## SELECT

SQL **SELECT** statement is used to fetch the data from a database table which returns data in the form of result table. These result tables are called result-sets.

Syntax:

The basic syntax of SELECT statement is as follows:

```
SELECT column1, column2, columnN FROM table_name;
```

Here, column1, column2...are the fields of a table whose values you want to fetch. If you want to fetch all the fields available in the field, then you can use the following syntax:

```
SELECT * FROM table_name;
```

Example:

Consider the CUSTOMERS table having the following records:

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

Following is an example, which would fetch ID, Name and Salary fields of the customers available in CUSTOMERS table:

```
SQL> SELECT ID, NAME, SALARY FROM CUSTOMERS;
```

This would produce the following result:

```
+----+----------+----------+
| ID | NAME     | SALARY   |
+----+----------+----------+
|  1 | Ramesh   |  2000.00 |
|  2 | Khilan   |  1500.00 |
|  3 | kaushik  |  2000.00 |
|  4 | Chaitali |  6500.00 |
|  5 | Hardik   |  8500.00 |
|  6 | Komal    |  4500.00 |
|  7 | Muffy    | 10000.00 |
+----+----------+----------+
```

If you want to fetch all the fields of CUSTOMERS table, then use the following query:

```
SQL> SELECT * FROM CUSTOMERS;
```

This would produce the following result:

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

The SQL **WHERE** clause is used to specify a condition while fetching the data from single table or joining with multiple tables.

If the given condition is satisfied then only it returns specific value from the table. You would use WHERE clause to filter the records and fetching only necessary records.

The WHERE clause is not only used in SELECT statement, but it is also used in UPDATE, DELETE statement, etc., which we would examine in subsequent chapters.

### Syntax:

The basic syntax of SELECT statement with WHERE clause is as follows:

```
SELECT column1, column2, columnN
FROM table_name
WHERE [condition]
```

You can specify a condition using comparison or logical operators like >, <, =, LIKE, NOT, etc. Below examples would make this concept clear.

Example:

Consider the CUSTOMERS table having the following records:

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

Following is an example which would fetch ID, Name and Salary fields from the CUSTOMERS table where salary is greater than 2000:

```
SQL> SELECT ID, NAME, SALARY
FROM CUSTOMERS
WHERE SALARY > 2000;
```

This would produce the following result:

```
+----+----------+----------+
| ID | NAME     | SALARY   |
+----+----------+----------+
|  4 | Chaitali |  6500.00 |
|  5 | Hardik   |  8500.00 |
|  6 | Komal    |  4500.00 |
|  7 | Muffy    | 10000.00 |
+----+----------+----------+
```

Following is an example, which would fetch ID, Name and Salary fields from the CUSTOMERS table for a customer with name **Hardik**. Here, it is important to note that all the strings should be given inside single quotes ('') where as numeric values should be given without any quote as in above example:

```
SQL> SELECT ID, NAME, SALARY
FROM CUSTOMERS
WHERE NAME = 'Hardik';
```

This would produce the following result:

```
+----+----------+----------+
| ID | NAME     | SALARY   |
+----+----------+----------+
|  5 | Hardik   |  8500.00 |
+----+----------+----------+
```

The SQL **AND** and **OR** operators are used to combine multiple conditions to narrow data in an SQL statement. These two operators are called conjunctive operators.

These operators provide a means to make multiple comparisons with different operators in the same SQL statement.

## The AND Operator:

The **AND** operator allows the existence of multiple conditions in an SQL statement's WHERE clause.

### Syntax:

The basic syntax of AND operator with WHERE clause is as follows:

```
SELECT column1, column2, columnN
FROM table_name
WHERE [condition1] AND [condition2]...AND [conditionN];
```

You can combine N number of conditions using AND operator. For an action to be taken by the SQL statement, whether it be a transaction or query, all conditions separated by the AND must be TRUE.

### Example:

Consider the CUSTOMERS table having the following records:

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

Following is an example, which would fetch ID, Name and Salary fields from the CUSTOMERS table where salary is greater than 2000 AND age is less tan 25 years:

```
SQL> SELECT ID, NAME, SALARY
FROM CUSTOMERS
WHERE SALARY > 2000 AND age < 25;
```

This would produce the following result:

```
+----+-------+----------+
| ID | NAME  | SALARY   |
+----+-------+----------+
|  6 | Komal |  4500.00 |
|  7 | Muffy | 10000.00 |
+----+-------+----------+
```

### The OR Operator:

The OR operator is used to combine multiple conditions in an SQL statement's WHERE clause.

### Syntax:

The basic syntax of OR operator with WHERE clause is as follows:

```
SELECT column1, column2, columnN
FROM table_name
WHERE [condition1] OR [condition2]...OR [conditionN]
```

You can combine N number of conditions using OR operator. For an action to be taken by the SQL statement, whether it be a transaction or query, only any ONE of the conditions separated by the OR must be TRUE.

### Example:

Consider the CUSTOMERS table having the following records:

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

Following is an example, which would fetch ID, Name and Salary fields from the CUSTOMERS table where salary is greater than 2000 OR age is less tan 25 years:

```
SQL> SELECT ID, NAME, SALARY
FROM CUSTOMERS
WHERE SALARY > 2000 OR age < 25;
```

This would produce the following result:

```
+----+----------+----------+
| ID | NAME     | SALARY   |
+----+----------+----------+
|  3 | kaushik  |  2000.00 |
|  4 | Chaitali |  6500.00 |
|  5 | Hardik   |  8500.00 |
|  6 | Komal    |  4500.00 |
|  7 | Muffy    | 10000.00 |
+----+----------+----------+
```

## JOINS

- JOINs can be used to combine tables
    - There are many types of JOIN
        - **CROSS JOIN**
        - **INNER JOIN**
        - **NATURAL JOIN**
        - **OUTER JOIN**
    - **OUTER JOIN**s are linked with **NULL**s - more later

**A CROSS JOIN B**

- returns all pairs of rows from A and B

**A NATURAL JOIN B**

- returns pairs of rows with common values for identically named columns and without duplicating columns

**A INNER JOIN B**

- returns pairs of rows satisfying a condition

Student

| ID | Name |
|---|---|
| 123 | John |
| 124 | Mary |
| 125 | Mark |
| 126 | Jane |

Enrolment

| ID | Code |
|---|---|
| 123 | DBS |
| 124 | PRG |
| 124 | DBS |
| 126 | PRG |

**SELECT * FROM  Student CROSS JOIN Enrolment**

| ID | Name | ID | Code |
|---|---|---|---|
| 123 | John | 123 | DBS |
| 124 | Mary | 123 | DBS |
| 125 | Mark | 123 | DBS |
| 126 | Jane | 123 | DBS |
| 123 | John | 124 | PRG |
| 124 | Mary | 124 | PRG |
| 125 | Mark | 124 | PRG |
| 126 | Jane | 124 | PRG |
| 123 | John | 124 | DBS |
| 124 | Mary | 124 | DBS |

NATURAL JOIN

Student

| ID | Name |
|---|---|

| | |
|---|---|
| 123 | John |
| 124 | Mary |
| 125 | Mark |
| 126 | Jane |

Enrolment

| ID | Code |
|---|---|
| 123 | DBS |
| 124 | PRG |
| 124 | DBS |
| 126 | PRG |

**SELECT * FROM  Student NATURAL JOIN Enrolment**

| ID | Name | Code |
|---|---|---|
| 123 | John | DBS |
| 124 | Mary | PRG |
| 124 | Mary | DBS |
| 126 | Jane | PRG |

CROSS and NATURAL JOIN

**SELECT * FROM  A CROSS JOIN B**

- is the same as

**SELECT * FROM A, B**

**SELECT * FROM A NATURAL JOIN B**

- is the same as

**SELECT A.col1,… A.coln, [and all other columns apart from B.col1,…B.coln] FROM A, B**

**WHERE A.col1 = B.col1  AND A.col2 = B.col2…AND A.coln = B.col.n**

**(this assumes that col1… coln in A and B have common names)**

INNER JOIN

- **INNER JOIN**s specify a condition which the pairs of rows satisfy

**SELECT * FROM A INNER JOIN B ON <condition>**

- Can also use

**SELECT * FROM A INNER JOIN B  USING (col1, col2,...)**

- Chooses rows where the given columns are equal

Student

| ID | Name |
|---|---|
| 123 | John |
| 124 | Mary |
| 125 | Mark |
| 126 | Jane |

Enrolment

| ID | Code |
|---|---|
| 123 | DBS |
| 124 | PRG |
| 124 | DBS |
| 126 | PRG |

**SELECT * FROM Student INNER JOIN Enrolment USING (ID)**

| ID | Name | ID | Code |
|---|---|---|---|
| 123 | John | 123 | DBS |
| 124 | Mary | 124 | PRG |
| 124 | Mary | 124 | DBS |
| 126 | Jane | 126 | PRG |

Buyer

| Name | Budget |
|---|---|
| Smith | 100,000 |
| Jones | 150,000 |
| Green | 80,000 |

Property

| Address | Price |
|---|---|
| 15 High St | 85,000 |
| 12 Queen St | 125,000 |
| 87 Oak Row | 175,000 |

**SELECT * FROM Buyer INNER JOIN Property ON Price <= Budget**

| Name | Budget | Address | Price |
|---|---|---|---|
| Smith | 100,000 | 15 High St | 85,000 |
| Jones | 150,000 | 15 High St | 85,000 |
| Jones | 150,000 | 12 Queen St | 125,000 |

**SELECT * FROM A INNER JOIN B ON <condition>**

- is the same as

**SELECT * FROM A, B WHERE <condition>**

**SELECT * FROM A INNER JOIN B USING(col1, col2,...)**

- is the same as

**SELECT * FROM A, B WHERE A.col1 = B.col1 AND A.col2 = B.col2 AND ...**

JOINs vs WHERE Clauses

- JOINs (so far) are not needed
  - You can have the same effect by selecting from multiple tables with an appropriate WHERE clause
  - So should you use JOINs or not?

- Yes, because
  - They often lead to concise queries
  - NATURAL JOINs are very common
- No, because
  - Support for JOINs varies a fair bit among SQL dialects

This Lecture in Exams

Track

| cID | Num | Title | Time | aID |
|---|---|---|---|---|
| 1 | | Violent | 239 | 1 |
| 2 | | Every Girl | 410 | 1 |
| 3 | | Breather | 217 | 1 |
| 4 | | Part of Me | 279 | 1 |
| 1 | | Star | 362 | 1 |
| 2 | | Teaboy | 417 | 2 |

CD

| cID | Title | Price |
|---|---|---|
| 1 | Mix | 9.99 |
| 2 | Compilation | 12.99 |

Artist

| aID | Name |
|---|---|
| 1 | Stellar |
| 2 | Cloudboy |

## OPERATIONS IN SQL PLUS

### What is an Operator in SQL?

An operator is a reserved word or a character used primarily in an SQL statement's WHERE clause to perform operation(s), such as comparisons and arithmetic operations.

Operators are used to specify conditions in an SQL statement and to serve as conjunctions for multiple conditions in a statement.

- Arithmetic operators
- Comparison operators
- Logical operators
- Operators used to negate conditions

### SQL Arithmetic Operators:

Assume variable a holds 10 and variable b holds 20, then:

Show Examples

| Operator | Description | Example |
|---|---|---|
| + | Addition - Adds values on either side of the operator | a + b will give 30 |
| - | Subtraction - Subtracts right hand operand from left hand operand | a - b will give -10 |
| * | Multiplication - Multiplies values on either side of the operator | a * b will give 200 |
| / | Division - Divides left hand operand by right hand operand | b / a will give 2 |
| % | Modulus - Divides left hand operand by right hand operand and returns remainder | b % a will give 0 |

SQL Comparison Operators:

Assume variable a holds 10 and variable b holds 20, then:

Show Examples

| Operator | Description | Example |
|---|---|---|
| = | Checks if the values of two operands are equal or not, if yes then condition becomes true. | (a = b) is not true. |
| != | Checks if the values of two operands are equal or not, if values are not equal then condition becomes true. | (a != b) is true. |
| <> | Checks if the values of two operands are equal or not, if values are not equal then condition becomes true. | (a <> b) is true. |
| > | Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true. | (a > b) is not true. |

| | | |
|---|---|---|
| < | Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true. | (a < b) is true. |
| >= | Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true. | (a >= b) is not true. |
| <= | Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true. | (a <= b) is true. |
| !< | Checks if the value of left operand is not less than the value of right operand, if yes then condition becomes true. | (a !< b) is false. |
| !> | Checks if the value of left operand is not greater than the value of right operand, if yes then condition becomes true. | (a !> b) is true. |

SQL Logical Operators:

Here is a list of all the logical operators available in SQL.

Show Examples

| **Operator** | **Description** |
|---|---|
| ALL | The ALL operator is used to compare a value to all values in another value set. |
| AND | The AND operator allows the existence of multiple conditions in an SQL statement's WHERE clause. |
| ANY | The ANY operator is used to compare a value to any applicable value in the list according to the condition. |
| BETWEEN | The BETWEEN operator is used to search for values that are within a set of values, given the minimum value and the maximum value. |
| EXISTS | The EXISTS operator is used to search for the presence of a row in a specified table that meets certain criteria. |
| IN | The IN operator is used to compare a value to a list of literal values that have been specified. |

| | |
|---|---|
| LIKE | The LIKE operator is used to compare a value to similar values using wildcard operators. |
| NOT | The NOT operator reverses the meaning of the logical operator with which it is used. Eg: NOT EXISTS, NOT BETWEEN, NOT IN, etc. **This is a negate operator.** |
| OR | The OR operator is used to combine multiple conditions in an SQL statement's WHERE clause. |
| IS NULL | The NULL operator is used to compare a value with a NULL value. |
| UNIQUE | The UNIQUE operator searches every row of a specified table for uniqueness (no duplicates). |

## SQL FUNCTIONS

SQL has many built-in functions for performing calculations on data.

SQL Aggregate Functions

SQL aggregate functions return a single value, calculated from values in a column.

Useful aggregate functions:

- AVG() - Returns the average value
- COUNT() - Returns the number of rows
- FIRST() - Returns the first value
- LAST() - Returns the last value
- MAX() - Returns the largest value
- MIN() - Returns the smallest value
- SUM() - Returns the sum

SQL Scalar functions

SQL scalar functions return a single value, based on the input value.

Useful scalar functions:

- UCASE() - Converts a field to upper case
- LCASE() - Converts a field to lower case
- MID() - Extract characters from a text field
- LEN() - Returns the length of a text field

- ROUND() - Rounds a numeric field to the number of decimals specified
- NOW() - Returns the current system date and time
- FORMAT() - Formats how a field is to be displayed

SQL numeric functions are used primarily for numeric manipulation and/or mathematical calculations. The following table details the numeric functions:

| Name | Description |
|---|---|
| **ABS()** | Returns the absolute value of numeric expression. |
| **ACOS()** | Returns the arccosine of numeric expression. Returns NULL if the value is not in the range -1 to 1. |
| **ASIN()** | Returns the arcsine of numeric expression. Returns NULL if value is not in the range -1 to 1 |
| **ATAN()** | Returns the arctangent of numeric expression. |
| **ATAN2()** | Returns the arctangent of the two variables passed to it. |
| **BIT_AND()** | Returns the bitwise AND all the bits in expression. |
| **BIT_COUNT()** | Returns the string representation of the binary value passed to it. |
| **BIT_OR()** | Returns the bitwise OR of all the bits in the passed expression. |
| **CEIL()** | Returns the smallest integer value that is not less than passed numeric expression |
| **CEILING()** | Returns the smallest integer value that is not less than passed numeric expression |
| **CONV()** | Convert numeric expression from one base to another. |
| **COS()** | Returns the cosine of passed numeric expression. The numeric expression should be expressed in radians. |
| **COT()** | Returns the cotangent of passed numeric expression. |
| **DEGREES()** | Returns numeric expression converted from radians to degrees. |
| **EXP()** | Returns the base of the natural logarithm (e) raised to the power of passed numeric expression. |
| **FLOOR()** | Returns the largest integer value that is not greater than passed numeric expression. |
| **FORMAT()** | Returns a numeric expression rounded to a number of decimal places. |
| **GREATEST()** | Returns the largest value of the input expressions. |

| | |
|---|---|
| **INTERVAL()** | Takes multiple expressions exp1, exp2 and exp3 so on.. and returns 0 if exp1 is less than exp2, returns 1 if exp1 is less than exp3 and so on. |
| **LEAST()** | Returns the minimum-valued input when given two or more. |
| **LOG()** | Returns the natural logarithm of the passed numeric expression. |
| **LOG10()** | Returns the base-10 logarithm of the passed numeric expression. |
| **MOD()** | Returns the remainder of one expression by diving by another expression. |
| **OCT()** | Returns the string representation of the octal value of the passed numeric expression. Returns NULL if passed value is NULL. |
| **PI()** | Returns the value of pi |
| **POW()** | Returns the value of one expression raised to the power of another expression |
| **POWER()** | Returns the value of one expression raised to the power of another expression |
| **RADIANS()** | Returns the value of passed expression converted from degrees to radians. |
| **ROUND()** | Returns numeric expression rounded to an integer. Can be used to round an expression to a number of decimal points |
| **SIN()** | Returns the sine of numeric expression given in radians. |
| **SQRT()** | Returns the non-negative square root of numeric expression. |
| **STD()** | Returns the standard deviation of the numeric expression. |
| **STDDEV()** | Returns the standard deviation of the numeric expression. |
| **TAN()** | Returns the tangent of numeric expression expressed in radians. |
| **TRUNCATE()** | Returns numeric exp1 truncated to exp2 decimal places. If exp2 is 0, then the result will have no decimal point. |

SQL string functions are used primarily for string manipulation. The following table details the important string functions:

| Name | Description |
|---|---|
| **ASCII()** | Returns numeric value of left-most character |

| | |
|---|---|
| **BIN()** | Returns a string representation of the argument |
| **BIT_LENGTH()** | Returns length of argument in bits |
| **CHAR_LENGTH()** | Returns number of characters in argument |
| **CHAR()** | Returns the character for each integer passed |
| **CHARACTER_LENGTH()** | A synonym for CHAR_LENGTH() |
| **CONCAT_WS()** | Returns concatenate with separator |
| **CONCAT()** | Returns concatenated string |
| **CONV()** | Converts numbers between different number bases |
| **ELT()** | Returns string at index number |
| **EXPORT_SET()** | Returns a string such that for every bit set in the value bits, you get an on string and for every unset bit, you get an off string |
| **FIELD()** | Returns the index (position) of the first argument in the subsequent arguments |
| **FIND_IN_SET()** | Returns the index position of the first argument within the second argument |
| **FORMAT()** | Returns a number formatted to specified number of decimal places |
| **HEX()** | Returns a string representation of a hex value |
| **INSERT()** | Inserts a substring at the specified position up to the specified number of characters |
| **INSTR()** | Returns the index of the first occurrence of substring |
| **LCASE()** | Synonym for LOWER() |
| **LEFT()** | Returns the leftmost number of characters as specified |
| **LENGTH()** | Returns the length of a string in bytes |
| **LOAD_FILE()** | Loads the named file |
| **LOCATE()** | Returns the position of the first occurrence of substring |
| **LOWER()** | Returns the argument in lowercase |
| **LPAD()** | Returns the string argument, left-padded with the specified string |

| | |
|---|---|
| **LTRIM()** | Removes leading spaces |
| **MAKE_SET()** | Returns a set of comma-separated strings that have the corresponding bit in bits set |
| **MID()** | Returns a substring starting from the specified position |
| **OCT()** | Returns a string representation of the octal argument |
| **OCTET_LENGTH()** | A synonym for LENGTH() |
| **ORD()** | If the leftmost character of the argument is a multi-byte character, returns the code for that character |
| **POSITION()** | A synonym for LOCATE() |
| **QUOTE()** | Escapes the argument for use in an SQL statement |
| **REGEXP** | Pattern matching using regular expressions |
| **REPEAT()** | Repeats a string the specified number of times |
| **REPLACE()** | Replaces occurrences of a specified string |
| **REVERSE()** | Reverses the characters in a string |
| **RIGHT()** | Returns the specified rightmost number of characters |
| **RPAD()** | Appends string the specified number of times |
| **RTRIM()** | Removes trailing spaces |
| **SOUNDEX()** | Returns a soundex string |
| **SOUNDS LIKE** | Compares sounds |
| **SPACE()** | Returns a string of the specified number of spaces |
| **STRCMP()** | Compares two strings |
| **SUBSTRING_INDEX()** | Returns a substring from a string before the specified number of occurrences of the delimiter |
| **SUBSTRING(), SUBSTR()** | Returns the substring as specified |
| **TRIM()** | Removes leading and trailing spaces |
| **UCASE()** | Synonym for UPPER() |
| **UNHEX()** | Converts each pair of hexadecimal digits to a character |
| **UPPER()** | Converts to uppercase |

## SET OPERATORS

Set operators are used to join the results of two (or more) SELECT statements.The SET operators available in Oracle 11g are UNION,UNION ALL,INTERSECT,and MINUS.

The UNION set operator returns the combined results of the two SELECT statements.Essentially,it removes duplicates from the results i.e. only one row will be listed for each duplicated result.To counter this behavior,use the UNION ALL set operator which retains the duplicates in the final result.INTERSECT lists only records that are common to both the SELECT queries; the MINUS set operator removes the second query's results from the output if they are also found in the first query's results. INTERSECT and MINUS set operations produce unduplicated results.

All the SET operators share the same degree of precedence among them.Instead,during query execution, Oracle starts evaluation from left to right or from top to bottom.If explicitly parentheses are used, then the order may differ as parentheses would be given priority over dangling operators.

Points to remember -

- Same number of columns must be selected by all participating SELECT statements.Column names used in the display are taken from the first query.
- Data types of the column list must be compatible/implicitly convertible by oracle. Oracle will not perform implicit type conversion if corresponding columns in the component queries belong to different data type groups.For example, if a column in the first component query is of data type DATE, and the corresponding column in the second component query is of data type CHAR,Oracle will not perform implicit conversion, but raise ORA-01790 error.
- Positional ordering must be used to sort the result set. Individual result set ordering is not allowed with Set operators. ORDER BY can appear once at the end of the query. For example,
- UNION and INTERSECT operators are commutative, i.e. the order of queries is not important; it doesn't change the final result.
- Performance wise, UNION ALL shows better performance as compared to UNION because resources are not wasted in filtering duplicates and sorting the result set.
- Set operators can be the part of sub queries.
- Set operators can't be used in SELECT statements containing TABLE collection expressions.
- The LONG, BLOB, CLOB, BFILE, VARRAY,or nested table are not permitted for use in Set operators.For update clause is not allowed with the set operators.

UNION

When multiple SELECT queries are joined using UNION operator, Oracle displays the combined result from all the compounded SELECT queries,after removing all duplicates and in sorted order (ascending by default), without ignoring the NULL values.

Consider the below five queries joined using UNION operator.The final combined result set contains value from all the SQLs. Note the duplication removal and sorting of data.

```
SELECT 1 NUM FROM DUAL

UNION

SELECT 5 FROM DUAL

UNION

SELECT 3 FROM DUAL

UNION

SELECT 6 FROM DUAL

UNION

SELECT 3 FROM DUAL;


NUM

-------

1

3

5

6
```

To be noted, the columns selected in the SELECT queries must be of compatible data type. Oracle throws an error message when the rule is violated.

```
SELECT TO_DATE('12-OCT-03') FROM DUAL

UNION

SELECT '13-OCT-03' FROM DUAL;


SELECT TO_DATE('12-OCT-03') FROM DUAL

      *

ERROR at line 1:

ORA-01790: expression must have same datatype as corresponding expression
```

UNION ALL

UNION and UNION ALL are similar in their functioning with a slight difference. But UNION ALL gives the result set without removing duplication and sorting the data. For example,in above query UNION is replaced by UNION ALL to see the effect.

Consider the query demonstrated in UNION section. Note the difference in the output which is generated without sorting and deduplication.

```
SELECT 1 NUM FROM DUAL
UNION ALL
SELECT 5 FROM DUAL
UNION ALL
SELECT 3 FROM DUAL
UNION ALL
SELECT 6 FROM DUAL
UNION ALL
SELECT 3 FROM DUAL;

NUM
-------
1
5
3
6
3
```

## INTERSECT

Using INTERSECT operator, Oracle displays the common rows from both the SELECT statements, with no duplicates and data arranged in sorted order (ascending by default).

For example,the below SELECT query retrieves the salary which are common in department 10 and 20.As per ISO SQL Standards, INTERSECT is above others in precedence of evaluation of set operators but this is not still incorporated by Oracle.

```
SELECT SALARY
FROM employees
WHERE DEPARTMENT_ID = 10
INTRESECT
SELECT SALARY
FROM employees
```

```
WHERE DEPARTMENT_ID = 20

SALARY
---------
1500
1200
2000
```

## MINUS

Minus operator displays the rows which are present in the first query but absent in the second query, with no duplicates and data arranged in ascending order by default.

```
SELECT JOB_ID
FROM employees
WHERE DEPARTMENT_ID = 10
MINUS
SELECT JOB_ID
FROM employees
WHERE DEPARTMENT_ID = 20;

JOB_ID
-------------
HR
FIN
ADMIN
```

## Matching the SELECT statement

There may be the scenarios where the compound SELECT statements may have different count and data type of selected columns. Therefore, to match the column list explicitly, NULL columns are inserted at the missing positions so as match the count and data type of selected columns in each SELECT statement. For number columns, zero can also be substituted to match the type of the columns selected in the query.

In the below query, the data type of employee name (varchar2) and location id (number) do not match. Therefore, execution of the below query would raise error due to compatibility issue.

```
SELECT DEPARTMENT_ID "Dept", first_name "Employee"
FROM employees
UNION
SELECT DEPARTMENT_ID, LOCATION_ID
```

```
FROM departments;


ERROR at line 1:
ORA-01790: expression must have same datatype as corresponding expression
```

Explicitly, columns can be matched by substituting NULL for location id and Employee name.

```
SELECT DEPARTMENT_ID "Dept", first_name "Employee", NULL "Location"
FROM employees
UNION
SELECT DEPARTMENT_ID, NULL "Employee", LOCATION_ID
FROM departments;
```

### Using ORDER BY clause in SET operations

The ORDER BY clause can appear only once at the end of the query containing compound SELECT statements.It implies that individual SELECT statements cannot have ORDER BY clause. Additionally, the sorting can be based on the columns which appear in the first SELECT query only. For this reason, it is recommended to sort the compound query using column positions.

The compund query below unifies the results from two departments and sorts by the SALARY column.

```
SELECT employee_id, first_name, salary
FROM employees
WHERE department_id=10
UNION
SELECT employee_id, first_name, salary
FROM employees
WHERE department_id=20
ORDER BY 3;
```

## SUBQU SUBQUERIES

- A *subquery* is a query within a query.
- Subqueries enable you to write queries that select data rows for criteria that are actually developed while the query is executing at *run time*.

**Example:**

SELECT emp_last_name "Last Name", emp_first_name "First Name", emp_salary "Salary" FROM employee WHERE emp_salary = (SELECT MIN(emp_salary) FROM employee);

```
Last Name    First Name   Salary

-------------- -------------- --------

Markis        Marcia        $25,000

Amin          Hyder         $25,000

Prescott      Sherri        $25,000
```

**SUBQUERY TYPES**

- There are three basic types of subqueries. We will study each of these in the remainder of this chapter.

1. Subqueries that operate on lists by use of the IN operator or with a comparison operator modified by the ANY or ALL optional keywords. These subqueries can return a group of values, but the values must be from a single column of a table.
2. Subqueries that use an unmodified comparison operator (=, <, >, <>) – these subqueries must return only a single, *scalar* value.
3. Subqueries that use the EXISTS operator to test the *existence* of data rows satisfying specified criteria.

**SUBQUERY – General Rules**

A subquery SELECT statement is very similar to the SELECT statement used to begin a regular or outer query. The complete syntax of a subquery is shown below.

( SELECT [DISTINCT] subquery_select_argument FROM {table_name | view_name} {table_name | view_name} ... [WHERE search_conditions] [GROUP BY aggregate_expression [, aggregate_expression] ...] [HAVING search_conditions] )

**Rules Cont'd**

- The SELECT clause of a subquery must contain only one expression, only one aggregate function, or only one column name.
- The value(s) returned by a subquery must be *join-compatible* with the WHERE clause of the outer query.

Example

SELECT emp_last_name "Last Name", emp_first_name "First Name" FROM employee WHERE emp_ssn IN (SELECT dep_emp_ssn FROM dependent);

```
Last Name First Name

------------- ---------------

Bock       Douglas

Zhu        Waiman

Joyner      Suzanne
```

**Rules Cont'd**

- In addition to concerns about the domain of values returned from a subquery, the data type of the returned column value(s) must be *join-compatible*.
- Join-compatible data types are data types that the Oracle Server will convert automatically when matching data in criteria conditions.
- The Oracle Server will automatically convert among any of the following ANSI numeric data types when making comparisons of numeric values because they all map into the Oracle NUMBER data type.
    - int (integer)
    - smallint (small integer)
    - decimal
    - float
- Oracle does not make comparisons based on column names.
- Columns from two tables that are being compared may have different names as long as they have a shared domain and the same data type or convertible data types.

  There are additional restrictions for subqueries.

- The DISTINCT keyword cannot be used in subqueries that include a GROUP BY clause.
- Subqueries cannot manipulate their results internally. This means that a subquery cannot include the ORDER BY clause, the COMPUTE clause, or the INTO keyword.

**SUBQUERIES AND THE IN Operator**

- Subqueries that are introduced with the keyword IN take the general form:
    - WHERE expression [NOT] IN (subquery)
- The only difference in the use of the IN operator with subqueries is that the list does not consist of *hard-coded* values.

Example

```
SELECT emp_last_name "Last Name", emp_first_name "First Name" FROM employee WHERE emp_ssn
IN (SELECT dep_emp_ssn FROM dependent WHERE dep_gender = 'M');

Last Name   First Name

--------------- ---------------

Bock         Douglas

Zhu          Waiman

Joyner        Suzanne
```

- Conceptually, this statement is evaluated in two steps.
- First, the inner query returns the identification numbers of those employees that have male dependents.

```
SELECT dep_emp_ssn FROM dependent WHERE dep_gender = 'M';

DEP_EMP_S
```

```
---------

999444444

999555555

999111111
```

- Next, these social security number values are substituted into the outer query as the listing that is the object of the IN operator. So, from a conceptual perspective, the outer query now looks like the following.

```
SELECT emp_last_name "Last Name",  emp_first_name "First Name" FROM employee WHERE emp_ssn
IN (999444444, 999555555, 999111111);

Last Name    First Name

-------------- ---------------

Joyner        Suzanne

Zhu           Waiman

Bock          Douglas
```

**The NOT IN Operator**

- Like the IN operator, the NOT IN operator can take the result of a subquery as the operator object.

```
SELECT emp_last_name "Last Name", emp_first_name "First Name" FROM employee WHERE emp_ssn
NOT IN (SELECT dep_emp_ssn FROM dependent);

Last Name    First Name

-------------- ---------------

Bordoloi      Bijoy

Markis        Marcia

Amin          Hyder
```

*more rows are displayed . . .*

- The subquery shown above produces an intermediate result table containing the social security numbers of employees who have dependents in the dependent table.
- Conceptually, the outer query compares each row of the employee table against the result table. If the employee social security number is NOT found in the result table produced by the inner query, then it is included in the final result table.

**MULTIPLE LEVELS OF NESTING**

- Subqueries may themselves contain subqueries.
- When the WHERE clause of a subquery has as its object another subquery, these are termed *nested subqueries*.
- Oracle places no practical limit on the number of queries that can be nested in a WHERE clause.
- Consider the problem of producing a listing of employees that worked more than 10 hours on the project named *Order Entry*.

Example

```
SELECT emp_last_name "Last Name", emp_first_name "First Name" FROM employee WHERE emp_ssn
IN (SELECT work_emp_ssn FROM assignment WHERE work_hours > 10 AND work_pro_number IN
(SELECT pro_number FROM project WHERE pro_name = 'Order Entry') );
```

```
Last Name    First Name

--------------- ---------------

Bock           Douglas

Prescott       Sherri
```

**Understanding SUBQUERIES**

- In order to understand how this query executes, we begin our examination with the lowest subquery.
- We will execute it independently of the outer queries.

```
SELECT pro_number FROM project WHERE pro_name = 'Order Entry';
```

```
PRO_NUMBER

----------

      1
```

- Now, let's substitute the project number into the IN operator list for the intermediate subquery and execute it.
- The intermediate result table lists two employee social security numbers for employees that worked more than 10 hours on project #1.

```
SELECT work_emp_ssn FROM assignment WHERE work_hours > 10 AND work_pro_number IN (1);
```

```
WORK_EMP_SSN

-----------------------

999111111

999888888
```

- Finally, we will substitute these two social security numbers into the IN operator listing for the outer query in place of the subquery.

```
SELECT emp_last_name "Last Name", emp_first_name "First Name" FROM employee WHERE emp_ssn
IN (999111111, 999888888);

Last Name    First Name

--------------- ---------------

Bock           Douglas

Prescott       Sherri
```

**SUBQUERIES AND COMPARISON OPERATORS**

- The general form of the WHERE clause with a comparison operator is similar to that used thus far in the text.
- Note that the subquery is again enclosed by parentheses.

```
WHERE <expression> <comparison_operator> (subquery)
```

- The most important point to remember when using a subquery with a comparison operator is that the subquery can only return a single or *scalar* value.
- This is also termed a *scalar subquery* because a single column of a single row is returned by the subquery.
- If a subquery returns more than one value, the Oracle Server will generate the " ORA-01427: *single-row subquery returns more than one row "* error message, and the query will fail to execute.
- Let's examine a subquery that will not execute because it violates the "single value" rule.
- The query shown below returns multiple values for the *emp_salary* column.

```
SELECT emp_salary FROM employee WHERE emp_salary > 40000; EMP_SALARY

-------------------

          55000

          43000

          43000
```

- If  we substitute this query as a subquery in another SELECT statement, then that SELECT statement will fail.
- This is demonstrated in the next SELECT statement.  Here the SQL code will fail because the subquery uses the greater than (>) comparison operator and the subquery returns multiple values.

```
SELECT emp_ssn FROM employee WHERE emp_salary > (SELECT emp_salary FROM employee
WHERE emp_salary > 40000);

ERROR at line 4:

ORA-01427: single-row subquery returns more than one row
```

**Aggregate Functions and Comparison Operators**

- The aggregate functions (AVG, SUM, MAX, MIN, and COUNT) always return a *scalar* result table.
- Thus, a subquery with an aggregate function as the object of a comparison operator will always execute provided you have formulated the query properly.

```
SELECT emp_last_name "Last Name", emp_first_name "First Name", emp_salary "Salary" FROM
employee WHERE emp_salary > (SELECT AVG(emp_salary) FROM employee);
```

```
Last Name    First Name   Salary

--------------- --------------- ----------

Bordoloi      Bijoy         $55,000

Joyner        Suzanne       $43,000

Zhu           Waiman        $43,000

Joshi         Dinesh        $38,000
```

**<u>Comparison Operators Modified with the ALL or ANY Keywords</u>**

- The ALL and ANY keywords can modify a comparison operator to allow an outer query to accept multiple values from a subquery.
- The general form of the WHERE clause for this type of query is shown here.

  WHERE <expression> <comparison_operator> [ALL | ANY] (subquery)

- Subqueries that use these keywords may also include GROUP BY and HAVING clauses.

***<u>The ALL Keyword</u>***

- The ALL keyword modifies the greater than comparison operator to mean greater than <u>all</u> values.

```
SELECT emp_last_name "Last Name",   emp_first_name "First Name",   emp_salary "Salary" FROM
employee WHERE emp_salary > ALL (SELECT emp_salary FROM employee WHERE emp_dpt_number
= 7);
```

```
Last Name    First Name   Salary

--------------- --------------- --------

Bordoloi      Bijoy        $55,000
```

- The ANY keyword is not as restrictive as the ALL keyword.
- When used with the greater than comparison operator, "> ANY" means greater than <u>some</u> value.

<u>Example</u>

```
SELECT emp_last_name "Last Name", emp_first_name "First Name", emp_salary "Salary" FROM
employee WHERE emp_salary > ANY (SELECT emp_salary FROM employee WHERE emp_salary >
30000);
```

```
Last Name    First Name   Salary

--------------- --------------- --------

Bordoloi      Bijoy          $55,000

Joyner        Suzanne        $43,000

Zhu           Waiman         $43,000
```

***An "= ANY" (Equal Any) Example***

- The "= ANY" operator is exactly equivalent to the IN operator.
- For example, to find the names of employees that have male dependents, you can use either IN or "= ANY" – both of the queries shown below will produce an identical result table.

```
SELECT emp_last_name "Last Name", emp_first_name "First Name" FROM employee WHERE emp_ssn
IN (SELECT dep_emp_ssn FROM dependent WHERE dep_gender = 'M');

SELECT emp_last_name "Last Name", emp_first_name "First Name" FROM employee WHERE emp_ssn
= ANY (SELECT dep_emp_ssn FROM dependent WHERE dep_gender = 'M');
```

OUTPUT

```
Last Name   First Name

--------------- ---------------

Bock          Douglas

Zhu           Waiman

Joyner        Suzanne
```

***A "!= ANY" (Not Equal Any) Example***

- The "= ANY" is identical to the IN operator.
- However, the "!= ANY" (not equal any) is not equivalent to the NOT IN operator.
- If a subquery of employee salaries produces an intermediate result table with the salaries $38,000, $43,000, and $55,000, then the WHERE clause shown here means "NOT $38,000" AND "NOT $43,000" AND "NOT $55,000".

```
WHERE NOT IN (38000, 43000, 55000);
```

- However, the "!= ANY" comparison operator and keyword combination shown in this next WHERE clause means "NOT $38,000" OR "NOT $43,000" OR "NOT $55,000".

***CORRELATED SUBQUERIES***

- A *correlated subquery* is one where the inner query depends on values provided by the outer query.
- This means the inner query is executed repeatedly, once for each row that might be selected by the outer query.

```
SELECT emp_last_name "Last Name", emp_first_name "First Name", emp_dpt_number "Dept",
emp_salary "Salary" FROM employee e1 WHERE emp_salary = (SELECT MAX(emp_salary) FROM
employee WHERE emp_dpt_number = e1.emp_dpt_number);
```

Output

```
Last Name  FirstName  Dept  Salary

---------- ---------- ----- --------

Bordoloi   Bijoy        1  $55,000

Joyner     Suzanne      3  $43,000

Zhu        Waiman       7  $43,000
```

- The subquery in this SELECT statement cannot be resolved independently of the main query.
- Notice that the outer query specifies that rows are selected from the *employee* table with an alias name of *e1*.
- The inner query compares the employee department number column (*emp_dpt_number*) of the *employee* table to the same column for the alias table name *e1*.
- The value of *e1.emp_dpt_number* is treated like a variable – it changes as the Oracle server examines each row of the employee table.
- The subquery's results are correlated with each individual row of the main query – thus, the term *correlated subquery*.

***Subqueries and the ORDER BY Clause***

- The SELECT statement shown below adds the ORDER BY clause to specify sorting by first name within last name.
- Note that the ORDER BY clause is placed after the WHERE clause, and that this includes the subquery as part of the WHERE clause.

```
SELECT emp_last_name "Last Name", emp_first_name "First Name" FROM employee WHERE EXISTS
(SELECT * FROM dependent WHERE emp_ssn = dep_emp_ssn) ORDER BY emp_last_name,
emp_first_name;
```

Output:

```
Last Name  First Name

---------- ---------------

Bock       Douglas

Joyner     Suzanne

Zhu        Waiman
```

**LOCK TABLE**

**Lock**:

A lock is a mechanism to control concurrent access to a data item

Lock Manager: Managing locks on data items.

**Types of Locks**

1. **Binary locks**
2. **Shared/Exclusive locks**

**Binary Locks**

A binary lock can have two states

Locked (1)

Unlocked (0)

**Rules to be followed by Transactions**

- Transaction must lock data item before read_item or write_item operations
- Transaction must unlock data item after all read(x)and write(x) operation are completed in T
- A transaction T will not issue a lock_item(x) operation if it already holds the lock on item(x)
- A transaction T will not issue an unlock_item(x) operation unless it already holds the lock on item(x)

The following code performs the lock operation:

```
B:  if LOCK (X) = 0 (*item is unlocked*)
    then LOCK (X) ← 1 (*lock the item*)
    else begin
            wait (until lock (X) = 0) and
            the lock manager wakes up the transaction);
    goto B
    end;
```

The following code performs the unlock operation:

```
LOCK (X) ← 0 (*unlock the item*)
if any transactions are waiting then
        wake up one of the waiting the transactions;
```

Managing User privileges-