MATERIALIZED VIEW LOG Clause

The MATERIALIZED VIEW LOG clause lets you specify whether a materialized view log defined on the table is to be preserved or purged when the table is truncated. This clause permits materialized view master tables to be reorganized through export or import without affecting the ability of primary key materialized views defined on the master to be fast refreshed. To support continued fast refresh of primary key materialized views, the materialized view log must record primary key information.

Note:

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

PRESERVE

Specify PRESERVE if any materialized view log should be preserved when the master table is truncated. This is the default.

PURGE

Specify PURGE if any materialized view log should be purged when the master table is truncated.

See Also:

Oracle Database Administrator's Guide for more information about materialized view logs and the TRUNCATE statement

STORAGE Clauses

The STORAGE clauses let you determine what happens to the space freed by the truncated rows. The DROP STORAGE clause, DROP ALL STORAGE clause, and REUSE STORAGE clause also apply to the space freed by the data deleted from associated indexes.

DROP STORAGE

Specify DROP STORAGE to deallocate all space from the deleted rows from the table except the space allocated by the MINEXTENTS parameter of the table. This space can subsequently be used by other objects in the tablespace. Oracle Database also sets the NEXT storage parameter to the size of the last extent removed from the segment in the truncation process. This setting, which is the default, is useful for small and medium-sized objects. The extent management in locally managed tablespace is very fast in these cases, so there is no need to reserve space.

DROP ALL STORAGE

Specify DROP ALL STORAGE to deallocate all space from the deleted rows from the table, including the space allocated by the MINEXTENTS parameter. All segments for the table, as well as all segments for its dependent objects, will be deallocated.

Restrictions on DROP ALL STORAGE

This clause is subject to the same restrictions as described in "Restrictions on Deferred Segment Creation".

REUSE STORAGE

Specify `REUSE STORAGE` to retain the space from the deleted rows allocated to the table. Storage values are not reset to the values when the table was created. This space can subsequently be used only by new data in the table resulting from insert or update operations. This clause leaves storage parameters at their current settings.

This setting is useful as an alternative to deleting all rows of a very large table—when the number of rows is very large, the table entails many thousands of extents, and when data is to be reinserted in the future.

This clause is not valid for temporary tables. A session becomes unbound from the temporary table when the table is truncated, so the storage is automatically dropped.

If you have specified more than one free list for the object you are truncating, then the `REUSE STORAGE` clause also removes any mapping of free lists to instances and resets the high-water mark to the beginning of the first extent.

CASCADE

If you specify `CASCADE`, then Oracle Database truncates all child tables that reference *`table`* with an enabled `ON DELETE CASCADE` referential constraint. This is a recursive operation that will truncate all child tables, granchild tables, and so on, using the specified options.

Examples

Truncating a Table: Example

The following statement removes all rows from a hypothetical copy of the sample table `hr.employees` and returns the freed space to the tablespace containing `employees`:

```
TRUNCATE TABLE employees_demo;
```

The preceding statement also removes all data from all indexes on `employees` and returns the freed space to the tablespaces containing them.

Preserving Materialized View Logs After Truncate: Example

The following statements are examples of `TRUNCATE` statements that preserve materialized view logs:

```
TRUNCATE TABLE sales_demo PRESERVE MATERIALIZED VIEW LOG;

TRUNCATE TABLE orders_demo;
```

UPDATE

Purpose

Use the `UPDATE` statement to change existing values in a table or in the base table of a view or the master table of a materialized view.

Prerequisites

For you to update values in a table, the table must be in your own schema or you must have the `UPDATE` object privilege on the table.

For you to update values in the base table of a view:

- You must have the `UPDATE` object privilege on the view, and
- Whoever owns the schema containing the view must have the `UPDATE` object privilege on the base table.

The `UPDATE ANY TABLE` system privilege also allows you to update values in any table or in the base table of any view.

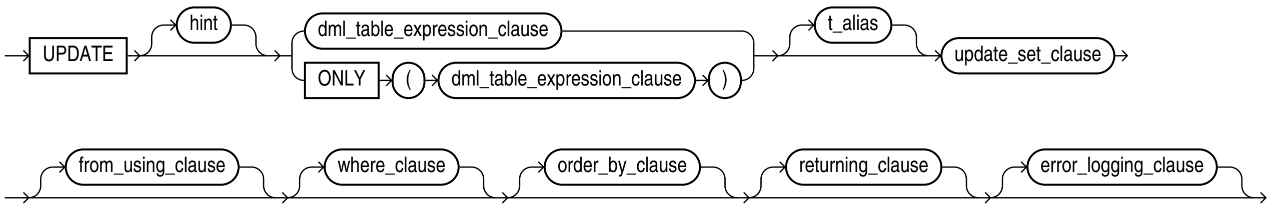
To update values in an object on a remote database, you must also have the `READ` or `SELECT` object privilege on the object.

To specify the *`returning_clause`*, you must have the `READ` or `SELECT` object privilege on the object.

If the `SQL92_SECURITY` initialization parameter is set to `TRUE` and the `UPDATE` operation references table columns, such as the columns in a *`where_clause`* or *`returning_clause`*, then you must have the `SELECT` object privilege on the object you want to update.

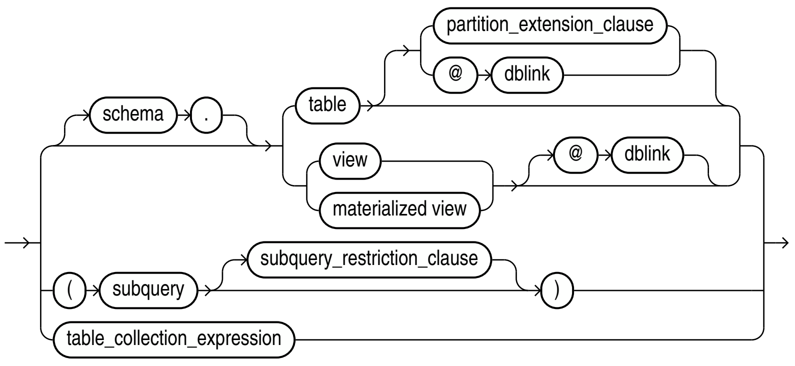
Syntax

***update*::=**



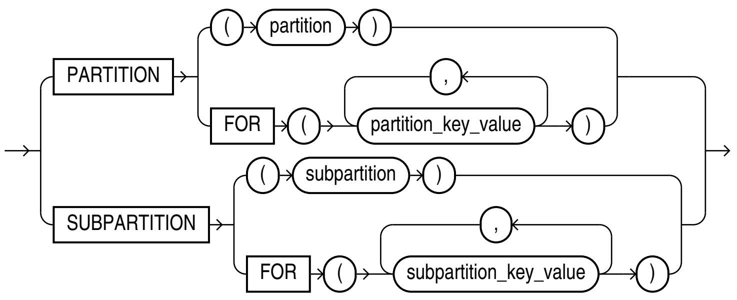
(*DML_table_expression_clause*::=, *update_set_clause*::=, *where_clause*::=, *returning_clause*::=, *error_logging_clause*::=, *from_using_clause*::=)

***DML_table_expression_clause*::=**

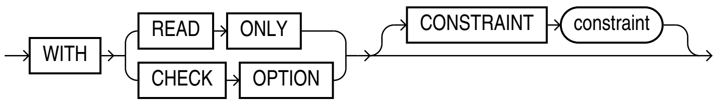

partition_extension_clause
@
dblink
schema
.
table
view
@
dblink
materialized view
subquery_restriction_clause
(
subquery
)
table_collection_expression

(*partition_extension_clause*::=, *subquery*::=--part of `SELECT`, *subquery_restriction_clause*::=, *table_collection_expression*::=)

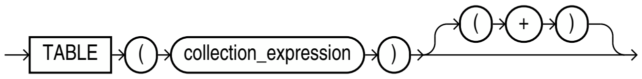
***partition_extension_clause*::=**



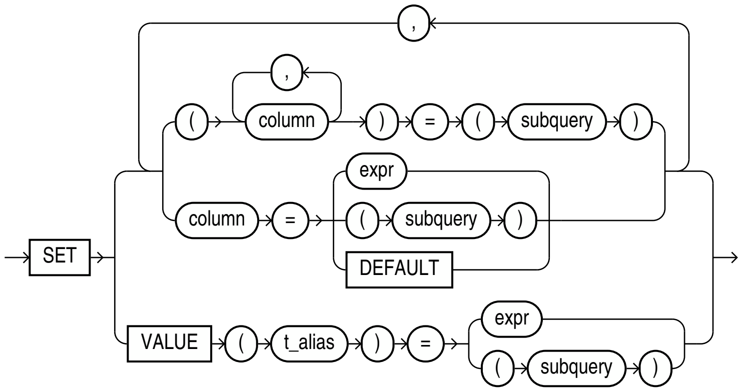
***subquery_restriction_clause*::=**



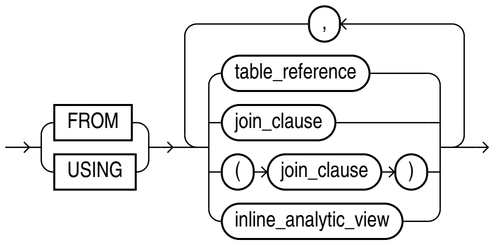
***table_collection_expression*::=**



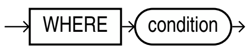
***update_set_clause*::=**



***from_using_clause*::=**



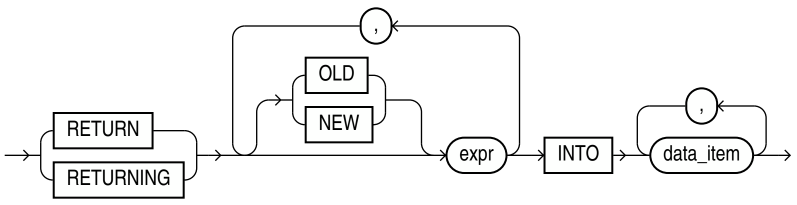
***where_clause*::=**



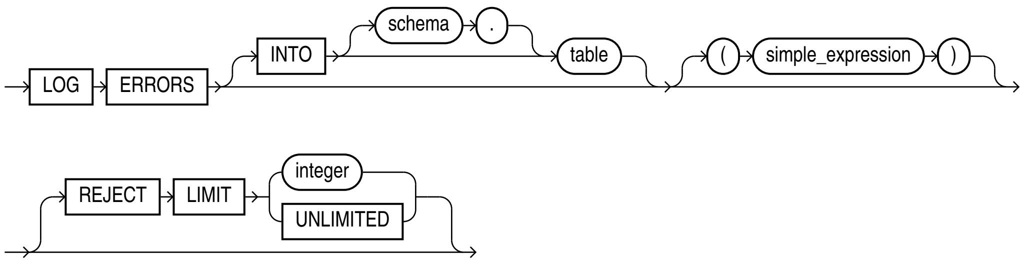
***order_by_clause*::=**

See *order_by_clause*::=

***returning_clause*::=**



***error_logging_clause*::=**



Semantics

hint

Specify a comment that passes instructions to the optimizer on choosing an execution plan for the statement.

You can place a parallel hint immediately after the `UPDATE` keyword to parallelize both the underlying scan and `UPDATE` operations.

See Also:

- "Hints " for the syntax and description of hints
- *Oracle Database Concepts* for detailed information about parallel execution

DML_table_expression_clause

The ONLY clause applies only to views. Specify ONLY syntax if the view in the UPDATE clause is a view that belongs to a hierarchy and you do not want to update rows from any of its subviews.

See Also:

"Restrictions on the DML_table_expression_clause" and "Updating a Table: Examples"

schema

Specify the schema containing the object to be updated. If you omit *schema*, then the database assumes the object is in your own schema.

table* | *view* | *materialized_view* |*subquery

Specify the name of the table, view, materialized view, or the columns returned by a subquery to be updated. Issuing an UPDATE statement against a table fires any UPDATE triggers associated with the table.

- If you specify *view*, then the database updates the base table of the view. You cannot update a view except with INSTEAD OF triggers if the defining query of the view contains one of the following constructs:

  A set operator
  A DISTINCT operator
  An aggregate or analytic function
  A GROUP BY, ORDER BY, MODEL, CONNECT BY, or START WITH clause
  A collection expression in a SELECT list
  A subquery in a SELECT list
  A subquery designated WITH READ ONLY
  A recursive WITH clause
  Joins, with some exceptions, as documented in *Oracle Database Administrator's Guide*

- You cannot update more than one base table through a view.
- In addition, if the view was created with the WITH CHECK OPTION, then you can update the view only if the resulting data satisfies the view's defining query.
- If *table* or the base table of *view* contains one or more domain index columns, then this statement executes the appropriate indextype update routine.
- You cannot update rows in a read-only materialized view. If you update rows in a writable materialized view, then the database updates the rows from the underlying container table. However, the updates are overwritten at the next refresh operation. If you update rows in an updatable materialized view that is part of a materialized view group, then the database also updates the corresponding rows in the master table.

See Also:

- *Oracle Database Data Cartridge Developer's Guide* for more information on the indextype update routines
- CREATE MATERIALIZED VIEW for information on creating updatable materialized views

partition_extension_clause

Specify the name or partition key value of the partition or subpartition within *table* targeted for updates. You need not specify the partition name when updating values in a partitioned table. However in some cases specifying the partition name can be more efficient than a complicated *where_clause*.

See Also:

"References to Partitioned Tables and Indexes " and "Updating a Partition: Example"

dblink

Specify a complete or partial name of a database link to a remote database where the object is located. You can use a database link to update a remote object only if you are using Oracle Database distributed functionality.

If you omit *dblink*, then the database assumes the object is on the local database.

Note:

Starting with Oracle Database 12*c* Release 2 (12.2), the UPDATE statement accepts remote LOB locators as bind variables. Refer to the “Distributed LOBs” chapter in *Oracle Database SecureFiles and Large Objects Developer's Guide* for more information.

See Also:

"References to Objects in Remote Databases " for information on referring to database links

subquery_restriction_clause

Use the *subquery_restriction_clause* to restrict the subquery in one of the following ways:

WITH READ ONLY

Specify WITH READ ONLY to indicate that the table or view cannot be updated.

WITH CHECK OPTION

Specify `WITH CHECK OPTION` to indicate that Oracle Database prohibits any changes to the table or view that would produce rows that are not included in the subquery. When used in the subquery of a DML statement, you can specify this clause in a subquery in the `FROM` clause but not in subquery in the `WHERE` clause.

CONSTRAINT *constraint*

Specify the name of the `CHECK OPTION` constraint. If you omit this identifier, then Oracle automatically assigns the constraint a name of the form `SYS_C`*`n`*, where n is an integer that makes the constraint name unique within the database.

See Also:

"Using the WITH CHECK OPTION Clause: Example"

table_collection_expression

The *`table_collection_expression`* lets you inform Oracle that the value of *`collection_expression`* should be treated as a table for purposes of query and DML operations. The *`collection_expression`* can be a subquery, a column, a function, or a collection constructor. Regardless of its form, it must return a collection value—that is, a value whose type is nested table or varray. This process of extracting the elements of a collection is called **collection unnesting**.

The optional plus (+) is relevant if you are joining the `TABLE` collection expression with the parent table. The + creates an outer join of the two, so that the query returns rows from the outer table even if the collection expression is null.

Note:

In earlier releases of Oracle, when *`collection_expression`* was a subquery, *`table_collection_expression`* was expressed as `THE` *`subquery`*. That usage is now deprecated.

You can use a *`table_collection_expression`* to update rows in one table based on rows from another table. For example, you could roll up four quarterly sales tables into a yearly sales table.

t_alias

Specify a **correlation name** (alias) for the table, view, or subquery to be referenced elsewhere in the statement. This alias is required if the *`DML_table_expression_clause`* references any object type attributes or object type methods.

See Also:

"Correlated Update: Example"

Restrictions on the *DML_table_expression_clause*

This clause is subject to the following restrictions:

- You cannot execute this statement if *table* or the base table of *view* contains any domain indexes marked IN_PROGRESS or FAILED.
- You cannot insert into a partition if any affected index partitions are marked UNUSABLE.
- You cannot specify the *order_by_clause* in the subquery of the *DML_table_expression_clause*.
- If you specify an index, index partition, or index subpartition that has been marked UNUSABLE, then the UPDATE statement will fail unless the SKIP_UNUSABLE_INDEXES session parameter has been set to TRUE.

See Also:

ALTER SESSION for information on the SKIP_UNUSABLE_INDEXES session parameter

update_set_clause

The *update_set_clause* lets you set column values.

column

Specify the name of a column of the object that is to be updated. If you omit a column of the table from the *update_set_clause*, then the value of that column remains unchanged.

If *column* refers to a LOB object attribute, then you must first initialize it with a value of empty or null. You cannot update it with a literal. Also, if you are updating a LOB value using some method other than a direct UPDATE SQL statement, then you must first lock the row containing the LOB. See *for_update_clause* for more information.

If *column* is a virtual column, you cannot specify it here. Rather, you must update the values from which the virtual column is derived.

If *column* is part of the partitioning key of a partitioned table, then UPDATE will fail if you change a value in the column that would move the row to a different partition or subpartition, unless you enable row movement. Refer to the *row_movement_clause* of CREATE TABLE or ALTER TABLE.

In addition, if *column* is part of the partitioning key of a list-partitioned table, then UPDATE will fail if you specify a value for the column that does not already exist in the *partition_key_value* list of one of the partitions.

subquery

Specify a subquery that returns exactly one row for each row updated.

- If you specify only one column in the *update_set_clause*, then the subquery can return only one value.
- If you specify multiple columns in the *update_set_clause*, then the subquery must return as many values as you have specified columns.
- If the subquery returns no rows, then the column is assigned a null.
- If this *subquery* refers to remote objects, then the UPDATE operation can run in parallel as long as the reference does not loop back to an object on the local database. However, if

the *subquery* in the *DML_table_expression_clause* refers to any remote objects, then the `UPDATE` operation will run serially without notification.

You can use the *flashback_query_clause* within the subquery to update *table* with past data. Refer to the *flashback_query_clause* of `SELECT` for more information on this clause.

See Also:

- SELECT and "Using Subqueries "
- parallel_clause in the CREATE TABLE documentation

expr

Specify an expression that resolves to the new value assigned to the corresponding column.

Note:

Expressions for the syntax of *expr* and "Updating an Object Table: Example"

DEFAULT

Specify `DEFAULT` to set the column to the value previously specified as the default value for the column. If no default value for the corresponding column has been specified, then the database sets the column to null.

Restriction on Updating to Default Values

You cannot specify `DEFAULT` if you are updating a view.

You cannot use the `DEFAULT` clause in an `UPDATE` statement if the table that you are specifying has an Oracle Label Security policy enabled.

VALUE Clause

The `VALUE` clause lets you specify the entire row of an object table.

Restriction on the VALUE clause

You can specify this clause only for an object table.

Note:

If you insert string literals into a `RAW` column, then during subsequent queries, Oracle Database will perform a full table scan rather than using any index that might exist on the `RAW` column.

See Also:

"Updating an Object Table: Example"

from_using_clause

Use this clause to filter the rows `UPDATE` changes, or to provide the values for the columns in the target table. Specify the join conditions in the *`where_clause`*. You can outer join source tables to the target table with (+). The target table cannot be the outer table in the join.

You can join many tables, views, and inline views. Specify the join conditions in the *`where_clause`* or use the *`join_clause`* to join these to each other with ANSI join syntax.

You can specify the same table in the *`dml_table_expression_clause`* and *`from_clause`*. When you do so they must have unique aliases.

Example: Update With Direct-Join

In this example, the join condition between table employees `e` and table jobs `j` determines which rows of employees are updated. The column `jobs.max_salary` supplies the new values for `employees.salary`:

```
UPDATE employees e
SET    e.salary = j.max_salary
FROM   jobs j
WHERE  j.job_id = e.job_id;
```

Direct joins for `UPDATE` have the same semantics and restrictions as `SELECT` in the *`from_clause`* and *`where_clause`*. The target table has the same restrictions as `UPDATE`. Triggers on the target table fire as normal.

Restrictions

- You cannot specify ANSI join syntax involving the *`dml_table_expression_clause`*. However, ANSI join syntax is allowed between the tables specified in the `FROM` clause. Right and full outer joins are not allowed.
- The `UPDATE` can change each row at most once. If the join condition results in the same row being updated more than once, the statement will raise an `ORA-30926` error.
- You can only specify one table, view, or materialized view in *`dml_table_expression_clause`* when the *`from_clause`* is present.
- The left-hand side of *`update_set_clause`* must be a column from the *`dml_table_expression_clause`* and not from the *`from_clause`*.
- You can use a lateral view in the `FROM` clause, but it cannot reference a column from the update target. It may be outer-joined.
- Order by position is not allowed in the *`order_by_clause`*.
- `UPDATE` with *`from_clause`* supports *`returning_clause`* and *`error_logging_clause`*.
- Hint clause can be used to specify instructions to the optimizer for joins involving the *`from_clause`*.

where_clause

The *`where_clause`* lets you restrict the rows updated to those for which the specified *`condition`* is true. If you omit this clause, then the database updates all rows in the table or view. Refer to Conditions for the syntax of *`condition`*.

The *`where_clause`* determines the rows in which values are updated. If you do not specify the *`where_clause`*, then all rows are updated. For each row that satisfies the *`where_clause`*, the columns to the left of the equality operator (=) in the *`update_set_clause`* are set to the values of the corresponding expressions to the right of the operator. The expressions are evaluated as the row is updated.

order_by_clause

The following restrictions apply to the `ORDER BY` clause:

- When used in an analytic function, the *`order_by_clause`* must take an expression *`expr`*.
- The `SIBLINGS` keyword is not valid (it is relevant only in hierarchical queries)
- The *`position`* alias is invalid.

See *order_by_clause*

returning_clause

You can specify this clause for tables, views, and materialized views with a single base table.

When operating on a single row, a DML statement with a *`returning_clause`* can retrieve column values using the affected row, rowid, and `REF`s to the affected row and store them in host variables or PL/SQL variables.

When operating on multiple rows, a DML statement with the *`returning_clause`* returns values from expressions, rowids, and `REF`s involving the affected rows in bind arrays.

expr

Each item in the *`expr`* list must be a valid expression syntax.

INTO

The `INTO` clause indicates that the values of the changed rows are to be stored in the variable(s) specified in *`data_item`* list.

data_item

Each *`data_item`* is a host variable or PL/SQL variable that stores the retrieved *`expr`* value.

For each expression in the `RETURNING` list, you must specify a corresponding type-compatible PL/SQL variable or host variable in the `INTO` list.

Given columns `c1` and `c2` in a table, you can specify `OLD` for a column `c1`, (for example `OLD c1`). You can also specify `OLD` for a column referenced by a column expression (for example `c1+OLD c2`). When `OLD` is specified for a column, the column value before the update is returned. In the case of a column referenced by a column expression, what is returned is the result from evaluating the column expression using the column value before the update.

`NEW` can be explicitly specified for a column, or column referenced in an expression to return a column value after the update, or an expression result that uses the after update value of a column.

When OLD and NEW are both omitted for a column or an expression, the after update column value, or expression result computed using after update column values, is returned.

Restrictions

The following restrictions apply to the RETURNING clause:

- The *expr* is restricted as follows:
  - For UPDATE and DELETE statements each *expr* must be a simple expression or a single-set aggregate function expression. You cannot combine simple expressions and single-set aggregate function expressions in the same *returning_clause*. For INSERT statements, each *expr* must be a simple expression. Aggregate functions are not supported in an INSERT statement RETURNING clause.
  - Single-set aggregate function expressions cannot include the DISTINCT keyword.
- If the *expr* list contains a primary key column or other NOT NULL column, then the update statement fails if the table has a BEFORE UPDATE trigger defined on it.
- You cannot specify the *returning_clause* for a multitable insert.
- You cannot use this clause with parallel DML or with remote objects.
- You cannot retrieve LONG types with this clause.
- You cannot specify this clause for a view on which an INSTEAD OF trigger has been defined.

See Also:

Oracle Database PL/SQL Language Reference for information on using the BULK COLLECT clause to return multiple values to collection variables

error_logging_clause

The *error_logging_clause* has the same behavior in an UPDATE statement as it does in an INSERT statement. Refer to the INSERT statement *error_logging_clause* for more information.

See Also:

"Inserting Into a Table with Error Logging: Example"

Examples

Updating a Table: Examples

The following statement gives null commissions to all employees with the job SH_CLERK:

```
UPDATE employees
   SET commission_pct = NULL
   WHERE job_id = 'SH_CLERK';
```

The following statement promotes Douglas Grant to manager of Department 20 with a $1,000 raise:

```
UPDATE employees SET
    job_id = 'SA_MAN', salary = salary + 1000, department_id = 120
    WHERE first_name||' '||last_name = 'Douglas Grant';
```

The following statement increases the salary of an employee in the `employees` table on the `remote` database:

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

The next example shows the following syntactic constructs of the `UPDATE` statement:

- Both forms of the *`update_set_clause`* together in a single statement
- A correlated subquery
- A *`where_clause`* to limit the updated rows

```
UPDATE employees a
    SET department_id =
        (SELECT department_id
            FROM departments
            WHERE location_id = '2100'),
        (salary, commission_pct) =
        (SELECT 1.1*AVG(salary), 1.5*AVG(commission_pct)
          FROM employees b
          WHERE a.department_id = b.department_id)
    WHERE department_id IN
        (SELECT department_id
          FROM departments
          WHERE location_id = 2900
              OR location_id = 2700);
```

The preceding `UPDATE` statement performs the following operations:

- Updates only those employees who work in Geneva or Munich (locations 2900 and 2700)
- Sets `department_id` for these employees to the `department_id` corresponding to Bombay (`location_id` 2100)
- Sets each employee's salary to 1.1 times the average salary of their department
- Sets each employee's commission to 1.5 times the average commission of their department

Updating a Partition: Example

The following example updates values in a single partition of the `sales` table:

```
UPDATE sales PARTITION (sales_q1_1999) s
   SET s.promo_id = 494
   WHERE amount_sold > 1000;
```

Updating an Object Table: Example

The following statement creates two object tables, `people_demo1` and `people_demo2`, of the `people_typ` object created in Table Collections: Examples. The example shows how to update a row of `people_demo1` by selecting a row from `people_demo2`:

```
CREATE TABLE people_demo1 OF people_typ;

CREATE TABLE people_demo2 OF people_typ;
```

```
UPDATE people_demo1 p SET VALUE(p) =
   (SELECT VALUE(q) FROM people_demo2 q
    WHERE p.department_id = q.department_id)
   WHERE p.department_id = 10;
```

The example uses the `VALUE` object reference function in both the `SET` clause and the subquery.

Correlated Update: Example

For an example that uses a correlated subquery to update nested table rows, refer to "Table Collections: Examples".

Using the RETURNING Clause During UPDATE: Example

The following example returns values from the updated row and stores the result in PL/SQL variables `bnd1`, `bnd2`, `bnd3`:

```
UPDATE employees
  SET job_id ='SA_MAN', salary = salary + 1000, department_id = 140
  WHERE last_name = 'Jones'
  RETURNING salary*0.25, last_name, department_id
    INTO :bnd1, :bnd2, :bnd3;
```

The following example shows that you can specify a single-set aggregate function in the expression of the returning clause:

```
UPDATE employees
   SET salary = salary * 1.1
   WHERE department_id = 100
   RETURNING SUM(salary) INTO :bnd1;
```

Update Using Direct Join: Example

The following example sets every employee's salary to the max salary for their job:

```
UPDATE hr.employees e
    SET e.salary = j.max_salary
    FROM hr.jobs j
    WHERE e.job_id = j.job_id;
```