

### UNIT-III

#### QUERIES, CONSTRAINTS, TRIGGERS

**Introduction to SQL:** SQL stands for Structured Query Language. It is the most widely used commercial relational database language. There are numerous versions of SQL. The original SQL version was developed at IBM's San Jose Research Laboratory, California in 1970. This was originally called as Sequel Language of system R project in the early 1970's. The sequel was renamed as SQL during the years 1974 to 1979. The SQL is a combination of relational algebra and relational calculus. This was a run under the DOS operating system in 1981. The SQL language was announced as the MVS version, called DB2 in 1983.

In 1986, the ANSI/ISO published a SQL standard, called SQL-86. IBM published its own corporate SQL standard, called the System Application Architecture database interface (SAA-SQL) in 1987. An extended standard of SQL was published in 1989 by ANSI/ISO. The Next versions of ANSI/ISO are SQL-92, SQL-94, SQL-96, and SQL-99. Some Other Organizations developed Relational Data Base Software's, called INGRES, SYBASE etc.

In the DBMS, SQL can create the tables, translate user requests, maintain the data dictionary, maintain the system catalog, update and maintain the tables, establish security and carry out backup and recovery procedures.

#### The SQL language has several parts

**The Data Manipulation Language (DML):** This subset of SQL allows users to pose queries and to insert, delete, and modify rows. Queries are the main focus of this chapter. We covered DML commands to insert, delete, and modify rows.

**The Data Definition Language (DDL):** This subset of SQL supports the creation, deletion, and modification of definitions for tables and views. Integrity constraints can be defined on tables, either when the table is created or later. Although the standard does not discuss indexes, commercial implementations also provide commands for creating and deleting indexes.

**Triggers and Advanced Integrity Constraints:** The new SQL: 1999 standard includes support for triggers, which are actions executed by the DBMS whenever changes to the database meet conditions specified in the trigger

**Embedded and Dynamic SQL:** Embedded SQL features allow SQL code to be called from a host language such as C or COBOL. Dynamic SQL features allow a query to be constructed (and executed) at run-time.

**Client-Server Execution and Remote Database Access:** These commands control how a client application program can connect to an SQL database server, or access data from a database over a network.

**Transaction Management:** Various commands allow a user to explicitly control aspects of how a transaction is to be executed.

**Security:** SQL provides mechanisms to control users' access to data objects such as tables and views.

**Advanced features:** The SQL: 1999 standard includes object-oriented features, recursive queries, decision support queries, and also addresses emerging areas such as data mining, spatial data, and text and XML data management.

#### Role of SQL in Database Architecture :

A SQL based Relational database application is used to interface with a user to create the tables in database and maintain the data using different queries such as Enter the data, update the data, retrieve the data and carry out backup and recovery procedures.

A relational DBMS (RDBMS) is a data management system that manages data as a collection of tables in which all data relationships are represented by common values in related tables.

**The SQL standards:** The SQL standards are used

1. To specify the syntax and semantics of SQL data definition and data manipulation language commands.

2. To define the data structures and basic operations for designing, accessing, maintaining, controlling and protecting an SQL database.
3. To provide portability of database definitions (Tables) and application modules between different DBMS.
4. To specify both Level 1 and Level 2 standards. I.e., simple and complex queries.
5. To provides an initial standards for handling integrity, transaction management, user-defined functions, join operators, set operators, etc.

**SQL Benefits or Advantages :** The benefits of relational language include the following. They are

1. Reduced training costs: This reduces the training costs in organization.
2. Productivity: The user can learn SQL language thoroughly and become proficient. The programmers also write the programs and quickly maintain existing programs in SQL.
3. Application Portability: Applications can be easily moved from machine to machine and can develop the standards.
4. Application longevity: The old applications developed in SQL old versions can be developed using the advanced SQL do not taking a long time.
5. Reduced dependence on a single vendor: The learners can get assistance, training and educational services from the vendors. Because, the SQL is easy language. So user does not depend on single person. [i.e., more vendors are available to give the assistance].
6. Cross-system communication: Different DBMSs and application programs can more easily communicate and co-operate in managing data and processing user programs.

### **THE FORM OF A BASIC SQL QUERY**

The syntax of a simple SQL query and explains its meaning through a conceptual Evaluation strategy. A conceptual evaluation strategy is a way to evaluate the query that is intended to be easy to understand rather than efficient. A DBMS would typically execute a query in a different and more efficient way.

The basic form of an SQL query is as follows:

**SELECT [DISTINCT] select-list**  
**FROM from-list**  
**WHERE qualification**

Every query must have a SELECT clause, which specifies columns to be retained in the result, and a FROM clause, which specifies a cross-product of tables. The optional WHERE clause specifies selection conditions on the tables mentioned in the FROM clause.

Such a query intuitively corresponds to a relational algebra expression involving selections, projections, and cross-products. The close relationship between SQL and relational algebra is the basis for query optimization in a relational DBMS, as we will see in Chapters 12 and 15. Indeed, execution plans for SQL queries are represented using a variation of relational algebra expressions

we will present a number of sample queries using the following table definitions:

Sailors(sid: integer, sname: string, rating: integer, age: real)

Boats(bid: integer, bname: string, color: string)

Reserves (sid: integer, bid: integer, day: date)

#### **Create the sailors Table:**

```
CREATE TABLE sailors ( sid number(3),sname varchar2(12),rating number(2),age number(3,1),
CONSTRAINT PK_sailors PRIMARY KEY (sid) );
```

#### **Create the boats Table:**

```
CREATE TABLE boats(bid number(3) primary key,bname varchar2(12),color varchar2(10);
```

#### **Create the reserves Table:**

```
CREATE TABLE reserves ( sid number(3),bid number(3),day date,
CONSTRAINT PK_reserves PRIMARY KEY (sid, bid, day),
```

FOREIGN KEY (sid) REFERENCES sailors(sid),  
FOREIGN KEY (bid) REFERENCES boats(bid) );

Sailors

Sid	Sname	Rating	Age
22	Dustin	7	45
29	Brutus	1	33
31	Lubber	8	55.5
32	Andy	8	25.5
58	Rusty	10	35
64	Horatio	7	35
71	Zorba	10	16
74	Horatio	9	40
85	Art	3	25.5
95	Bob	3	63.5

Boats

bid	bname	color
101	Interlake	blue
102	Interlake	red
103	Clipper	green
104	Marine	red

Reserves

sid	bid	day
22	101	1998-10-10
22	102	1998-10-10
22	103	1998-10-8
22	104	1998-10-7
31	102	1998-11-10
31	103	1998-11-6
31	104	1998-11-12
64	101	1998-9-5
64	102	1998-9-8
74	103	1998-9-8

EX: Find all sailors with a rating above 7.

SQL>SELECT S.sid, S.sname, S.rating, S.age FROM Sailors AS S WHERE S.rating > 7

This query uses the optional keyword AS to introduce a range variable. Incidentally, when we want to retrieve all columns, as in this query, SQL provides convenient shorthand: we can simply write SELECT \*. This notation is useful for interactive querying, but it is poor style for queries that are intended to be reused and maintained because the schema of the result is not clear from the query itself; we have to refer to the schema of the underlying Sailors table.

### Now we consider the syntax of a basic SQL query in more detail.

- The from-list in the FROM clause is a list of table names. A table name can be followed by a range variable; a range variable is particularly useful when the same table name appears more than once in the from-list.
- The select-list is a list of (expressions involving) column names of tables named in the from-list. Column names can be prefixed by a range variable.
- The qualification in the WHERE clause is a Boolean combination (i.e., an expression using the logical connectives AND, OR, and NOT) of conditions of the form expression op expression, where op is one of the comparison operators {<, <=, =, <>, >=, >}.<sup>2</sup> An expression is a column name, a constant, or an (arithmetic or string) expression.
- The DISTINCT keyword is optional. It indicates that the table computed as an answer to this query should not contain duplicates, that is, two copies of the same row. The default is that duplicates are not eliminated.

The syntax of a basic SQL query, they do not tell us the meaning of a query. The answer to a query is itself a relation which is a multi set of rows in SQL considering the following conceptual evaluation strategy:

1. Compute the cross-product of the tables in the from-list.
2. Delete rows in the cross-product that fail the qualification conditions.
3. Delete all columns that do not appear in the select-list.
4. If DISTINCT is specified, eliminate duplicate rows.

### UNION, INTERSECT AND EXCEPT

SQL provides three set-manipulation constructs that extend the basic query it is natural to consider the use of operations such as union, intersection, and difference. SQL supports these operations under the names UNION, INTERSECT, and EXCEPT.

SQL also provides other set operations: IN (to check if an element is in a given set), op ANY, op ALL (to compare a value with the elements in a given set, using comparison operator op), and EXISTS (to

check if a set is empty). IN and EXISTS can be prefixed by NOT, with the obvious modification to their meaning.

Ex: Find the names of sailors who have reserved a red or a green boat

```
SQL>SELECT S.sname FROM Sailors S, Reserves R, Boats B WHERE S.sid = R.sid AND R.bid = B.bid AND B.color = 'red' UNION SELECT S1.sname FROM Sailors S1, Boats B1, Reserves R1 WHERE S1.sid = R1.sid AND R1.bid = B1.bid AND B1.color = 'green';
```

Ex: Find the names of sailor's who have reserved both a red and a green boats.

```
SQL>SELECT S.sname FROM Sailors S, Reserves R, Boats B WHERE S.sid = R.sid AND R.bid = B.bid AND B.color = 'red' INTERSECT SELECT S1.sname FROM Sailors S1, Boats B1, Reserves R1 WHERE S1.sid = R1.sid AND R1.bid = B1.bid AND B1.color = 'green';
```

Ex: Find the sids of all sailor's who have reserved red boats but not green boats.

```
SQL>SELECT S.sid FROM Sailors S, Reserves R, Boats B WHERE S.sid = R.sid AND R.bid = B.bid AND B.color = 'red' EXCEPT SELECT S1.sid FROM Sailors S1, Boats B1, Reserves R1 WHERE S1.sid = R1.sid AND R1.bid = B1.bid AND B1.color = 'green';
```

Sailors 22, 64, and 31 have reserved red boats. Sailors 22, 74, and 31 have reserved green boats. Hence, the answer contains just the sid 64.

### **NESTED QUERIES:**

A Sub query or Inner query or a Nested query is a query within another SQL query and embedded within the WHERE clause.

A sub query is used to return data that will be used in the main query as a condition to further restrict the data to be retrieved.

Sub queries can be used with the SELECT, INSERT, UPDATE, and DELETE statements along with the operators like =, <, >, >=, <=, IN, BETWEEN, etc.

There are a few rules that sub queries must follow

- Sub queries must be enclosed within parentheses.
- A sub query can have only one column in the SELECT clause, unless multiple columns are in the main query for the sub query to compare its selected columns.
- An ORDER BY command cannot be used in a sub query, although the main query can use an ORDER BY. The GROUP BY command can be used to perform the same function as the ORDER BY in a sub query.
- Sub queries that return more than one row can only be used with multiple value operators such as the IN operator.
- A sub query cannot be immediately enclosed in a set function.
- The BETWEEN operator cannot be used with a sub query. However, the BETWEEN operator can be used within the sub query.

#### **Syntax of nested query:**

```
SELECT column_name [,column_name]
FROM table1 [,table2]
WHERE column_name OPERATOR
      (SELECT column_name [,column_name]
      FROM table1 [,table2]
      WHERE condition);
```

Ex: Find the names of sailors who have reserved boat 103.

```
SQL>SELECT S.sname FROM Sailors S WHERE S.sid IN ( SELECT R.sid FROM Reserves R WHERE R.bid = 103 );
```

The nested sub query computes the (multi)set of sids for sailors who have reserved boat 103 (the set contains 22,31, and 74) and the top-level query retrieves the names of sailors whose sid is in this set. The IN operator allows us to test whether a value is in a given set of elements;



**CORRELATED NESTED QUERIES:**

In the nested queries the inner sub query has been completely independent of the outer query. In general, the inner sub query could depend on the row currently being examined in the outer query.

Ex: Find the names of sailors who have reserved boat number 103.

```
SQL>SELECT S.sname FROM Sailors S WHERE EXISTS (SELECT * FROM Reserves R WHERE
R.bid = 103 AND R.sid = S.sid );
```

The EXISTS operator is another set comparison operator, such as IN. It allows us to test whether a set is nonempty, an implicit comparison with the empty set.

This query also illustrates the use of the special symbol \* in situations where all we want to do is to check that a qualifying row exists, and do not really want to retrieve any columns from the row. This is one of the two uses of \* in the SELECT clause that is good programming style; the other is as an argument of the COUNT aggregate operation. By using NOT EXISTS instead of EXISTS, we can compute the names of sailors who have not reserved a red boat. Closely related to EXISTS is the UNIQUE predicate.

**Set-Comparison Operators:**

The set-comparison operators EXIST, IN, and UNIQUE, along with their negated versions. SQL also supports op ANY and op ALL, where operator is one of the arithmetic comparison operators {<, <=, =, <>, >=, >}.

Ex: Find sailors whose rating is better than some sailor called Horatio.

```
SELECT S.sid FROM Sailors S WHERE S.rating > ANY (SELECT S2.rating FROM Sailors S2
WHERE S2.sname = 'Horatio');
```

If there are several sailors called Horatio, this query finds all sailors whose rating is better than that of some sailor called Horatio. On instance sailors this computes the sids 31, 32, 58, 71, and 74.

Ex: Find sailors whose rating is better than every sailor' called Horatio.

```
SQL> SELECT S.sid FROM Sailors S WHERE S.rating > ALL (SELECT S2.rating FROM Sailors S2
WHERE S2.sname = 'Horatio');
```

On instance sailors we would get the sid", 58 and 71

**AGGREGATE OPERATORS:**

To simply retrieving data we often want to perform some computation or summarization. SQL allows the use of arithmetic expressions. We now consider a powerful class of constructs for computing aggregate values such as MIN and SUM. These features represent a significant extension of relational algebra. SQL supports five aggregate operations, which can be applied on any column.

1. **COUNT** ([DISTINCT] A): The number of (unique) values in the A column.

Ex: Count the number of sailors.

```
SQL>select count(*) from sailors;
```

Ex: Count the number of different sailor names.

```
SQL>select count(distinct sname) from sailors;
```

2. **SUM** ([DISTINCT] A): The sum of all (unique) values in the A column.

Ex: sum the rating of all sailors;

```
SQL>select sum(rating) from sailors;
```

3. **AVG** ([DISTINCT] A): The average of all (unique) values in the A column.

Ex: Average age of all sailors.

```
SQL> select avg(age) from sailors;
```

4. **MAX (A)**: The maximum value in the A column.

Ex: Maximum rating of sailors.

SQL>select max(rating) from sailors;

5. **MIN (A)**: The minimum value in the A column.

Ex: Minimum rating of sailors.

SQL>select min(rating) from sailors;

### **The Group By And Having Clauses:**

**GROUP BY clause** : This clause is used in SELECT command to **group** the rows of specified table which column has same values and display only one name for each group. It is most useful for aggregate functions such as sum(), avg(), count() etc.,

Syntax : SELECT column\_name [, aggregate name] FROM <table name> [WHERE <condition>]  
GROUP BY column\_name;

→ The 'column\_name' of SELECT command and which used in GROUP BY clause must be same.

→ The <table name> is name of table.

→ The aggregate name is any function such as sum(), avg() etc.,

Eg: 1) Find the age of the youngest sailor for each rating level.

SQL> select sname,min(age) from sailors group by rating;

→ It will display only one sailors name and age for each rating group

Eg: 2) Find the age of the youngest sailor who is eligible to vote (i.e., is at least 18 years old) for each rating level with at least h.uo such sailors.

SQL> SELECT S.rating, MIN (S.age) AS minage FROM Sailors S WHERE S.age >= 18 GROUP BY S.rating HAVING COUNT (\*) > 1;

**HAVING clause** : This clause is used in SELECT command and it is similar to WHERE clause, but it identifies groups which match in a specified condition.

Syntax : SELECT column\_name [, aggregate name] FROM <table name> [WHERE <condition>]  
GROUP BY column\_name HAVING <condition>;

→ The 'column\_name' of SELECT command and which used in GROUP BY clause must be same.

→ The <table name> is name of table.

→ The aggregate name is any function such as sum(), avg() etc.,

→ The HAVING < condition> contains relational expression and one of the column must be Aggregate function.

Eg: 1) SELECT JOB FROM EMP GROUP BY JOB HAVING AVG(SAL) > 3000;

→ It will display only one job name for each group and whose employee average salary is > 3000.

Eg: 2) SELECT AVG(SAL), JOB FROM EMP GROUP BY JOB HAVING MAX(SAL) > 3000;

→ It will display only one job name, and average salary for each group and whose maximum salary > 3000.

### **NULL VALUES:**

SQL provides a special column value called null to use in such situations. We use null when the column value is either unknown or inapplicable. Using the Sailor table definition, we might enter the row (98, Dan, null, 39). The presence of null values complicates many issues these issues are

- Comparisons using null values
- Logical connectivity AND, OR and NOT
- Impact on SQL construct
- Outer joins
- Disallowing Null Values

**Comparisons using null values:**

SQL also provides a special comparison operator IS NULL to test whether a column value is null; for example, we can say rating IS NULL, which would evaluate to true on the row representing Dan. We can also say rating IS NOT NULL, which would evaluate to false on the row for Dan.

SQL> select \* from sailors where rating is null;

**Logical connectivity AND, OR and NOT:**


Boolean expressions such as rating = 8 OR age < 40 and rating = 8 AND age < 40? Considering the row for Dan again, because age < 40, the first expression evaluates to true regardless of the value of rating, but what about the second? We can only say unknown

We must define the logical operators AND, OR, and NOT using a three-valued logic in which expressions evaluate to true, false, or unknown. We extend the use interpretations of AND, OR, and NOT to cover the case when one of the arguments is unknown as follows. The expression NOT unknown is defined to be unknown. OR of two arguments evaluates to true if either argument evaluates to true, and to unknown if one argument evaluates to false and the other evaluates to unknown. (If both arguments are false, of course, OR evaluates to false.) AND of two arguments evaluates to false if either argument evaluates to false, and to unknown if one argument evaluates to unknown and the other evaluates to true or unknown. (If both arguments are true, AND evaluates to true.)

**Impact on SQL construct:**

the arithmetic operations +, -, \*, and / all return null if one of their arguments is null. However, nulls can cause some unexpected behavior with aggregate operations. COUNT(\*) handles 'null values just like other values; that is, they get counted. All the other aggregate operations (COUNT, SUM, AVG, MIN, MAX, and variations using DISTINCT) simply discard null values

**Outer Joins:**

Interesting variants of the join operation that rely on null values, called outer joins, are supported in SQL. Consider the join of two tables, say Sailors  Reserves. Tuples of Sailors that do not match some row in Reserves according to the join condition do not appear in the result. In an outer join, on the other, Sailor rows without a matching Reserves row appear exactly once in the result, with the result columns inherited from Reserves assigned null values.

In a left outer join,

Sailor rows without a matching Reserves row appear in the result but not vice versa. In a right outer join, Reserves rows without a matching Sailors row appear in the result, but not vice versa. In a **full** outer join, both Sailors and Reserves rows without a match appear in the result.

SQL allows the desired type of join to be specified in the FROM clause. For example, the following query lists

SQL>SELECT S.sid, R.bid FROM Sailors S LEFT OUTER JOIN Reserves R on s.sid=r.sid;

**Disallowing null values:**

Sql also provides one constraint not null, disallow null values by specifying NOT NULL as part of the field definition for example, sname CHAR(20) NOT NULL. In addition, the fields in a primary key are not allowed to take on null values. Thus, there is an implicit NOT NULL constraint for every field listed in a PRIMARY KEY constraint.

**COMPLEX INTEGRITY CONSTRAINTS IN SQL:**

complex integrity constraints that utilize the full power of SQL queries, complement the integrity constraint provides several features. These features are

- Constraints over a single table

- Domain constraints and distinct types
- Assertions: ICs over several tables

### Constraints over a single table:

Complex constraints over a single table using table constraints, which have the form CHECK conditional-expression. For example, to ensure that rating must be an integer in the range 1 to 10, we could use

```
CREATE TABLE Sailors (
    sid number(3),
    sname CHAR(10),
    rating number(3),
    age number(3,1),
    PRIMARY KEY (sid),
    CHECK (rating >= 1 AND rating <= 10 ));
```

### Domain constraints and distinct types:

A user can define a new domain using the CREATE DOMAIN statement, which uses CHECK constraints

```
CREATE DOMAIN ratingval INTEGER DEFAULT 1 CHECK ( VALUE >= 1 AND VALUE <= 10 )
```

INTEGER is the underlying, or source, type for the domain ratingval, and every ratingval value must be of this type. Values in ratingval are further restricted by using a CHECK constraint; in defining this constraint, we use the keyword VALUE to refer to a value in the domain. By using this facility, we can constrain the values that belong to a domain using the full power of SQL queries. Once a domain is defined, the name of the domain can be used to restrict column values in a table.

The optional DEFAULT keyword is used to associate a default value with a domain. If the domain ratingval is used for a column in some relation and no value is entered for this column in an inserted tuple, the default value 1 associated with ratingval is used.

### Assertions: ICs over several tables:

Table constraints are associated with a single table, although the conditional expression in the CHECK clause can refer to other tables. Table constraints are required to hold only if the associated table is nonempty. When a constraint involves two or more tables, To cover such situations, SQL supports the creation of assertions, which are constraints not associated with anyone table.

As an example, suppose that we wish to enforce the constraint that the number of boats plus the number of sailors should be less than 100.

```
CREATE ASSERTION smallClub CHECK (( SELECT COUNT (S.sid) FROM Sailors S )
+ ( SELECT COUNT (B. bid) FROM Boats B)< 100 );
```

### TRIGGERS AND ACTIVE DATABASES:

A trigger is a procedure that is automatically invoked by the DBMS in response to specified changes to the database, and is typically specified by the DBA. A database that has a set of associated triggers is called an **active database**. A trigger description contains three parts:

- **Event:** A change to the database that activates the trigger.
- **Condition:** A query or test that is run when the trigger is activated.
- **Action:** A procedure that is executed when the trigger is activated and its condition is true.

A trigger can be thought of as a 'daemon' that monitors a database, and is executed when the database is modified in a way that matches the event specification. An insert, delete, or update statement could activate a trigger, regardless of which user or application invoked the activating statement; users may not even be aware that a trigger was executed as a side effect of their program.



A condition in a trigger can be a true/false statement (e.g., all employee salaries are less than \$100,000) or a query. A query is interpreted as true if the answer set is nonempty and false if the query has no answers. If the condition part evaluates to true, the action associated with the trigger is executed

for example, we have to examine the age field of the inserted Students record to decide whether to increment the count.

```
CREATE TRIGGER inc_count BEFORE INSERT ON Students      * Event *
DECLARE
count INTEGER;
BEGIN                                                  * Action *
count := 0;
END;

CREATE TRIGGER inc_count AFTER INSERT ON Students      * Event *
WHEN (new.age < 18)                                   * Condition; 'new' is just-inserted tuple *
FOR EACH ROW
BEGIN                                                  * Action; a procedure in Oracle's PL/SQL syntax *
Count := count + 1;
END;
```

Trigger event should be defined to occur for each modified record; the FOR EACH ROW clause is used to do this. Such a trigger is called a **row-level trigger**. On the other hand, the inc\_count trigger is executed just once per INSERT statement, regardless of the number of records inserted, because we have omitted the FOR EACH ROW phrase. Such a trigger is called a **statement-level trigger**.

The keyword new refers to the newly inserted tuple. If an existing tuple were modified, the keywords old and new could be used to refer to the values before and after the modification.

#### Ex: Create the Before insert row level trigger

```
Create or replace trigger trgbir
before
insert on student for each row
begin
    :new.sname := upper(:new.sname);
end;
```

#### Create the After delete statement level trigger

```
create or replace trigger trgads
after
delete on student
begin
raise_application_error(-20015,'you cannot delete this row');
end;
```

**SQL operators :** The SQL operators are used to perform the operation on values of columns. The

SQL operators are **four** types. They are 1) Arithmetic operators

2) Relational operators 3) Logical operators 4) Special operators.

1) Arithmetic operators : The Arithmetic operators are used to link the two operands for getting a new results. The Arithmetic operators are Addition(+), Subtraction(-) multiplication (\*), division (/), and modulo (%).

Eg 1 : SELECT emp\_no, emp\_name, sal+hra FROM emp;

It will display emp number, employee name and sal with adding hra from emp table.

Eg 2 : SELECT emp\_no, emp\_name, sal - tax FROM emp;

It will display emp number, employee name and sal by subtracting tax from emp table.

Eg 3 : SELECT emp\_no, emp\_name, sal \* 12 / 100 FROM emp;

2) Relational operators : The Relational operators are used to compare two operands for further process. The Relational operators are

Operator	Meaning
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to
=	equal to
<>	not equal to

Eg 1 : SELECT \*FROM emp WHERE emp\_no >=10;

It will display rows of data which emp\_no is greater than or equal to 10.

Eg 2 : SELECT \*FROM emp WHERE emp\_no <=10;

It will display rows of data which emp\_no is less than or equal to 10.

Eg 3 : SELECT \*FROM emp WHERE emp\_no < > 10;

It will display rows of data which emp\_no is not equal to 10.

3) Logical operators : The Logical operators are used to compare two relational expressions. The Logical operators are AND, OR, and NOT operators.

Eg 1 : SELECT \*FROM emp WHERE emp\_no >= 10 AND emp\_no <= 20;

It will display rows of data which emp\_no are in between 10 and 20.

Eg 2 : SELECT \*FROM emp WHERE emp\_no >=10 or sal = 4500;

It will display rows of data which emp\_no is greater than or equal to 10 or which sal is equal to 4500.

4) Special operators : The special operators are used to match the values in given list.

The special operators are

a) BETWEEN operator      b) IN/NOT IN operator

c) LIKE operator              d) IS NULL operator

a)BETWEEN operator : This operator is used to match the particular column values in the given range. The range consists lowest value and highest value.

Syntax : BETWEEN low value AND high value ;

Eg: 1 SELECT \*FROM emp WHERE emp\_no BETWEEN 5 AND 10;

It will display the rows of data from emp table which employee number in between 5 and 10.

Eg: 2 SELECT \*FROM emp WHERE sal BETWEEN 5000 AND 10000;

It will display the rows of data from emp table which employee salary is in between 5000 and 10000.

**b) i) IN operator:** The IN operator is used in where clause to match the values of column in the given list.

**Syntax:** IN (list)

Eg: 1 SELECT \*FROM emp WHERE emp\_no IN (5,6,7);

It retrieve the rows/records from emp table which emp\_no will match in the list.

Eg: 2 SELECT \*FROM emp WHERE emp\_name IN ('vasu','anil','ssn');

It retrieve rows/records from emp table which emp\_name will match in the list.

**b) ii) NOT IN operator:** The NOT IN operator is used in where clause to which column values are not match in the given list.

**Syntax:** NOT IN (list)

Eg: 1 SELECT \*FROM emp WHERE emp\_no NOT IN (5,6,7);

It will display the rows of data from emp table which emp\_no is not matched in the list.

Eg: 2 SELECT \*FROM emp WHERE emp\_name NOT IN ('vasu','anil','ssn');

It will display the rows of data from emp table which emp\_name is not matching in the list.

**c) LIKE operator:** The LIKE operator is used in where clause to match the possible values in the given list. I.e. it is possible to select rows that match a character pattern. The character pattern matching operation may be referred to as 'wild-card' search. The symbols can be used to construct the search string.

**Syntax:** LIKE 'symbol';

→ The Symbols are % and - (hyphen). The % symbol represents any sequence of zero or more characters and - (hyphen) represents any single character.

Eg: SELECT \*FROM emp WHERE empname LIKE 'a%';

It will display the rows of data from emp table which employee name starts with letter a.

Eg: SELECT \*FROM emp WHERE empname LIKE '----';

It will display the rows of data from emp table which employee name contain four letter data..

**d) IS NULL operator:** The IS NULL operator is used in where clause to test the NULL value in the given column name.

Eg: SELECT \*FROM emp WHERE hra IS NULL;

It will display the rows of data from emp table which hra value is null value.

**FUNCTION :** A function is a process that will manipulates data values and returns the results. It is similar to an operator. But it should contain the argument. The argument has a constant value.

**Rules:** 1. If you call a function with an argument, ORACLE converts the argument into the expected **type**. For example, Floor(15.7) returns 15. Here, input is float type and output is integer type.

2. If you call a function with a null argument, the function automatically returns null value Eg: ABS(-5), its absolute value is 5.

**Syntax :** SELECT function\_name(argument1, argument2 . . . ) FROM DUAL;

→ The function type is keyword such as ABS(), FLOOR(), MIN(), MAX() etc.

→ The argument is any operand (or) value (or) more than one value.

**Types of functions** : The SQL functions are divided into two types based on the function process. They are I). Single row functions II). Group functions.

I. **Single row functions** : A single row functions perform the process based on the input value and return the output value. In this case, it takes input one type of constant and produces another type. For example, Floor(15.7) returns 15. Here, input is float type and output is integer. The single row functions must use the keyword **dual**. The single row functions are

- i) Number functions      ii) Character functions      iii) Date functions
- iv) Conversion functions      v) Other functions.

I) **Number functions** : Number functions accept numeric value as input and return numeric value as output. Most of these functions return values that are accurate to 38 decimal digits. The functions are

1) **ABS (Absolute Value)** : This function returns the absolute value of n.

**Syntax** : ABS( n)    Where n is numeric value.

Eg: SELECT ABS(-15) FROM DUAL;

It will display the Absolute value as 15.

2) **CEIL (Ceiling)** : This function requires float value and returns the integer value. Suppose, the value after decimal point is greater than 0 then it will add one to integer part.

**Syntax** : CEIL( n)    Where n is numeric value.

Eg: SELECT CEIL(15.7) FROM DUAL;

It will display the output value as 16.

Eg: SELECT CEIL(15.4) FROM DUAL;

It will display the output value as 16.

Eg: SELECT CEIL(15.0) FROM DUAL;

It will display the output value as 15.

3) **COS(cosine)** : This function is used to returns the cosine of n

**Syntax** : COS( n)    Where n is numeric value.

Eg: SELECT COS(180) FROM DUAL;

It will display the output value as -.5984601

Eg: SELECT COS(0) FROM DUAL;

It will display the output value as 1

4) **COSH(cosine hyperbolic)**: This function is used to returns the hyperbolic cosine of n

**Syntax** : COSH( n)    Where n is numeric value.

Eg: SELECT COSH(0) FROM DUAL;

It will display the output value as 1

5) **EXP(exponentiation)** : This function is used to returns e to the power of n

Where e is 2.71828183

**Syntax** : EXP( n)    Where n is numeric value.

Eg: SELECT EXP(4) FROM DUAL;

It will display the e to the 4<sup>th</sup> power as 54.59815

6) **FLOOR()** : This function is used to returns the integer i.e. it will drop the decimal part .

**Syntax** : FLOOR( n)    Where n is numeric value.

Eg: `SELECT FLOOR(15.7) FROM DUAL;`  
It will display the output value as 15.

7) LN(Logarithm to base E) : This function is used to returns the natural logarithm of n. where n is greater than 0.

Syntax : `LN( n )` Where n is numeric value.

Eg: `SELECT LN(95) FROM DUAL;` It will display the natural log of 95 as 4.5538769

8) LOG(Logarithm to base 10): This function is used to returns the logarithm, base m of n. The base 'm' can be any positive number other than 0 or 1 and 'n' can be any positive number.

Syntax: `LOG(m,n)` Where 'm' can be positive number and 'n' can be any positive number.

Eg: `SELECT LOG(10,100) FROM DUAL;` It will display the log base 10 of 100 as 2.

9) MOD(modulus): This function is used to returns the remainder of m divided by n. It returns m if n is 0.

Syntax: `MOD(m,n)` Where 'm' and 'n' are numeric values.

Eg: `SELECT MOD(45,12) FROM DUAL;` It will display the remainder value of m/n as 9.

11) POWER(): This function is used to returns the m to the power of n.

Syntax: `POWER(m,n)` Where 'm' and 'n' are numeric values.

Eg: `SELECT POWER(3,2) FROM DUAL;` It will display the 3 to the power of 2 is as 9.

12) ROUND(): This function is used to return the n places with rounding from the value m. The m must be the decimal point. If n is omitted, it return the 0 places.  
The n must be an integer.

Syntax: `ROUND(m,n)` Where 'm' and 'n' are numeric values.

Eg: `SELECT ROUND(15.193,1) FROM DUAL;` It will display the output as 15.2.

Eg: `SELECT ROUND(15.193,2) FROM DUAL;`  
It will display the output as 15.19.

Eg: `SELECT ROUND(15.193,3) FROM DUAL;`  
It will display the output as 15.193.

Eg: `SELECT ROUND(15.193,-1) FROM DUAL;`  
It will display the output as 20.

13) SIGN(): This function is used to returns -1 if n < 0, returns 0 if n = 0, (or) returns 1 if n > 0.

Syntax: `SIGN( n )` Where 'n' is numeric value.

Eg: `SELECT SIGN(-15) FROM DUAL;`  
It will display the output as -1 because -15 is less than zero.

14) SIN(): This function is used to returns sine value of n.

Syntax: `SIN( n )` Where 'n' is numeric value.



Eg: SELECT SIN(60) FROM DUAL;  
It will display the output as -.3048106.

15 SINH(): This function is used to returns hyperbolic sine value of n.

Syntax: SINH( n ) Where 'n' is numeric value.

Eg: SELECT SINH(1) FROM DUAL;  
It will display the output as 1.1752012.

16 SQRT(): This function is used to returns the square root of n. The n value must not be negative. Sqrt returns a "real" result.

Syntax: SQRT( n ) Where 'n' is numeric value.

Eg: SELECT SQRT(16) FROM DUAL;  
It will display the Square root of 16 as 4.

17 TAN(): This function is used to returns the tangent of n.

Syntax: TAN( n ) Where 'n' is numeric value.

Eg: SELECT TAN(90) FROM DUAL;  
It will display the tangent value of 90 as -1.9952.

18 TANH(): This function is used to returns the hyperbolic tangent of n.

Syntax: TANH( n ) Where 'n' is numeric value.

Eg: SELECT TANH(90) FROM DUAL;  
It will display the hyperbolic tangent value of 90 as 1.

19 TRUNC(): This function is used to truncate the decimal values and display specified number of decimal places from the given number.

Syntax: TRUN( m , n ) Where 'm' and 'n' are numeric values.

Eg: SELECT TRUN(15.79655,1) FROM DUAL;  
It will truncate the decimal values based on 'n' value. Here n is 1. i.e. 15.7

Eg: SELECT TRUN(15.79655,3) FROM DUAL;  
It will truncate the decimal values based on the 'n' value. Here n is 3 i.e. 15.796

Eg: SELECT TRUN(15.79655,4) FROM DUAL;  
It will truncate the decimal values based on the 'n' value. Here 'n' is 4. i.e. 15.7965

II) **CHARACTER FUNCTIONS**: This function is used accept single row character as input and can return both character and number values. The character type value (or) column may be defined with the keyword char or varchar2. The character functions are

1) **CHR ( )** : This function is used to returns the character value. I.e. ASCII character.

Syntax : chr(n) where n is integer number.

Eg: SELECT CHR(75) FROM DUAL;  
It will the Ascii character of 75 as 'K'.

2) CONCAT() : This function is used to join the two character strings .

Syntax: CONCAT(char1,char2) The char1 and char2 are character data or character variables.

Eg: SELECT CONCAT('vasu','reddy') FROM DUAL;

It will concatenate (join) the two words 'vasu' and 'reddy' as 'vasu reddy'

Eg: SELECT CONCAT(empname,' is a boy') FROM EMP;

It will concatenate employee name with 's a person' . For example, if empname is hari means, it will display as hari is a person.

3) INITCAP() : This function is used to display the first letter of each word in uppercase, all other letters in lowercase. It is allowed in alphabetic characters only.

Syntax: INITCAP(char) The char must be alphabetic data only.

Eg: SELECT INITCAP('the indian lady') FROM DUAL;

It will display the first letter of each word in upper case. I.e. it will display as 'The Indian Lady'

4) LOWER() : This function is used to display all letters in lower case.

Syntax: LOWER(char) The char must be alphabetic type data only.

Eg: SELECT LOWER('INDIA') FROM DUAL;

It will display the upper case INDIA in lower case letters as 'india'

5) LPAD() : This function is used to display the specified symbol if 'n' is more than the given character data.

Syntax: LPAD(char) The char is alphanumeric type data.

Eg: SELECT LPAD('INDIA',5,'\*') FROM DUAL;

It will display word as INDIA only.

Eg: SELECT LPAD('INDIA',6,'\*') FROM DUAL;

It will display word as \*INDIA. Here number is more than character data, so it print \* symbol at first place.

Eg: SELECT LPAD('INDIA',8,'\*') FROM DUAL;

It will display word as \*\*\*INDIA. Here number is more than character data, so it print \*\*\* symbols at first place.

6) LTRIM() : This function is used to remove the characters from the left of character data. That is, if the specified right most characters are same of first two characters in the character data.

Syntax: LTRIM(char) The char is alphanumeric type data.

Eg: `SELECT LTRIM('indian','in') FROM DUAL;`  
It will remove first two character and display output as 'dian'.

Eg: `SELECT LTRIM('indian','ab') FROM DUAL;`  
It will not remove first two character, because the given characters are not same as the first two characters. So, the output is indian.

7) `NLS_INITCAP()` : This function is used to display the first character in upper case for all words which in the character data. It is similar to `INITCAP()` function.

Syntax: `NLS_INITCAP(char)` The char is alphabetic only.

Eg: `SELECT NLS_INITCAP('india is my country') FROM DUAL;`  
It will display the output as follows. 'India Is My Country. In this first letter upper case letter.

8) `NLS_LOWER()` : This function is used to display all characters in lower case. It is same as the `LOWER()` function.

Syntax: `NLS_LOWER(char)` The char must be alphabetic type data only.

Eg: `SELECT NLS_LOWER('INDIA') FROM DUAL;`  
It will display the upper case word 'INDIA' in lower case letters as 'india'

9) `NLS_UPPER()` : This function is used to display all characters in upper case. It is same as the `UPPER()` function.

Syntax: `NLS_UPPER(char)` The char must be alphabetic type data only.

Eg: `SELECT NLS_UPPER(' india ') FROM DUAL;`  
It will display the lower case word 'inida' in upper case letters as 'INDIA'

10) `REPLACE function()` : This function is used to replace the letter or word with given letter or word.

Syntax: `REPLACE(char)` The char is alphanumeric data.

Eg: `SELECT REPLACE(' India is my Country','India','INDIA') FROM DUAL;`  
It will display the out put as follows. 'INDIA is my Country' In this, the first word 'India' is replaced by 'INDIA'.

Eg: `SELECT REPLACE(' NSS students are obedience','NSS','SSN') FROM DUAL;`  
It will display the input 'NSS students are obedience' as 'SSN students are obedience'.

11) `RPAD function()` : This function is used to replicate the given character a specified number of time. That is, if the given number is grater than the stirng(character data) it will print the given character.

Syntax: `RPAD(char)` The char is alphanumeric data.

Eg: `SELECT REPLACE(' India',8,'*') FROM DUAL;`  
It will display the out put as follows.  
'India\*\*\*' It displays 3 stars at right side of the string. Because the number is greater than the string (India).

12) RTRIM() : This function is used to remove the characters from the right of character data. That is, if the specified characters are same of last characters in the character data then it removes those characters.

Syntax: RTRIM(char) The char is alphanumeric type data.

Eg: SELECT RTRIM('Indian','an') FROM DUAL;

It will remove last two character and display output as 'Indi'.

Eg: SELECT RTRIM('Indian','ab') FROM DUAL;

It will not remove last two character, because the given characters are not same as the first two characters. So, the output is Indian.

13) SUBSTR() : This function is used to display the character from specified position to the specified number characters in the given character data.

Syntax: SUBSTR(char,pos, n)

The char is alphanumeric type data.

The pos is position in the given data.

The n is number that specifies number characters from the position.

Eg: SELECT SUBSTR('SSN COLLEGE',5,7) FROM DUAL;

It will display the character data from 5th position to 7 characters. I.e. 'COLLEGE'

Eg: SELECT SUBSTR('SSN COLLEGE',1,3) FROM DUAL;

It will display the character data from 1st position to 3 characters. I.e. 'SSN'

14) UPPER() : This function is used to display all characters in upper case. It is same as the NLS\_UPPER() function.

Syntax: UPPER(char) The char must be alphabetic type data only.

Eg: SELECT UPPER(' india ') FROM DUAL;

It will display the lower case word 'inida' in upper case letters as 'INDIA'

15) ASCII() : This function is used to convert the given character into equivalent ascii code.

Syntax: ASCII(char) The char must be alphabetic type data only.

Eg: SELECT ASCII('A') FROM DUAL;

It will display the equivalent ASCII value 65 for given character 'A'

Eg: SELECT ASCII('a') FROM DUAL;

It will display the equivalent ASCII value 97 for given character 'a'

16) INSTR() : This function is used to display the specified position character at what location in the given character data.

Syntax: INSTR(char,pos, n)

The char is any alphabetic type data.

The pos is used to specify that the character is at what position.

The 'n' specifies how many times the character is in the data.

Eg: SELECT INSTR('we are all ssn studnets','s',12,2) FROM DUAL;

It will display the second letter 's' at what position in data. That is at 13.

Eg: SELECT INSTR('we are all ssn studnets','s',12,1) FROM DUAL;

It will display the first letter 's' at what position in data. That is at 12.

Eg: SELECT INSTR('we are all ssn studnets','s',12,3) FROM DUAL;

It will display the third letter 's' at what position in data. That is at 16.

17) LENGTH() : This function is used to display the length of given character data (string). It counts the null space also.

Syntax: LENGTH(char)

The char is any alphanumeric type data.

Eg: SELECT LENGTH('ssn college') FROM DUAL;

It will display the length of character data (string). I.e. 11

**III) DATE functions :** Date functions operate on values of DATE data type. All date functions return a value of DATE data type, except the MONTHS\_BETWEEN function, which returns a number.

1) ADD\_MONTHS() : This function is used to return the DATE plus n months. It means, return a date by adding specified number of months. For example, the given date is '17-dec-08', it will be displayed as '17-jan-09' when we added 'one' to it.

Syntax : ADD\_MONTHS(d,n)

The d may be date (or) date variable.

The n is the integer number, it is used to add number of months to the month. For

example

Eg : select hiredate from emp;

It will display 14 rows of column hiredate values.

Eg: select add\_months(hiredate,1) from emp;

It will display 14 rows of column hiredate values from emp with adding one month

Eg: select add\_months(hiredate,3) from emp;

It will display 14 rows of column hiredate values from emp with adding 3 months

2) LAST\_DAY() : This function returns the date corresponding to the last day of the month. Suppose, the system date is 17-sep-08. The last day of this month is 30-sep-08.

Syntax : Last\_day(Column\_name); The column\_name must be date type.

Eg: SELECT sysdate, last\_day(sysdate) from dual;

It will display system date and last day of today system date. I.e Today date is 17-sep-08 then last day is 30-sep-08.

3) MONTHS\_BETWEEN() : This function returns a number of months between dates i.e date1 and date2. If date 1 is later than date2, then result is negative. If the date1 is earlier than date2, result is positive.

Syntax: MONTHS\_BETWEEN(date1,date2)

The date1 and date2 are date type.

For example, 1)select months\_between('17-sep-08','1-oct-08') from dual;

2)select months\_between(hiredate,sysdate) from emp;

4) NEXT\_DAY() : This function is used to return the date of the first week day.

It means later than the date 'd'.

Syntax: NEXT\_DAY(d,char)

➤ The 'd' is date value (or) date type identifier (or) sysdate keyword.

➤ The char is any week day such as 'Monday','Tuesday', . . . . .

This day must later day of 'd'

Eg: Select next\_day('17-sep-08','Thursday') from dual;



- In this '17-sep-08' is today date and its day is 'Wednesday' . But, it will display the next day of Thursday. That is 'Friday' . So, it will display Friday date as '19-sep-08'.

5)NEW\_TIME(): This function is used to return the date and time.

Syntax: NEXT\_TIME(d,z1,z2)

- The 'd' is date value (or) date type identifier (or) sysdate keyword.
- The z1 and z2 are time zones
- The z2 is used to display the date of z1 format in z2 format.

i.e if z1 is AST format and z2 is CST format, then the date of z1 format will be displayed in z2 format.

#### Time zones are

1. AST/ADT -> Atlantic standard
2. BST/BDT -> Bearing standard
3. CST/CDT -> Central standard
4. EST/EDT -> Eastern standard
5. GMT -> Green which mean Time
6. HST/HDT -> Alaska – Hawli standard Time
7. MST/MDT -> Mountain standard time
8. NST -> New found Land standard time
9. PST/PDT -> Pacific standard time
10. YST/YDT -> Yukan standard time

6) ROUND(): This function returns the date which rounded to the unit specified by the format 'fmt' . It means, it round to the nearest day.

Syntax: ROUND(d, fmt)

- The letter 'd' represents date.
  - The word 'fmt' represents year, month (or) day
- Eg: 1)SELECT hiredate, ROUND(hiredate,'year') from emp;  
 Eg: 2)SELECT hiredate, ROUND(hiredate,'month') from emp;  
 Eg: 3)SELECT hiredate, ROUND(hiredate,'day') from emp;  
 Eg: 4)SELECT ROUND(sysdate) from dual;  
 Eg: 5)SELECT ROUND(sysdate,'year') from dual;  
 Eg: 6)SELECT ROUND(sysdate,'month') from dual;  
 Eg: 7)SELECT ROUND(sysdate,'day') from dual;

7)SYSDATE: This keyword is used to return the current date and time.

Syntax : SYSDATE;

Eg: SELECT SYSDATE FROM DUAL;

Eg: SELECT SYSDATE 'mm-dd-yyyy hh24' NOW FROM DUAL;

8) TRUNC() : This function is used to returns date with the time portion of the day truncated which is specified by format 'fmt'. If 'fmt' is omitted, the date is converted to the nearest day.

For example, if the SYSDATE is '27-jan-99', then truncated result is 01-jan-99. It means, 27 truncated as 01 in the date.

Syntax: TRUNC(d, fmt)

- Eg: 1)SELECT TRUNC(sysdate,'year') from dual;  
 Eg: 2)SELECT TRUNC(sysdate,'month') from dual;  
 Eg: 3)SELECT TRUNC(sysdate,'day') from dual;  
 Eg: 4)SELECT TRUNC(hiredate,'year') from emp;  
 Eg: 5)SELECT TRUNC(hiredate,'month') from emp;  
 Eg: 6)SELECT TRUNC(hiredate,'day') from emp;  
 Eg: 7)SELECT TRUNC(sysdate) from dual;

**IV : CONVERSION FUNCTIONS :** The conversion functions are used to convert a value from one 'data type' to another. It means, char type to number type and so on. The Conversion functions are

1) **TO\_CHAR():** The function is used to convert date into varchar2 type in the default date format.  
Syntax : TO\_CHAR(date,[fmt]);

Eg: 1) SELECT TO\_CHAR(sysdate,'dd month yyyy') from dual;

It will display system date as 18 September 2008.

Eg: 2) SELECT TO\_CHAR(sysdate,'yyyy "years" mm "month" dd "days"') from dual;  
It will display system date as 2008 years September month 18 days.

2) **TO\_NUMBER():** This function allows the conversion of string (varchar2) type data (number) into number type in the specified format.

Syntax: TO\_NUMBER(char[,fmt]);

➤ The word 'char' is character type data (or) variable

➤ The word 'fmt' is optional that represent format (i.e. AST/BST/CST etc.,

Eg: SELECT TO\_NUMBER('100') FROM DUAL;

3) **TO\_DATE() :** This function is used to converts **character type** data into **date type** data.

Syntax: TO\_DATE(char[,fmt]);

➤ The word 'char' is character type data (or) variable

➤ The word 'fmt' is optional that represent format (i.e. AST/BST/CST etc.,

Eg: SELECT ('September 18 2008' , ' month-dd-yyyy') from dual;

4) **ROWIDTOCHAR() :** This function is used to convert **ROWID** value into **character type** value (I.e varchar2 type). Syntax: ROWIDTOCHAR(rowid);

➤ The word 'rowid' is a keyword that generate numbers automatically at the time of adding rows into table.

Eg: SELECT ROWIDTOCHAR(rowid) from dual/emp;

**V) OTHER FUNCTIONS :** The other functions are GREATEST(), LEAST(), NVL() and VSIZE().

1) **GREATEST():** This function is used to find greatest string from given list of strings.

Syntax : GREATEST(string1,string2,string3, . . . . .);

Eg: SELECT GREATEST('BVS', 'SSN', 'RISE', 'PASE', 'QIS') FROM DUAL;

2) **LEAST():** This function is used to find least string from given list of strings.

Syntax : LEAST(string1,string2,string3, . . . . .);

Eg: SELECT LEAST('BVS', 'SSN', 'RISE', 'PASE', 'QIS') FROM DUAL;

3) **NVL():** This function is used to display the specified message which column contain 'null' values in a row. Syntax : NVL(exp1,exp2, . . . . .);

→ The word 'exp1' is any column name of specified table

→ The word 'exp2' is any string that used to replace by null value.

Eg: SELECT NVL(comm,'NOT Applicable') FROM EMP;

It will replace all null values with 'NOT Applicable' which are in the column 'comm' of EMP table.

4) **VSIZE() :** This function is used to find the number of bytes the value occupies in the given variable or column name.

Syntax: VSIZE(column\_name);

Eg: SELECT ENAME VSIZE(ENAME) 'SIZE IN BYTES' FROM EMP;