**Structure Query Language(SQL)**

Structure Query Language(SQL) is a language used for storing and managing data in RDBMS. SQL was the first commercial language introduced for E.F Codd's Relational model. Today almost all RDBMS(MySql, Oracle, Informix, Sybase, MS Access) uses SQL as the standard database language. SQL is used to perform all type of data operations in RDBMS.

SQL is widely popular because it offers the following **advantages:**

• Allows users to access data in the relational database management systems.

• Allows users to describe the data.

• Allows users to define the data in a database and manipulate that data.

• Allows embedding within other languages using SQL modules, libraries & pre-compilers.

• Allows users to create and drop databases and tables.

• Allows users to create view, stored procedure, functions in a database.

• Allows users to set permissions on tables, procedures and views.

**A Brief History of SQL**

• 1970 − Dr. Edgar F. Ted Codd of IBM is known as the father of relational databases. He described a

relational model for databases.

• 1974 − Structured Query Language appeared.

• 1978 − IBM worked to develop Codd's ideas and released a product named System/R.

• 1986 − IBM developed the first prototype of relational database and standardized by ANSI.

The first relational database was released by Relational Software which later came to be known as Oracle. Oracle being an example, even implemented SQL as the relational database language for their DBMSs concurrently with IBM. Later on, SQL language was standardized by ANSI and ISO. The ANSI SQL standards were first published in 1986 and updated in 1989, 1992, and 1999.

**Data types in SQL**

When we create a table we must specify a data type for each of its columns. These data types define domain of values that each column can take. Oracle provides a number of built-in data types as well as several categories for user-defined types that can be used as data types. Some of the built-in data types are string data type to store characters, number data type to store numerical value, and date and time data type to store when the event happened (history, date of birth, etc.).

• CHARACTER [(length)] or CHAR [(length)]

• VARCHAR (length)

• BOOLEAN

• NUMBER

• DATE

**CHARACTER [(length)] or CHAR [(length)]**

The CHARACTER data type accepts character strings, including Unicode, of a fixed length. The length of the character string should be specified in the data type declaration; for example, CHARACTER (n) where n represents the desired length of the character string. If no length is specified during the declaration, the default length is 1.

The minimum length of the CHARACTER data type is 1 and it can have a maximum length up to the table page size. Character strings that are larger than the page size of the table can be stored as a Character Large Object (CLOB).

CHAR(10) or CHARACTER(10)

EX: 'Race car','RACECAR','24865'

**VARCHAR vs. VARCHAR2**:

The VARCHAR data type behaves like VARCHAR2 data type in the current version of Oracle. The varchar2 datatype supports a variable length character string. It also stores alpha numeric values. The size for this datatype ranges from 1-4000 bytes. In order to justify the above statement, let us create a table CHAMPION, which refers to Wimbledon Champions.

The attributes of the table CHAMPION are Name, Nation, and Year (the year in which the sportsman has won the title). For our example, let us use the data type VARCHAR for the attribute Name and VARCHAR2 for the data type Nation.

**BOOLEAN:**

The BOOLEAN data type supports the storage of two values: TRUE or FALSE. No parameters are required when declaring a BOOLEAN data type.

**NUMBER**:

Datatype the NUMBER datatype stores zero, positive, and negative fixed- and floating-point numbers.

The syntax to store fixed-point number is NUMBER (p, q)

―p is the total number of digits and

―q is the number of digits to the right of decimal point. The syntax to specify an integer is NUMBER (p).

**DATE**:

Data type The DATE data type is used to store the date and time information. For each DATE value,Oracle stores the century, year, month, date, hour, minute, and second information.

The ANSI date literal contains no time portion, and must be specified in **DD-MON-YY** format where Y stands for Year, M for month, and D for date.

**LOB** Data type Multimedia data like sound, picture, and video need more storage space.

The LOB data types such as BLOB, CLOB, and BFILE allows us to store large block of data.

**BLOB** Data type The BLOB data type stores unstructured binary data in the database. BLOBs can store up to 4 GB of binary data.

**CLOB** Data type The CLOB data type can store up to 4 GB of character data in the database.

**BFILE** Data type The BFILE data type stores unstructured binary data in operating system files outside the database. A BFILE can store up to 4 GB of data.

**SQL Commands**

SQL commands are instructions, coded into SQL statements, which are used to communicate with the database to perform specific tasks, work, functions and queries with data.

SQL commands can be used not only for searching the database but also to perform various other functions like, for example, you can create tables, add data to tables, or modify data, drop the table, set permissions for users. SQL commands are grouped into four major categories depending on their functionality:

**• Data Definition Language (DDL)** - These SQL commands are used for creating, modifying, and dropping the structure of database objects. The commands are **CREATE, ALTER, DROP, RENAME.**

**• Data Manipulation Language (DML)** - These SQL commands are used for storing, retrieving, modifying, and deleting data.

These Data Manipulation Language commands are: **SELECT, INSERT, UPDATE, and DELETE.**

**• Transaction Control Language (TCL)** - These SQL commands are used for managing changes affecting the data. These commands are **COMMIT, ROLLBACK, and SAVEPOINT**.

**• Data Control Language (DCL)** - These SQL commands are used for providing security to database objects. These commands are **GRANT and REVOKE**.

**DDL (Data Definition Language)**

DDL statements are used to alter/modify a database or table structure and schema. These statements handle the design and storage of database objects.

**CREATE** – create a new Table, database,schema

**ALTER** – alter existing table, column description

**DROP** – delete existing objects from database

Create Table Command – The CREATE TABLE command is used to implement the schemas of individual relations.

**Syntax**

CREATE TABLE table name (column name1 data type (size), column name2 data type (size), column nameN data type (size));

The example creates a table called "Persons" that contains five columns: PersonID, LastName, FirstName,Address:

CREATE TABLE Persons ( PersonID NUMBER(5),LastName varchar(10), FirstName varchar(10), Address varchar(15) );

Syntax: DESC table name;

The **DESC command** returns the attributes (columns) of the table, the datatype associated with the column, and also any constraint (if any) imposed on the column.

**ALTER TABLE** Statement

The ALTER TABLE statement is used to **add, delete, or modify** columns in an existing table.

ALTER TABLE statement is also used to **add and drop** various constraints on an existing table.

To **add** a column in a table, use the following syntax:

ALTER TABLE table\_name ADD column\_name datatype;

EX:

SQL> ALTER TABLE PERSON ADD CITY VARCHAR (10);

To **delete** a column in a table, use the following syntax:

ALTER TABLE table\_name DROP COLUMN column\_name;

EX:

SQL> ALTER TABLE PERSON DROP CITY;

To **change** the data type of a column in a table

ALTER TABLE table\_name MODIFY COLUMN column\_name datatype;

EX: ALTER TABLE MODIFY COLUMN ADDRESS CHAR(10);

**DROP TABLE Statement**

The DROP TABLE statement is used to drop an existing table in a database.

Syntax:

drop table <table name>;

If this statement is successfully executed then the message “table dropped” would be displayed.

**SQL Constraints**

Constraints are used to limit the type of data that can go into a table. Constraints can be specified when a table is created (with the CREATE TABLE statement) or after the table is created (with the ALTER TABLE statement).

We will focus on the following **constraints**:

* NOT NULL
* UNIQUE
* PRIMARY KEY
* FOREIGN KEY
* CHECK
* DEFAULT

**Not Null**

The NOT NULL constraint enforces a column to NOT accept NULL values. The NOT NULL constraint enforces a field to always contain a value. This means that you cannot insert a new record, or update a record without adding a value to this field.

The following SQL enforces the "P\_Id" column and the "LastName" column to not accept NULL values:

CREATE TABLE Persons ( P\_Id number(3) NOT NULL, LastName varchar(255) NOT NULL, FirstName varchar(255), Address varchar(255), City varchar(255));

**Unique:**

The UNIQUE constraint uniquely identifies each record in a database table. The UNIQUE and PRIMARY KEY constraints both provide a guarantee for uniqueness for a column or set of columns. A PRIMARY KEY constraint automatically has a UNIQUE constraint defined on it.

**Note** that you can have many UNIQUE constraints per table, but only one PRIMARY KEY constraint per table.

CREATE TABLE Persons ( P\_Id number(4) NOT NULL UNIQUE, LastName varchar(255) NOT NULL, FirstName varchar(255), Address varchar(255), City varchar(255) ) ;

**Primary Key**

The PRIMARY KEY constraint uniquely identifies each record in a database table. Primary keys must contain unique values. A primary key column cannot contain NULL values. Each table should have a primary key, and ***each table can have only ONE primary key***.

CREATE TABLE Persons ( P\_Id number(3) NOT NULL PRIMARY KEY, LastName varchar(255) NOT NULL, FirstName varchar(255), Address varchar(255), City varchar(255) ) ;

**Foreign Key**

A FOREIGN KEY in one table points to a PRIMARY KEY in another table. Let's illustrate the foreign key with an example. Look at the following two tables:

The "Persons" table:

P\_Id LastName FirstName Address City

1 Hansen Ola Timoteivn 10 Sandnes

2 Svendson Tove Borgvn 23 Sandnes

3 Pettersen Kari Storgt 20 Stavanger

The "Orders" table:

O\_Id OrderNo P\_Id

1 77895 3

2 44678 3

3 22456 2

4 24562 1

The Foreign Key constraint is used to prevent actions that would destroy links between tables. The Foreign Key constraint also prevents that invalid data form being inserted into the foreign key column, because it has to be one of the values contained in the table it points to.

CREATE TABLE Orders ( O\_Id number(4) NOT NULL PRIMARY KEY, OrderNo number(3) NOT NULL, P\_Id number(3) FOREIGN KEY REFERENCES Persons(P\_Id) ) ;

**CHECK Constraint**

The CHECK constraint is used to limit the value range that can be placed in a column. If you define a CHECK constraint on a single column it allows only certain values for this column. If you define a CHECK constraint on a table it can limit the values in certain columns based on values in other columns in the row.

CREATE TABLE Persons ( P\_Id number(3) NOT NULL CHECK (P\_Id>0), LastName varchar2(15) NOT NULL,FirstName varchar2(15), Address varchar2(15), City varchar2(15)) ;

**DEFAULT Constraint**

The DEFAULT constraint is used to insert a default value into a column. The default value will be added to all new records, if no other value is specified. The following SQL creates a DEFAULT constraint on the "City" column when the "Persons" table is created:

CREATE TABLE Persons ( P\_Id number(3) NOT NULL, LastName varchar(255) NOT NULL, FirstName varchar(255), Address varchar(255), City varchar(255) DEFAULT 'KURNOOL' ) ;

**DML (Data Manipulation Language) Commands**

The data manipulation language is used to add, update, and delete data in the database. The SQL commands

**INSERT** is used to add data into the database

**UPDAT**E is used to modify the data in the database,

**DELETE** is used to delete data in the database,

**SELECT** is used to describe the data from the database .

Here the term database refers to the table.

Adding a New Row to the Table

The INSERT command is to add new row to the table.

INSERT INTO table name VALUES (‗column1-name‘, ‗column2-name‘. . . columnN-name);

Ex: Insert into persons(1234,‘RAO‘,‘RAMANA‘,‘KURNOOL‘);

Inserting specified column values to a table

INSERT INTO table name(COL1,COL2,…) VALUES (‗column1-name‘, ‗column2-name‘. . . columnN-name);

Ex: Insert into persons(person\_id,last\_name,addr) values (1234,‘RAMANA‘,‘KURNOOL‘);

**Inserting rows at run time**

INSERT INTO table name VALUES (‗&column1-name‘, ‗&column2-name‘. . . &columnN-name);

**NOTE**: Here & is used to accept values at runtime **Character, date type** values has to represent in ‘ ‘ (single quotes)

Ex: Insert into persons(&person\_id,‘&last\_name‘,‘&first\_name‘,‘&addr‘);

**UPDATE Command**

The data in the table can be updated by using UPDATE command. The syntax of the UPDATE command is:

UPDATE table name SET attribute value=new value WHERE condition;

Ex: To change the addr of all the persons to NANDYAL

UPDATE persons set addr=‘NANDYAL‘;

Ex: To change first\_name of 1234 from ‗RAMANA‘ TO ‗RAM‘

UPDATE persons set FIRST\_NAME=‘RAM‘ WHERE PERSON\_ID=1234;

**DELETE Command**

The DELETE command in SQL is used to delete row(s) from the table. The syntax of DELETE command is

DELETE FROM table name WHERE condition;

Ex: TO delete all rows from a table

DELETE FROM PERSONS;

Ex: to delete a particular row

DELETE from PERSONS where person\_id=1234;

**SELECT Statement**

The SELECT command is the most commonly used command in SQL. It allows database users to retrieve the specific information they desire from an operational database.

**Syntax**

SELECT [ALL | DISTINCT] select\_list [INTO [new\_table\_name]] [FROM table\_name [, table\_name2][...,

table\_name16]]] [WHERE clause] [GROUP BY clause] [HAVING clause] [ORDER BY clause];

**ALL**: Retrieves all rows in the results. ALL is the default.

**DISTINCT**: Includes only unique rows in the results. Null values are considered equal for the purposes of the DISTINCT keyword; only one NULL is selected no matter how many are encountered.

**Select list**: Specifies the columns to select. Can be one or more of the following:

**Asterisk (\*)** representing all columns listed in the order in which they were specified in the CREATE TABLE statement for all tables in the FROM clause, in the order they appear.

- A list of column names, specified in the order in which you want to see them. If the select\_list contains multiple column names, separate the names with commas.

The command shown below retrieves all of the information contained within the EMP table. Note that the asterisk is used as a wildcard in SQL.

SELECT \* FROM EMP;

Alternatively, users may want to limit the attributes that are retrieved from the database.

For example, the Human Resources department may require a list of the last names of all employees in the company. The following SQL command would retrieve only that information:

SELECT last\_name FROM personal\_info;

**WHERE Clause:**

WHERE clause is used to extract only those records that fulfill a specified criterion.

SELECT \* FROM Persons WHERE City='KURNOOL';

**AND & OR Operators**

The AND operator displays a record if both the first condition and the second condition is true.

The OR operator displays a record if either the first condition or the second condition is true.

AND Ex: SELECT \* FROM Persons WHERE FirstName='ANAND' AND LastName='SAI' ;

OR Ex: SELECT \* FROM Persons WHERE FirstName='RAVI' OR FirstName='RAM' ;

**Transaction Control Language (TCL)**

TCL is a computer language and a subset of SQL, used to control transactional processing in a database. A transaction is logical unit of work that comprises one or more SQL statements, usually a group of DML statements.

TCL commands include:

**▪ COMMIT** This command is used to end a transaction. Only with the help of the commit command, transaction changes can be made permanent to the database. and also apply the transaction by saving the database changes.

**▪ ROLLBACK** this command is used to undo all changes of a transaction. we can either rollback the entire transaction so that all changes made by SQL statements are undone.

**▪ SAVE POINT** save points are like markers to divide a very lengthy transaction to smaller ones, They are used to identify a point in a transaction to which we can later rollback.

The general format is to issue a BEGIN WORK statement, one or more SQL statements, and then the COMMIT statement. Alternatively, a ROLLBACK statement can be issued, which undoes all the work performed since BEGIN WORK was issued. In terms of transactions, the opposite of commit is to discard the tentative changes of a transaction, a rollback.

**ADVANCED SELECT QUERIES**

One of the most important advantages of SQL is its ability to produce complex free-form queries. SQL provides useful functions that count, find minimum and maximum values, calculate averages, and so on.

Better yet, SQL allows the user to limit queries to only those entries that have no duplicates or entries whose duplicates can be grouped.

**ORDER BY**

The ORDER BY keyword is used to sort the result-set by a specified column. The ORDER BY keyword sorts the records in ascending order by default. If you want to sort the records in a descending order, you can use the **DESC** keyword.

SELECT column\_name(s) FROM table\_name ORDER BY column\_name(s) ASC|DESC;

EX1: SELECT \* FROM Persons ORDER BY LastName;

EX 2: SELECT \* FROM Persons ORDER BY LastName DESC;

Multilevel ordered sequence is known as a cascading order sequence, and it can be created easily by listing several attributes, separated by commas, after the ORDER By clause.

**DISTINCT clause**

Sometimes you will want to list only the different (distinct) values in a table. The DISTINCT keyword can be used to return only distinct (different) values.

Syntax: SELECT DISTINCT column\_name(s) FROM table name ;

Ex: SELECT DISTINCT City FROM Persons;

**SQL Aggregate Functions**

SQL Aggregate Function is built-in functions for counting and calculations (perform a calculation on a set of values and return a single value).

Syntax SELECT function (column) FROM table

**AVG** - Average value of columns

Select AVG (Sal) FROM EMP;

**COUNT** - number of rows

Select COUNT (\*) FROM EMP;

SELECT COUNT (DISTINCT Sal) FROM CUSTOMERS ;

**MAX** - Maximum or Highest number in a column

SELECT MAX (Salary) FROM CUSTOMERS ;

**MIN -** Minimum or Lowest number in a column

SELECT MIN (Salary) FROM CUSTOMERS ;

**SUM** - Total number in a column

SELECT SUM (Salary) FROM CUSTOMERS ;

**Group by clause**

GROUP BY clause can be used in a SELECT statement to collect data across multiple records and group the results by one or more columns.

SELECT Column1, Column2, …, Column\_N, Aggregate Function (Expression) FROM Tables WHERE <cond> GROUP BY <Column1, Column2, ...> HAVING <COND>;

Aggregate function can be a function such as SUM, COUNT, MIN, or MAX.

Group By Having will group values that have a particular value. This can be used in conjunction with other logical functions such as MIN, MAX, COUNT, and SUM.

The **HAVING clause** filters rows after the grouping with the Oracle GROUP BY clause. Oracle GROUP BY HAVING can be used to limit the returned rows after the grouping.

Ex: SUM function to return the name of the department and the total sales

SELECT department, SUM (sales) as "Total sales" FROM order details GROUP BY department;

Ex2: Select job, sum(sal) from emp group by job having job in(‗SALESMAN‘,‘CLERK‘);

**Virtual Tables: Creating a View**

A view consists of a stored query accessible as a virtual table composed of the result set of a query. Unlike ordinary tables (base tables) in a relational database, a view does not form part of the physical schema: it is a dynamic, virtual table computed or collated from data in the database.

Views can provide **advantages** over tables:

▪ Views can represent a subset of the data contained in a table

▪ Views can join and simplify multiple tables into a single virtual table

▪ Views can act as aggregated tables, where the database engine aggregates data (sum, average etc.) and presents the

calculated results as part of the data

▪ Views can hide the complexity of data;

For example a view could appear as Sales2000 or Sales2001, transparently partitioning the actual underlying table

▪ Depending on the SQL engine used, views can provide extra security

▪ Views can limit the degree of exposure of a table or tables to the outer world

**Syntax for creating a VIEW is:**

CREATE VIEW view\_name AS SELECT columns FROM table WHERE condition;

Ex: Create View Emp\_View As Select Empno,Ename,Job,Sal From Emp Where Sal>2000 And Hiredate Like ‗%80‘;

This would create a virtual table based on the result set of the select statement. You can now query the view as follows:

SELECT \* FROM EMP\_view;

**Joining Database Tables(Joins)**

The ability to combine (Join) tables on common attributes is perhaps the most important distinction between a relational database and other databases. A join is performed when data are retrieved from more than one table at a time.

To join tables, you simply list the tables in the FROM clause of the SELECT statement. The DBMS will create the Cartesian product of every table in the FROM clause. However, to get the correct result, you must select only the rows in which the common attribute values match. We use the WHERE clause to indicate the common attributes used to link the tables.

The join condition is generally composed of an equality comparison between the foreign key and the primary key of related tables.

**For example,** suppose you want to join the two tables EMP and DEPT. Because DEPTNO is the foreign key in the EMP table and the primary key in the DEPT table, the link is established on DEPTNO.

To join the EMP and DEPT tables, you would use the following,

SELECT EMPNO,ENAME,JOB,SAL,DNAME,LOC FROM EMP,DEPT WHERE EMP.DEPTNO=DEPT.DEPTNO;

When joining three or more tables, you need to specify a join condition for each pair of tables. The number of join conditions will always be N-1, where N represents the number of tables listed in the FROM clause.

For example, if you have three tables, you must have two join conditions; if you have five tables, you must have four join conditions; and so on.

**Joining Tables with an Alias**

An alias may be used to identify the source table from which the data are taken. The aliases E and D are used to label the EMP and DEPT tables in the next command sequence. Any legal table name may be used as an alias.

SELECT E.EMPNO, E.ENAME, E.JOB, E.DEPTNO FROM EMP E, DEPT D where E.DEPTNO=D.DEPTNO;

**Recursive Joins**

An alias is especially useful when a table must be joined to itself in a recursive query.

For example, suppose you are working with the EMP table. Using the data in the EMP table, you can generate a list of all employees with their managers' names by joining the EMP table to itself.

The SQL command sequence would look like this:

SELECT E.MGR, M.ENAME, E.EMPNO, E.ENAME FROM EMP E, EMP M WHERE E.MGR=M.EMPNO;

**Set Operator**

UNION All rows selected by either query.

UNION ALL All rows selected by either query, including all duplicates.

INTERSECT All distinct rows selected by both queries.

**Outer join**

An outer join does not require each record in the two joined tables to have a matching record. The joined table retains each record—even if no other matching record exists. Outer joins subdivide further into left outer joins, right outer joins, and full outer joins, depending on which table(s) one retains the rows from (left, right, or both).

**Left outer join**

Left outer join returns all the values from the left table, plus matched values from the right table If the right table returns one row and the left table returns more than one matching row for it, the values in the right table will be repeated for each distinct row on the left table.

For example, this allows us to find an employee's department, but still shows the employee(s) even when they have not been assigned to a department

SELECT \* FROM emp, dept WHERE emp.DeptNO = dept.DeptNO (+);

**Right outer join**

A right outer join (or right join) closely resembles a left outer join, except with the treatment of the tables reversed. Every row from the "right" table (B) will appear in the joined table at least once. If no matching row from the "left" table (A) exists, NULL will appear in columns from A for those records that have no match in B.

A right outer join returns all the values from the right table and matched values from the left table (NULL in case of no matching join predicate).

For example, this allows us to find each employee and his or her department, but still show departments that have no employees. Below is shown an example of right outer join, with the additional result row italicized:

SELECT \* FROM EMP RIGHT OUTER JOIN dept ON emp.DeptNO = dept.DeptNO;

**Relational Set Operators**

Set operators combine the results of two component queries into a single result. Queries containing set operators are called compound queries

**MINUS All** distinct rows selected by the first query but not the second.

All set operators have equal precedence. If a SQL statement contains multiple set operators, Oracle

evaluates them from the left to right if no parentheses explicitly specify another order.

**UNION** The following statement combines the results with the UNION operator, which eliminates duplicate selected rows. This statement shows how data type must match when columns do not exist in one or the other table:

SELECT part, partnum, to\_date(null) date\_in FROM orders\_list1 UNION SELECT part, to\_date (null), date\_in FROM orders\_list2;

Ex2: SELECT part FROM orders\_list1 UNION SELECT part FROM orders\_list2;

**UNION ALL** : The following statement combines the results with the UNION ALL operator, which does not eliminate duplicate selected rows:

SELECT part FROM orders\_list1 UNION ALL SELECT part FROM orders\_list2;

**Note** that the UNION operator returns only distinct rows that appear in either result, while the UNION ALL operator returns all rows. A PART value that appears multiple times in either or both queries (such as 'FUEL PUMP') is returned only once by the UNION operator, but multiple times by the UNION ALL operator.

**INTERSECT**

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

SELECT part FROM orders\_list1 INTERSECT SELECT part FROM orders\_list2;

**MINUS**

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

SELECT part FROM orders\_list1 MINUS SELECT part FROM orders\_list2;

**Sub Queries**

A sub query is a query within a query. In Oracle, you can create subqueries within your SQL statements. These subqueries can reside in the WHERE clause, the FROM clause, or the SELECT clause.

**WHERE clause**

Most often, the sub query will be found in the WHERE clause. These sub queries are also called nested sub queries.

For example:

Select \* from all\_tables tabs where tabs.table\_name IN (Select cols.table\_name from

all\_tab\_columns cols where cols.column\_name = 'SUPPLIER\_ID');

**NOTE**: Oracle allows up to 255 levels of sub queries in the WHERE clause.

FROM clause

A subquery can also be found in the FROM clause. These are called inline views.

For example:

Select suppliers.name, subquery1.total\_amt from suppliers, (select supplier\_id, Sum (orders. amount) as total\_amt from orders group by supplier\_id) subquery1, where subquery1.supplier\_id = suppliers.supplier\_id;

In this example, we've created a sub query in the FROM clause as follows:

(select supplier\_id, Sum(orders.amount) as total\_amt from orders group by supplier\_id) subquery1

This sub query has been aliased with the name subquery1. This will be the name used to reference this sub query or any of its fields.

**NOTE**: Oracle allows an unlimited number of sub queries in the FROM clause.

**SQL Functions**

SQL functions are similar to SQL operators in that both manipulate data items and both return a result. SQL functions differ from SQL operators in the format in which they appear with their arguments. The SQL function format enables functions to operate with zero, one, or more arguments.

SQL functions are used exclusively with SQL commands within SQL statements. There are two general types of SQL functions: single row (or scalar) functions and aggregate functions. These two types differ in the number of database rows on which they act. A single row function returns a value based on a single row in a query, whereas an aggregate function returns a value based on all the rows in a query.

What is a **DUAL** Table in Oracle?

This is a single row and single column dummy table provided by oracle. This is used to perform mathematical calculations without using a table.

Select \* from DUAL;

**1.Numeric Functions:**

Numeric functions are used to perform operations on numbers. They accept numeric values as input and return numeric values as output. Few of the Numeric functions are:

|  |  |
| --- | --- |
| Function Name | Return Value |
| ABS (x) | Absolute value of the number '*x*' |
| CEIL (x) | Integer value that is Greater than or equal to the number '*x*' |
| FLOOR (x) | Integer value that is Less than or equal to the number '*x*' |
| TRUNC (x, y) | Truncates value of number '*x*' up to '*y*' decimal places |
| ROUND (x, y) | Rounded off value of the number '*x*' up to the number '*y*' decimal places |

The following examples explains the usage of the above numeric functions

|  |  |  |
| --- | --- | --- |
| **Function Name** | **Examples** | **Return Value** |
| ABS (x) | ABS (1) ABS (-1) | 1 -1 |
| CEIL (x) | CEIL (2.83) CEIL (2.49) CEIL (-1.6) | 3 3 -1 |
| FLOOR (x) | FLOOR (2.83) FLOOR (2.49) FLOOR (-1.6) | 2 2 -2 |
| TRUNC (x, y) | ROUND (125.456, 1) ROUND (125.456, 0) ROUND (124.456, -1) | 125.4 125 120 |
| ROUND (x, y) | TRUNC (140.234, 2) TRUNC (-54, 1) TRUNC (5.7) TRUNC (142, -1) | 140.23 54 5 140 |

These functions can be used on database columns.

For Example: Let's consider the product table used in sql joins. We can use ROUND to round off the unit\_price to the nearest integer, if any product has prices in fraction.

SELECT ROUND (unit\_price) FROM product;

**2) Character or Text Functions:**

Character or text functions are used to manipulate text strings. They accept strings or characters as input and can return both character and number values as output.

Few of the character or text functions are as given below:

|  |  |
| --- | --- |
| **Function Name** | **Return Value** |
| LOWER (string\_value) | All the letters in *'string\_value'* is converted to lowercase. |
| UPPER (string\_value) | All the letters in *'string\_value'* is converted to uppercase. |
| INITCAP (string\_value) | All the letters in *'string\_value'* is converted to mixed case. |
| LTRIM (string\_value, trim\_text) | All occurrences of *'trim\_text'* is removed from the left of *'string\_value'*. |
| RTRIM (string\_value, trim\_text) | All occurrences of *'trim\_text'* is removed from the right of *'string\_value'* . |
| TRIM (trim\_text FROM string\_value) | All occurrences of *'trim\_text'* from the left and right of *'string\_value'* , *'trim\_text'* can also be only one character long . |
| SUBSTR (string\_value, m, n) | Returns *'n'* number of characters from *'string\_value'* starting from the '*m'* position. |
| LENGTH (string\_value) | Number of characters in *'string\_value'* in returned. |
| LPAD (string\_value, n, pad\_value) | Returns '*string\_value'* left-padded with *'pad\_value'* . The length of the whole string will be of *'n'*characters. |
| RPAD (string\_value, n, pad\_value) | Returns '*string\_value'* right-padded with *'pad\_value'* . The length of the whole string will be of *'n'*characters. |

For Example, we can use the above UPPER() text function with the column value as follows.

SELECT UPPER (product\_name) FROM product;

The following examples explains the usage of the above character or text functions

|  |  |  |
| --- | --- | --- |
| **Function Name** | **Examples** | **Return Value** |
| LOWER(string\_value) | LOWER('Good Morning') | good morning |
| UPPER(string\_value) | UPPER('Good Morning') | GOOD MORNING |
| INITCAP(string\_value) | INITCAP('GOOD MORNING') | Good Morning |
| LTRIM(string\_value, trim\_text) | LTRIM ('Good Morning', 'Good) | Morning |
| RTRIM (string\_value, trim\_text) | RTRIM ('Good Morning', ' Morning') | Good |
| TRIM (trim\_text FROM string\_value) | TRIM ('o' FROM 'Good Morning') | Gd Mrning |
| SUBSTR (string\_value, m, n) | SUBSTR ('Good Morning', 6, 7) | Morning |
| LENGTH (string\_value) | LENGTH ('Good Morning') | 12 |
| LPAD (string\_value, n, pad\_value) | LPAD ('Good', 6, '\*') | \*\*Good |
| RPAD (string\_value, n, pad\_value) | RPAD ('Good', 6, '\*') | Good\*\* |

**3) Date Functions:**

These are functions that take values that are of datatype DATE as input and return values of datatypes DATE, except for the MONTHS\_BETWEEN function, which returns a number as output.

**Sysdate: to Represent system date**

Few date functions are as given below.

|  |  |
| --- | --- |
| **Function Name** | **Return Value** |
| ADD\_MONTHS (date, n) | Returns a date value after adding *'n'* months to the date *'x'*. |
| MONTHS\_BETWEEN (x1, x2) | Returns the number of months between dates x1 and x2. |
| ROUND (x, date\_format) | Returns the date *'x'* rounded off to the nearest century, year, month, date, hour, minute, or second as specified by the *'date\_format'*. |
| TRUNC (x, date\_format) | Returns the date *'x'* lesser than or equal to the nearest century, year, month, date, hour, minute, or second as specified by the 'date\_format'. |
| NEXT\_DAY (x, week\_day) | Returns the next date of the *'week\_day'* on or after the date *'x'* occurs. |
| LAST\_DAY (x) | It is used to determine the number of days remaining in a month from the date *'x'* specified. |
| SYSDATE | Returns the systems current date and time. |
| NEW\_TIME (x, zone1, zone2) | Returns the date and time in zone2 if date 'x' represents the time in zone1. |

The below table provides the examples for the above functions

|  |  |  |
| --- | --- | --- |
| **Function Name** | **Examples** | **Return Value** |
| ADD\_MONTHS ( ) | ADD\_MONTHS ('16-Sep-81', 3) | 16-Dec-81 |
| MONTHS\_BETWEEN( ) | MONTHS\_BETWEEN ('16-Sep-81', '16-Dec-81') | 3 |
| NEXT\_DAY( ) | NEXT\_DAY ('01-Jun-08', 'Wednesday') | 04-JUN-08 |
| LAST\_DAY( ) | LAST\_DAY ('01-Jun-08') | 30-Jun-08 |
| NEW\_TIME( ) | NEW\_TIME ('01-Jun-08', 'IST', 'EST') | 31-May-08 |

**4) Conversion Functions:**

These are functions that help us to convert a value in one form to another form. For Ex: a null value into an actual value, or a value from one datatype to another datatype like NVL, TO\_CHAR, TO\_NUMBER, TO\_DATE.

Few of the conversion functions available in oracle are:

|  |  |
| --- | --- |
| **Function Name** | **Return Value** |
| TO\_CHAR (x [,y]) | Converts Numeric and Date values to a character string value. It cannot be used for calculations since it is a string value. |
| TO\_DATE (x [, date\_format]) | Converts a valid Numeric and Character values to a Date value. Date is formatted to the format specified by *'date\_format'*. |
| NVL (x, y) | If *'x'* is NULL, replace it with *'y'*. *'x'* and *'y'* must be of the same datatype. |

The below table provides the examples for the above functions

|  |  |  |
| --- | --- | --- |
| **Function Name** | **Examples** | **Return Value** |
| TO\_CHAR () | TO\_CHAR (3000, '$9999') TO\_CHAR (SYSDATE, 'Day, Month YYYY') | $3000 Monday, June 2008 |
| TO\_DATE () | TO\_DATE ('01-Jun-08') | 01-Jun-08 |
| NVL () | NVL (null, 1) | 1 |