9

SQL Queries and Subqueries

This chapter describes SQL queries and subqueries.

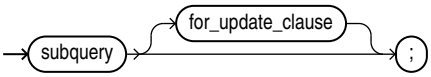
This chapter contains these sections:

About Queries and Subqueries

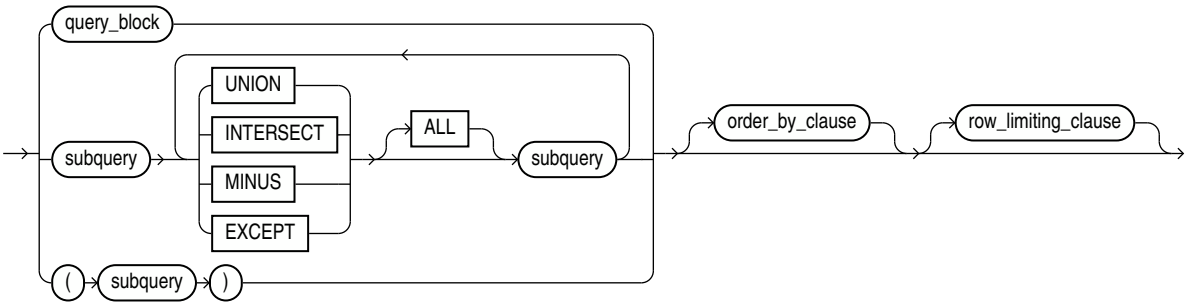
A **query** is an operation that retrieves data from one or more tables or views. In this reference, a top-level `SELECT` statement is called a **query**, and a query nested within another SQL statement is called a **subquery**.

This section describes some types of queries and subqueries and how to use them. The top level of the syntax is shown in this chapter. Refer to SELECT for the full syntax of all the clauses and the semantics of this statement.

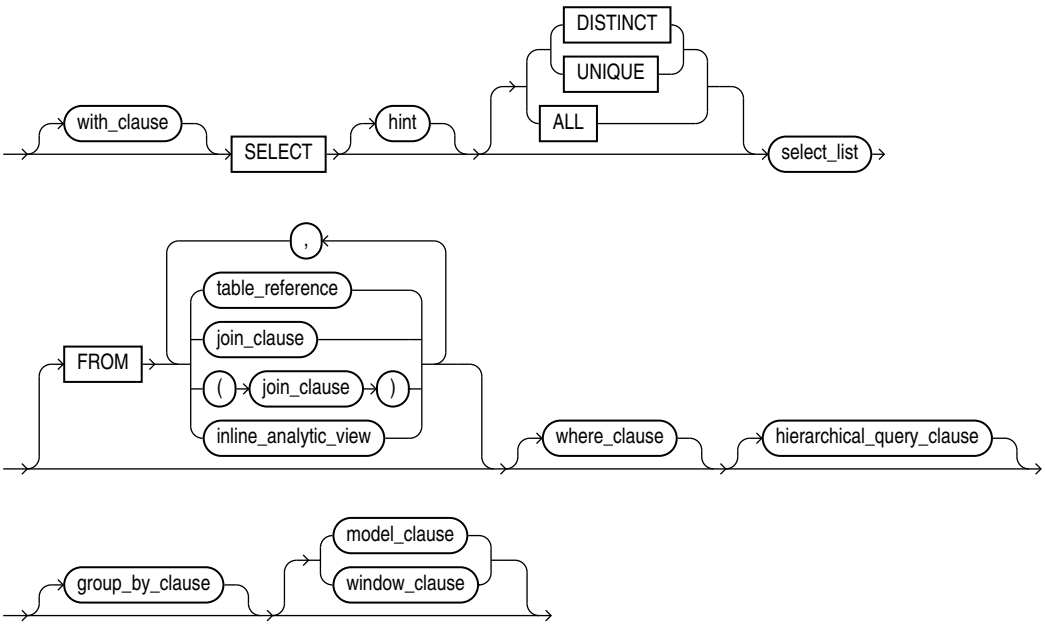
***select*::=**



***subquery*::=**



***query_block*::=**



Creating Simple Queries

The list of expressions that appears after the `SELECT` keyword and before the `FROM` clause is called the **select list**. Within the select list, you specify one or more columns in the set of rows you want Oracle Database to return from one or more tables, views, or materialized views. The number of columns, as well as their data type and length, are determined by the elements of the select list.

If two or more tables have some column names in common, then you must qualify column names with names of tables. Otherwise, fully qualified column names are optional. However, it is always a good idea to qualify table and column references explicitly. Oracle often does less work with fully qualified table and column names.

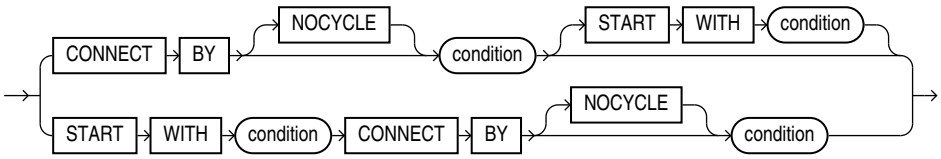
You can use a column alias, *`c_alias`*, to label the immediately preceding expression in the select list so that the column is displayed with a new heading. The alias effectively renames the select list item for the duration of the query. The alias can be used in the `ORDER BY` clause, but not other clauses in the query.

You can use comments in a `SELECT` statement to pass instructions, or **hints**, to the Oracle Database optimizer. The optimizer uses hints to choose an execution plan for the statement. Refer to "Hints " for more information on hints.

Hierarchical Queries

If a table contains hierarchical data, then you can select rows in a hierarchical order using the hierarchical query clause:

***hierarchical_query_clause*::=**



condition can be any condition as described in Conditions.

START WITH specifies the root row(s) of the hierarchy.

CONNECT BY specifies the relationship between parent rows and child rows of the hierarchy.

- The NOCYCLE parameter instructs Oracle Database to return rows from a query even if a CONNECT BY loop exists in the data. Use this parameter along with the CONNECT_BY_ISCYCLE pseudocolumn to see which rows contain the loop. Refer to CONNECT_BY_ISCYCLE Pseudocolumn for more information.
- In a hierarchical query, one expression in *condition* must be qualified with the PRIOR operator to refer to the parent row. For example,

  ```
  ... PRIOR expr = expr
  or
  ... expr = PRIOR expr
  ```

  If the CONNECT BY *condition* is compound, then only one condition requires the PRIOR operator, although you can have multiple PRIOR conditions. For example:

  ```
  CONNECT BY last_name != 'King' AND PRIOR employee_id = manager_id ...
  CONNECT BY PRIOR employee_id = manager_id and
             PRIOR account_mgr_id = customer_id ...
  ```

  PRIOR is a unary operator and has the same precedence as the unary + and - arithmetic operators. It evaluates the immediately following expression for the parent row of the current row in a hierarchical query.

  PRIOR is most commonly used when comparing column values with the equality operator. (The PRIOR keyword can be on either side of the operator.) PRIOR causes Oracle to use the value of the parent row in the column. Operators other than the equal sign (=) are theoretically possible in CONNECT BY clauses. However, the conditions created by these other operators can result in an infinite loop through the possible combinations. In this case Oracle detects the loop at run time and returns an error.

Both the CONNECT BY condition and the PRIOR expression can take the form of an uncorrelated subquery. However, CURRVAL and NEXTVAL are not valid PRIOR expressions, so the PRIOR expression cannot refer to a sequence.

You can further refine a hierarchical query by using the CONNECT_BY_ROOT operator to qualify a column in the select list. This operator extends the functionality of the CONNECT BY [PRIOR] condition of hierarchical queries by returning not only the immediate parent row but all ancestor rows in the hierarchy.

> **See Also:**
>
> CONNECT_BY_ROOT for more information about this operator and "Hierarchical Query Examples"

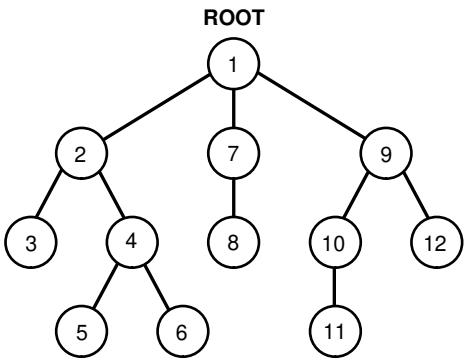
Oracle processes hierarchical queries as follows:

- 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.

Oracle then uses the information from these evaluations to form the hierarchy using the following steps:

1. Oracle selects the root row(s) of the hierarchy—those rows that satisfy the `START WITH` condition.
2. Oracle selects the child rows of each root row. Each child row must satisfy the condition of the `CONNECT BY` condition with respect to one of the root rows.
3. Oracle selects successive generations of child rows. Oracle first selects the children of the rows returned in step 2, and then the children of those children, and so on. Oracle always selects children by evaluating the `CONNECT BY` condition with respect to a current parent row.
4. If the query contains a `WHERE` clause without a join, then Oracle eliminates all rows from the hierarchy that do not satisfy the condition of the `WHERE` clause. Oracle evaluates this condition for each row individually, rather than removing all the children of a row that does not satisfy the condition.
5. Oracle returns the rows in the order shown in Figure 9-1. In the diagram, children appear below their parents. For an explanation of hierarchical trees, see Figure 3-1.

Figure 9-1 Hierarchical Queries



To find the children of a parent row, Oracle evaluates the `PRIOR` expression of the `CONNECT BY` condition for the parent row and the other expression for each row in the table. Rows for which the condition is true are the children of the parent. The `CONNECT BY` condition can contain other conditions to further filter the rows selected by the query.

If the CONNECT BY condition results in a loop in the hierarchy, then Oracle returns an error. A loop occurs if one row is both the parent (or grandparent or direct ancestor) and a child (or a grandchild or a direct descendent) of another row.

> **Note:**
>
> In a hierarchical query, do not specify either ORDER BY or GROUP BY, as they will override 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 *order_by_clause*.

Hierarchical Query Examples

CONNECT BY Example

The following hierarchical query uses the CONNECT BY clause to define the relationship between employees and managers:

```
SELECT employee_id, last_name, manager_id
   FROM employees
   CONNECT BY PRIOR employee_id = manager_id;

EMPLOYEE_ID LAST_NAME                 MANAGER_ID
----------- ------------------------- ----------
        101 Kochhar                           100
        108 Greenberg                         101
        109 Faviet                            108
        110 Chen                              108
        111 Sciarra                           108
        112 Urman                             108
        113 Popp                              108
        200 Whalen                            101
        203 Mavris                            101
        204 Baer                              101
. . .
```

LEVEL Example

The next example is similar to the preceding example, but uses the LEVEL pseudocolumn to show parent and child rows:

```
SELECT employee_id, last_name, manager_id, LEVEL
   FROM employees
   CONNECT BY PRIOR employee_id = manager_id;

EMPLOYEE_ID LAST_NAME                 MANAGER_ID      LEVEL
----------- ------------------------- ---------- ----------
        101 Kochhar                           100          1
        108 Greenberg                         101          2
        109 Faviet                            108          3
        110 Chen                              108          3
        111 Sciarra                           108          3
        112 Urman                             108          3
        113 Popp                              108          3
        200 Whalen                            101          2
        203 Mavris                            101          2
        204 Baer                              101          2
```

```
        205 Higgins                         101          2
        206 Gietz                           205          3
        102 De Haan                         100          1
...
```

START WITH Examples

The next example adds a `START WITH` clause to specify a root row for the hierarchy and an `ORDER BY` clause using the `SIBLINGS` keyword to preserve ordering within the hierarchy:

```
SELECT last_name, employee_id, manager_id, LEVEL
      FROM employees
      START WITH employee_id = 100
      CONNECT BY PRIOR employee_id = manager_id
      ORDER SIBLINGS BY last_name;

LAST_NAME                 EMPLOYEE_ID MANAGER_ID      LEVEL
------------------------- ----------- ---------- ----------
King                              100                     1
Cambrault                         148        100          2
Bates                             172        148          3
Bloom                             169        148          3
Fox                               170        148          3
Kumar                             173        148          3
Ozer                              168        148          3
Smith                             171        148          3
De Haan                           102        100          2
Hunold                            103        102          3
Austin                            105        103          4
Ernst                             104        103          4
Lorentz                           107        103          4
Pataballa                         106        103          4
Errazuriz                         147        100          2
Ande                              166        147          3
Banda                             167        147          3
...
```

In the `hr.employees` table, the employee Steven King is the head of the company and has no manager. Among his employees is John Russell, who is the manager of department 80. If you update the `employees` table to set Russell as King's manager, you create a loop in the data:

```
UPDATE employees SET manager_id = 145
   WHERE employee_id = 100;

SELECT last_name "Employee",
   LEVEL, SYS_CONNECT_BY_PATH(last_name, '/') "Path"
   FROM employees
   WHERE level <= 3 AND department_id = 80
   START WITH last_name = 'King'
   CONNECT BY PRIOR employee_id = manager_id AND LEVEL <= 4;

ERROR:
ORA-01436: CONNECT BY loop in user data
```

The `NOCYCLE` parameter in the `CONNECT BY` condition causes Oracle to return the rows in spite of the loop. The `CONNECT_BY_ISCYCLE` pseudocolumn shows you which rows contain the cycle:

```
SELECT last_name "Employee", CONNECT_BY_ISCYCLE "Cycle",
   LEVEL, SYS_CONNECT_BY_PATH(last_name, '/') "Path"
   FROM employees
   WHERE level <= 3 AND department_id = 80
   START WITH last_name = 'King'
```

```
   CONNECT BY NOCYCLE PRIOR employee_id = manager_id AND LEVEL <= 4
   ORDER BY "Employee", "Cycle", LEVEL, "Path";

Employee                       Cycle      LEVEL Path
------------------------- ---------- ---------- -------------------------
Abel                                0          3 /King/Zlotkey/Abel
Ande                                0          3 /King/Errazuriz/Ande
Banda                               0          3 /King/Errazuriz/Banda
Bates                               0          3 /King/Cambrault/Bates
Bernstein                           0          3 /King/Russell/Bernstein
Bloom                               0          3 /King/Cambrault/Bloom
Cambrault                           0          2 /King/Cambrault
Cambrault                           0          3 /King/Russell/Cambrault
Doran                               0          3 /King/Partners/Doran
Errazuriz                           0          2 /King/Errazuriz
Fox                                 0          3 /King/Cambrault/Fox
...
```

CONNECT_BY_ISLEAF Example

The following statement shows how you can use a hierarchical query to turn the values in a column into a comma-delimited list:

```
SELECT LTRIM(SYS_CONNECT_BY_PATH (warehouse_id,','),',') FROM
   (SELECT ROWNUM r, warehouse_id FROM warehouses)
   WHERE CONNECT_BY_ISLEAF = 1
   START WITH r = 1
   CONNECT BY r = PRIOR r + 1
   ORDER BY warehouse_id;

LTRIM(SYS_CONNECT_BY_PATH(WAREHOUSE_ID,','),',')
--------------------------------------------------------------------------------
1,2,3,4,5,6,7,8,9
```

CONNECT_BY_ROOT Examples

The following example returns the last name of each employee in department 110, each manager at the highest level above that employee in the hierarchy, the number of levels between manager and employee, and the path between the two:

```
SELECT last_name "Employee", CONNECT_BY_ROOT last_name "Manager",
   LEVEL-1 "Pathlen", SYS_CONNECT_BY_PATH(last_name, '/') "Path"
   FROM employees
   WHERE LEVEL > 1 and department_id = 110
   CONNECT BY PRIOR employee_id = manager_id
   ORDER BY "Employee", "Manager", "Pathlen", "Path";

Employee        Manager            Pathlen Path
--------------- --------------- ---------- ------------------------------
Gietz           Higgins                  1 /Higgins/Gietz
Gietz           King                     3 /King/Kochhar/Higgins/Gietz
Gietz           Kochhar                  2 /Kochhar/Higgins/Gietz
Higgins         King                     2 /King/Kochhar/Higgins
Higgins         Kochhar                  1 /Kochhar/Higgins
```

The following example uses a `GROUP BY` clause to return the total salary of each employee in department 110 and all employees above that employee in the hierarchy:

```
SELECT name, SUM(salary) "Total_Salary" FROM (
   SELECT CONNECT_BY_ROOT last_name as name, Salary
      FROM employees
      WHERE department_id = 110
```

```
      CONNECT BY PRIOR employee_id = manager_id)
      GROUP BY name
   ORDER BY name, "Total_Salary";

NAME                     Total_Salary
------------------------ ------------
Gietz                            8300
Higgins                         20300
King                            20300
Kochhar                         20300
```

See Also:

- LEVEL Pseudocolumn and CONNECT_BY_ISCYCLE Pseudocolumn for a discussion of how these pseudocolumns operate in a hierarchical query
- SYS_CONNECT_BY_PATH for information on retrieving the path of column values from root to node
- *order_by_clause* for more information on the `SIBLINGS` keyword of `ORDER BY` clauses
- subquery_factoring_clause, which supports recursive subquery factoring (recursive WITH) and lets you query hierarchical data. This feature is more powerful than `CONNECT BY` in that it provides depth-first search and breadth-first search, and supports multiple recursive branches.

The Set Operators

You can combine multiple queries using the set operators `UNION`, `UNION ALL`, `INTERSECT`, `INTERSECT ALL`, `EXCEPT`, `EXCEPT ALL`, `MINUS`, and `MINUS ALL`. All set operators have equal precedence. If a SQL statement contains multiple set operators, then Oracle Database evaluates them from the left to right unless parentheses explicitly specify another order.

The corresponding expressions in the select lists of the component queries of a compound query must match in number and must be in the same data type group (such as numeric or character).

If component queries select character data, then the data type of the return values are determined as follows:

- If both queries select values of data type `CHAR` of equal length, then the returned values have data type `CHAR` of that length. If the queries select values of `CHAR` with different lengths, then the returned value is `VARCHAR2` with the length of the larger `CHAR` value.
- If either or both of the queries select values of data type `VARCHAR2`, then the returned values have data type `VARCHAR2`.

If component queries select numeric data, then the data type of the return values is determined by numeric precedence:

- If any query selects values of type `BINARY_DOUBLE`, then the returned values have data type `BINARY_DOUBLE`.
- If no query selects values of type `BINARY_DOUBLE` but any query selects values of type `BINARY_FLOAT`, then the returned values have data type `BINARY_FLOAT`.

- If all queries select values of type NUMBER, then the returned values have data type NUMBER.

In queries using set operators, Oracle does not perform implicit conversion across data type groups. Therefore, if the corresponding expressions of component queries resolve to both character data and numeric data, Oracle returns an error.

The INTERSECT operator with the keyword ALL returns the result of two or more SELECT statements in which rows appear in all result sets. Null values that are common across the component queries of INTERSECT ALL are returned at the end of the result set.

The MINUS operator with the keyword ALL returns the result of two SELECT statements in which rows appear in the first result set but not in the second result set.

If the first query has x nulls and the second query has y nulls, and x is greater than y, then x minus y NULLS are returned at the end of the result query set. MINUS ALL returns no rows if the result set returned by the first SELECTstatement is a subset of the result set returned by the second SELECT.

The EXCEPT operator is a synonym for MINUS and has the exact same semantics. EXCEPT ALL returns rows that are present in the first result set but not in the second. However, duplicates may be present in the final result.

EXCEPT ALL, MINUS ALL INTERSECT ALL return equivalent instead of the original value, when NLS_SORT=BINARY_CI[AI] is acceptable for the SQL standard.

> **See Also:**
>
> Table 2-9 for more information on implicit conversion and "Numeric Precedence " for information on numeric precedence

Examples for Valid and Invalid Data Type Conversions for Set Operators

The following query is valid:

```
SELECT 3 FROM DUAL
   INTERSECT
SELECT 3f FROM DUAL;
```

This is implicitly converted to the following compound query:

```
SELECT TO_BINARY_FLOAT(3) FROM DUAL
   INTERSECT
SELECT 3f FROM DUAL;
```

The following query returns an error:

```
SELECT '3' FROM DUAL
   INTERSECT
SELECT 3f FROM DUAL;
```

Restrictions on the Set Operators

The set operators are subject to the following restrictions:

- The set operators are not valid on columns of type BLOB, CLOB, BFILE, VARRAY, or nested table.
- The UNION, INTERSECT, EXCEPT, and MINUS operators are not valid on LONG columns.

- If the select list preceding the set operator contains an expression, then you must provide a column alias for the expression in order to refer to it in the *order_by_clause*.
- You cannot also specify the *for_update_clause* with the set operators.
- You cannot specify the *order_by_clause* in the *subquery* of these operators.
- You cannot use these operators in SELECT statements containing TABLE collection expressions.

> **Note:**
>
> To comply with emerging SQL standards, a future release of Oracle will give the INTERSECT operator greater precedence than the other set operators. Therefore, you should use parentheses to specify order of evaluation in queries that use the INTERSECT operator with other set operators.

UNION Example

The following statement combines the results of two queries with the UNION operator, which eliminates duplicate selected rows. This statement shows that you must match data type (using the TO_CHAR function) when columns do not exist in one or the other table:

```
SELECT location_id, department_name "Department",
   TO_CHAR(NULL) "Warehouse"  FROM departments
   UNION
   SELECT location_id, TO_CHAR(NULL) "Department", warehouse_name
   FROM warehouses;

LOCATION_ID Department                     Warehouse
----------- ------------------------------ --------------------------
       1400 IT
       1400                               Southlake, Texas
       1500 Shipping
       1500                               San Francisco
       1600                               New Jersey
       1700 Accounting
       1700 Administration
       1700 Benefits
       1700 Construction
       1700 Contracting
       1700 Control And Credit
...
```

UNION ALL Example

The UNION operator returns only distinct rows that appear in either result, while the UNION ALL operator returns all rows. The UNION ALL operator does not eliminate duplicate selected rows:

```
SELECT product_id FROM order_items
UNION
SELECT product_id FROM inventories
ORDER BY product_id;

SELECT location_id  FROM locations
UNION ALL
SELECT location_id  FROM departments
ORDER BY location_id;
```

A location_id value that appears multiple times in either or both queries (such as '1700') is returned only once by the UNION operator, but multiple times by the UNION ALL operator.

INTERSECT Example

The following statement combines the results with the INTERSECT operator, which returns only those unique rows returned by both queries:

```
SELECT product_id FROM inventories
INTERSECT
SELECT product_id FROM order_items
ORDER BY product_id;
```

MINUS Example

The following statement combines results with the MINUS operator, which returns only unique rows returned by the first query but not by the second:

```
SELECT product_id FROM inventories
MINUS
SELECT product_id FROM order_items
ORDER BY product_id;
```

EXCEPT Example

You can use EXCEPT or MINUS when you want to exclude a result set from the final result set. In this example, the result of the second query is ignored.

The following statement combines results with the EXCEPT operator, which returns only unique rows returned by the first query but not by the second:

```
SELECT product_id FROM inventories
EXCEPT
SELECT product_id FROM order_items
ORDER BY product_id;
```

Sorting Query Results

Use the ORDER BY clause to order the rows selected by a query. Sorting by position is useful in the following cases:

- To order by a lengthy select list expression, you can specify its position in the ORDER BY clause rather than duplicate the entire expression.
- For compound queries containing set operators UNION, INTERSECT, MINUS, or UNION ALL, the ORDER BY clause must specify positions or aliases rather than explicit expressions. Also, the ORDER BY clause can appear only in the last component query. The ORDER BY clause orders all rows returned by the entire compound query.

The ordering method by which Oracle Database sorts character values for the ORDER BY clause, also known as the collation, is determined for each ORDER BY clause expression separately using the collation derivation rules.

If the determined collation of an expression is not the collation BINARY, then the character values are compared linguistically. In this case, they are first transformed to collation keys and then compared like RAW values. The collation keys are generated implicitly using the same method that the SQL function NLSSORT uses. Generated collation keys are subject to the same restrictions that are described in "NLSSORT". As a result of these restrictions, if the initialization parameter MAX_STRING_SIZE is set to STANDARD, two values may compare as linguistically equal

if they do not differ in the prefix that was used to produce the collation key, even if they differ in the rest of the value. If the parameter's value is `EXTENDED`, then the error "`ORA-12742: unable to create the collation key`" may be reported under certain circumstances. See the links below for further information on the restrictions.

> **See Also:**
>
> - Collation Derivation
> - Linguistic Sorting and Matching
> - Default Values for NLS Parameters in SQL Functions
> - NLSSORT

Joins

A **join** is a query that combines rows from two or more tables, views, or materialized views. Oracle Database performs a join whenever multiple tables appear in the `FROM` clause of the query. The select list of the query can select any columns from any of these tables. If any two of these tables have a column name in common, then you must qualify all references to these columns throughout the query with table names to avoid ambiguity.

Join Conditions

Most join queries contain at least one **join condition**, either in the `FROM` clause or in the `WHERE` clause. The join condition compares two columns, each from a different table. To execute a join, Oracle Database combines pairs of rows, each containing one row from each table, for which the join condition evaluates to `TRUE`. The columns in the join conditions need not also appear in the select list.

To execute a join of three or more tables, Oracle first joins two of the tables based on the join conditions comparing their columns and then joins the result to another table based on join conditions containing columns of the joined tables and the new table. Oracle continues this process until all tables are joined into the result. The optimizer determines the order in which Oracle joins tables based on the join conditions, indexes on the tables, and, any available statistics for the tables.

A `WHERE` clause that contains a join condition can also contain other conditions that refer to columns of only one table. These conditions can further restrict the rows returned by the join query.

> **Note:**
>
> You cannot specify LOB columns in the `WHERE` clause if the `WHERE` clause contains the join condition. The use of LOBs in `WHERE` clauses is also subject to other restrictions. See *Oracle Database SecureFiles and Large Objects Developer's Guide* for more information.

Equijoins

An **equijoin** is a join with a join condition containing an equality operator. An equijoin combines rows that have equivalent values for the specified columns. Depending on the internal algorithm the optimizer chooses to execute the join, the total size of the columns in the equijoin condition in a single table may be limited to the size of a data block minus some overhead. The size of a data block is specified by the initialization parameter `DB_BLOCK_SIZE`.

> **See Also:**
>
> "Using Join Queries: Examples"

Band Joins

A **band join** is a special type of nonequijoin in which key values in one data set must fall within the specified range (“band”) of the second data set. The same table can serve as both the first and second data sets.

> **See Also:**
>
> - *Database SQL Tuning Guide* for more information on band joins
> - USE_BAND Hint
> - NO_USE_BAND Hint

Self Joins

A **self join** is a join of a table to itself. This table appears twice in the `FROM` clause and is followed by table aliases that qualify column names in the join condition. To perform a self join, Oracle Database combines and returns rows of the table that satisfy the join condition.

> **See Also:**
>
> "Using Self Joins: Example"

Cartesian Products

If two tables in a join query have no join condition, then Oracle Database returns their **Cartesian product**. Oracle combines each row of one table with each row of the other. A Cartesian product always generates many rows and is rarely useful. For example, the Cartesian product of two tables, each with 100 rows, has 10,000 rows. Always include a join condition unless you specifically need a Cartesian product. If a query joins three or more tables and you do not specify a join condition for a specific pair, then the optimizer may choose a join order that avoids producing an intermediate Cartesian product.

Inner Joins

An **inner join** (sometimes called a **simple join**) is a join of two or more tables that returns only those rows that satisfy the join condition.

Outer Joins

An **outer join** extends the result of a simple join. An outer join returns all rows that satisfy the join condition and also returns some or all of those rows from one table for which no rows from the other satisfy the join condition.

- To write a query that performs an outer join of tables A and B and returns all rows from A (a **left outer join**), use the `LEFT [OUTER] JOIN` syntax in the `FROM` clause, or apply the outer join operator (+) to all columns of B in the join condition in the `WHERE` clause. For all rows in A that have no matching rows in B, Oracle Database returns null for any select list expressions containing columns of B.
- To write a query that performs an outer join of tables A and B and returns all rows from B (a **right outer join**), use the `RIGHT [OUTER] JOIN` syntax in the `FROM` clause, or apply the outer join operator (+) to all columns of A in the join condition in the `WHERE` clause. For all rows in B that have no matching rows in A, Oracle returns null for any select list expressions containing columns of A.
- To write a query that performs an outer join and returns all rows from A and B, extended with nulls if they do not satisfy the join condition (a **full outer join**), use the `FULL [OUTER] JOIN` syntax in the `FROM` clause.

You cannot compare a column with a subquery in the `WHERE` clause of any outer join, regardless which form you specify.

You can use outer joins to fill gaps in sparse data. Such a join is called a **partitioned outer join** and is formed using the *query_partition_clause* of the *join_clause* syntax. Sparse data is data that does not have rows for all possible values of a dimension such as time or department. For example, tables of sales data typically do not have rows for products that had no sales on a given date. Filling data gaps is useful in situations where data sparsity complicates analytic computation or where some data might be missed if the sparse data is queried directly.

> **See Also:**
>
> - *join_clause* for more information about using outer joins to fill gaps in sparse data
> - *Oracle Database Data Warehousing Guide* for a complete discussion of group outer joins and filling gaps in sparse data

Oracle recommends that you use the `FROM` clause `OUTER JOIN` syntax rather than the Oracle join operator. Outer join queries that use the Oracle join operator (+) are subject to the following rules and restrictions, which do not apply to the `FROM` clause `OUTER JOIN` syntax:

- You cannot specify the (+) operator in a query block that also contains `FROM` clause join syntax.

- The (+) operator can appear only in the `WHERE` clause or, in the context of left-correlation (when specifying the `TABLE` clause) in the `FROM` clause, and can be applied only to a column of a table or view.
- If A and B are joined by multiple join conditions, then you must use the (+) operator in all of these conditions. If you do not, then Oracle Database will return only the rows resulting from a simple join, but without a warning or error to advise you that you do not have the results of an outer join.
- The (+) operator does not produce an outer join if you specify one table in the outer query and the other table in an inner query.
- You cannot use the (+) operator to outer-join a table to itself, although self joins are valid. For example, the following statement is **not** valid:

```
-- The following statement is not valid:
SELECT employee_id, manager_id
   FROM employees
   WHERE employees.manager_id(+) = employees.employee_id;
```

  However, the following self join is valid:

```
SELECT e1.employee_id, e1.manager_id, e2.employee_id
   FROM employees e1, employees e2
   WHERE e1.manager_id(+) = e2.employee_id
   ORDER BY e1.employee_id, e1.manager_id, e2.employee_id;
```

- The (+) operator can be applied only to a column, not to an arbitrary expression. However, an arbitrary expression can contain one or more columns marked with the (+) operator.
- A `WHERE` condition containing the (+) operator cannot be combined with another condition using the `OR` logical operator.
- A `WHERE` condition cannot use the `IN` comparison condition to compare a column marked with the (+) operator with an expression.

If the `WHERE` clause contains a condition that compares a column from table B with a constant, then the (+) operator must be applied to the column so that Oracle returns the rows from table A for which it has generated nulls for this column. Otherwise Oracle returns only the results of a simple join.

In previous releases of Oracle Database, in a query that performed outer joins of more than two pairs of tables, a single table could be the null-generated table for only one other table. Beginning with Oracle Database 12*c*, a single table can be the null-generated table for multiple tables. For example, the following statement is allowed in Oracle Database 12*c*:

```
SELECT * FROM A, B, D
  WHERE A.c1 = B.c2(+) and D.c3 = B.c4(+);
```

In this example, `B`, the null-generated table, is outer-joined to two tables, `A` and `D`. Refer to SELECT for the syntax for an outer join.

Antijoins

An antijoin returns rows from the left side of the predicate for which there are no corresponding rows on the right side of the predicate. It returns rows that fail to match (`NOT IN`) the subquery on the right side.

> **See Also:**
>
> "Using Antijoins: Example"

Semijoins

A semijoin returns rows that match an `EXISTS` subquery without duplicating rows from the left side of the predicate when multiple rows on the right side satisfy the criteria of the subquery.

Semijoin and antijoin transformation cannot be done if the subquery is on an `OR` branch of the `WHERE` clause.

> **See Also:**
>
> "Using Semijoins: Example"

Using Subqueries

A **subquery** answers multiple-part questions. For example, to determine who works in Taylor's department, you can first use a subquery to determine the department in which Taylor works. You can then answer the original question with the parent `SELECT` statement. A subquery in the `FROM` clause of a `SELECT` statement is also called an **inline view**. you can nest any number of subqueries in an inline view. A subquery in the `WHERE` clause of a `SELECT` statement is also called a **nested subquery**. You can nest up to 255 levels of subqueries in a nested subquery.

A subquery can contain another subquery. Oracle Database imposes no limit on the number of subquery levels in the `FROM` clause of the top-level query. You can nest up to 255 levels of subqueries in the `WHERE` clause.

If columns in a subquery have the same name as columns in the containing statement, then you must prefix any reference to the column of the table from the containing statement with the table name or alias. To make your statements easier to read, always qualify the columns in a subquery with the name or alias of the table, view, or materialized view.

Oracle performs a **correlated subquery** when a nested subquery references a column from a table referred to a parent statement one or more levels above the subquery or nested subquery. The parent statement can be a `SELECT`, `UPDATE`, or `DELETE` statement in which the subquery is nested. A correlated subquery conceptually is evaluated once for each row processed by the parent statement. However, the optimizer may choose to rewrite the query as a join or use some other technique to formulate a query that is semantically equivalent. Oracle resolves unqualified columns in the subquery by looking in the tables named in the subquery and then in the tables named in the parent statement.

A correlated subquery answers a multiple-part question whose answer depends on the value in each row processed by the parent statement. For example, you can use a correlated subquery to determine which employees earn more than the average salaries for their departments. In this case, the correlated subquery specifically computes the average salary for each department.

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

Use subqueries for the following purposes:

- To define the set of rows to be inserted into the target table of an `INSERT` or `CREATE TABLE` statement
- To define the set of rows to be included in a view or materialized view in a `CREATE VIEW` or `CREATE MATERIALIZED VIEW` statement
- To define one or more values to be assigned to existing rows in an `UPDATE` statement
- To provide values for conditions in a `WHERE` clause, `HAVING` clause, or `START WITH` clause of `SELECT`, `UPDATE`, and `DELETE` statements
- To define a table to be operated on by a containing query

  You do this by placing the subquery in the `FROM` clause of the containing query as you would a table name. You may use subqueries in place of tables in this way as well in `INSERT`, `UPDATE`, and `DELETE` statements.

  Subqueries so used can employ correlation variables, both defined within the subquery itself and those defined in query blocks containing the subquery. Refer to *table_collection_expression* for more information.

  Scalar subqueries, which return a single column value from a single row, are a valid form of expression. You can use scalar subquery expressions in most of the places where *expr* is called for in syntax. Refer to "Scalar Subquery Expressions " for more information.

Unnesting of Nested Subqueries

The term *subquery* refers to a sub-query block that appears in the `WHERE` and `HAVING` clauses. A sub-query that appears in the `FROM` clause is called a *view* or derived table.

A `WHERE` clause subquery belongs to one of the following types: `SINGLE-ROW`, `EXISTS`, `NOT EXISTS`, `ANY`, or `ALL`. A single-row subquery must return at most one row, whereas the other types of subquery can return zero or more rows.

`ANY` and `ALL` subqueries are used with relational comparison operators: =, >,>=, <, <=, and <>.

In SQL, the set operator `IN` is used as a shorthand for =`ANY` and the set operator `NOT IN` is used as a shorthand for <>`ALL`.

Example: Correlated `EXISTS` Subquery

The subquery in the example is correlated, because the column *C.cust_id* comes from the table *customers*, that is not defined by the subquery.

```
SELECT C.cust_last_name, C.country_id
        FROM customers C
        WHERE EXISTS (SELECT 1
              FROM sales S
              WHERE S.quantity_sold > 1000 and
                    S.cust_id = C.cust_id);
```

Nested subqueries are those subqueries that appear in the `WHERE` and `HAVING` clauses of a parent statement like `SELECT`. When Oracle Database evaluates a statement with a nested subquery, it must evaluate the subquery portion multiple times and may overlook more efficient access paths or joins.

Subquery unnesting is an optimization that converts a subquery into a join in the outer query and allows the optimizer to consider subquery tables during access path, join method, and join order selection. Unnesting either merges the subquery into the body of the outer query block or turns it into an inline view.

When a subquery is unnested, it is merged into the statement that contains it, allowing the optimizer to consider them together when evaluating access paths and joins. The optimizer can unnest most subqueries, with some exceptions. Those exceptions include hierarchical subqueries and subqueries that contain a `ROWNUM` pseudocolumn, one of the set operators, a nested aggregate function, or a correlated reference to a query block that is not the immediate outer query block of the subquery.

Assuming no restrictions exist, the optimizer automatically unnests some (but not all) of the following nested subqueries:

- Uncorrelated `IN` subqueries
- `IN` and `EXISTS` correlated subqueries, as long as they do not contain aggregate functions or a `GROUP BY` clause

You can enable **extended subquery unnesting** by instructing the optimizer to unnest additional types of subqueries:

- You can unnest an uncorrelated `NOT IN` subquery by specifying the `HASH_AJ` or `MERGE_AJ` hint in the subquery.
- You can unnest other subqueries by specifying the `UNNEST` hint in the subquery.

> **See Also:**
>
> "Hints " for information on hints

Example: Uncorrelated `ANY` Subquery

```
SELECT C.cust_last_name, C.country_id
        FROM customers C
        WHERE C.cust_id =ANY (SELECT S.cust_id
                     FROM sales S
                     WHERE S.quantity_sold > 1000);
```

Example: `NOT EXISTS` Subquery

```
SELECT C.cust_last_name, C.country_id
FROM customers C
WHERE NOT EXISTS (SELECT 1
                  FROM sales S, products P
                  WHERE P.prod_id = S.prod_id and
                        P.prod_min_price > 90 and
                        S.cust_id = C.cust_id);
```

Selecting from the DUAL Table

`DUAL` is a table automatically created by Oracle Database along with the data dictionary. `DUAL` is in the schema of the user `SYS` but is accessible by the name `DUAL` to all users. It has one column, `DUMMY`, defined to be `VARCHAR2(1)`, and contains one row with a value `X`. Selecting from the `DUAL` table is useful for computing a constant expression with the `SELECT` statement. Because `DUAL` has only one row, the constant is returned only once. Alternatively, you can select a constant, pseudocolumn, or expression from any table, but the value will be returned as many times as there are rows in the table. Refer to "About SQL Functions " for many examples of selecting a constant value from `DUAL`.

Beginning with Oracle Database Release 23, it is now optional to select expressions using the `FROM DUAL` clause.

> **Note:**
>
> Beginning with Oracle Database 10*g* Release 1, logical I/O is not performed on the `DUAL` table when computing an expression that does not include the `DUMMY` column. This optimization is listed as `FAST DUAL` in the execution plan. If you `SELECT` the `DUMMY` column from `DUAL`, then this optimization does not take place and logical I/O occurs.

Distributed Queries

The Oracle distributed database management system architecture lets you access data in remote databases using Oracle Net and an Oracle Database server. You can identify a remote table, view, or materialized view by appending @*dblink* to the end of its name. The *dblink* must be a complete or partial name for a database link to the database containing the remote table, view, or materialized view.

> **See Also:**
>
> References to Objects in Remote Databases for more information on referring to database links

Restrictions on Distributed Queries

Distributed queries are currently subject to the restriction that all tables locked by a `FOR UPDATE` clause and all tables with `LONG` columns selected by the query must be located on the same database. In addition, Oracle Database currently does not support distributed queries that select user-defined types or object `REF` data types on remote tables.