Bachelor of Computer Applications (BCA)

MySQL (SQL/PL-SQL) (DBCASE305T24)

Self-Learning Material
(SEM III)

Jaipur National University
Centre for Distance and Online Education

Established by Government of Rajasthan
Approved by UGC under Sec 2(f) of UGC ACT 1956
&
NAAC A+ Accredited

Jaipur National University

Course Code: DBCASE305T24
MySQL (SQL/PL-SQL)

TABLE OF CONTENTS

COURSE INTRODUCTION

Welcome to the "MySQL (SQL/PL-SQL)" course, an in-depth exploration into the world of relational database management systems with a focus on MySQL and its associated query languages. This course is designed to provide you with a comprehensive understanding of SQL and PL/SQL, essential tools for managing and manipulating databases effectively.

This course has 3 credits and is divided into 11 Units. In this course, you will delve into the core concepts of MySQL, one of the most widely used open-source database systems. You will learn the foundational principles of SQL (Structured Query Language), which is used for querying and managing relational databases. Through practical exercises and real-world examples, you will become proficient in writing and optimizing SQL queries to retrieve, insert, update, and delete data.

Additionally, the course introduces PL/SQL (Procedural Language/SQL), an extension of SQL that adds procedural programming capabilities to SQL. You will gain hands-on experience with creating stored procedures, functions, and triggers, which allow for more complex data operations and automation within the database. This knowledge will enable you to develop robust and efficient database applications.

Throughout the course, you will engage with various tools and techniques to manage and manipulate databases, including creating and maintaining database schemas, defining relationships between tables, and ensuring data integrity. The practical approach of the course will equip you with the skills to handle real-life database challenges and optimize performance.

By the end of the course, you will have developed a solid understanding of both SQL and PL/SQL, empowering you to design and manage databases effectively. Whether you are aiming to advance your career in database management, software development, or data analysis, this course will provide you with the essential skills and knowledge to succeed in the field of database technology.

Course Outcomes:

At the completion of the course, a student will be able to:

1. Remember key concepts related to SQL including DDL, DML, DCL, DTL commands.
2. Understanding of PL/SQL elements like Cursors, Procedures, functions, triggers.
3. Applying cursors, procedures, functions and triggers on the student database to perform different updating and manipulations in existing tables in the database. Use of stored procedures, functions, cursors to ensure max reusability.
4. Analyze the limitations of SQL and supports provided by procedural language to develop an effective application.
5. Built a strong adherence to procedural language while creating applications.

Acknowledgements:

The content we have utilized is solely educational in nature. The copyright proprietors of the materials reproduced in this book have been tracked down as much as possible. The editors apologize for any violation that may have happened, and they will be happy to rectify any such material in later versions of this book.

Chapter 1:
Introduction to SQL

Learning Outcomes:

- Students will be able to understand basics and purpose of SQL
- Students will be able to understand about Data and its types
- Students will be able to relationship of SQL with Relational Database
- Students will be able to types of SQL statements with their commands set
- Students will be able to apply different Constraints on Data

Structure:

1.1 Introduction SQL

SQL, which stands for “Structured Query Language”, is a specialized programming language designed for managing and manipulating relational databases. It offers a standardized method for interacting with databases, enabling users to perform a variety of tasks. These tasks include querying data, inserting or updating records, and defining the structures of databases.

Key features and concepts of SQL:

1. Data Querying: SQL allows users to retrieve specific data from databases using "SELECT" statement. Queries can be tailored to filter, sort, and aggregate data based on specified criteria.

2. Data Manipulation: SQL provides commands like "INSERT", "UPDATE", and "DELETE" to modify records within database tables. These commands ensure efficient management and maintenance of data.

3. Database Schema Definition: SQL includes statements like "CREATE TABLE", "ALTER TABLE", and "DROP TABLE" to define and manage a structure of database tables. Users can specify column names, data types, constraints (e.g., "PRIMARY KEY", "FOREIGN KEY", "CHECK"), and other properties.

4. Data Control: SQL statements such as "GRANT" and "REVOKE" manage permissions and security, allowing administrators to control access to the database objects based on user roles and privileges.

SQL is widely used across various "database management systems (DBMS)" such as "Oracle, MySQL, PostgreSQL, SQL Server, and SQLite". While specific implementations may vary slightly, the core SQL syntax and functionality remain consistent, making it a versatile tool for database professionals and developers.

1.2 Query

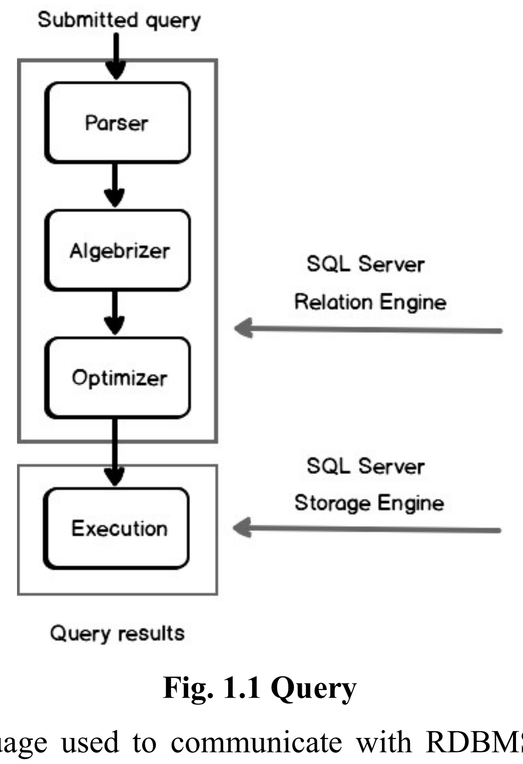


Fig. 1.1 Query

SQL is a declarative language used to communicate with RDBMS. Execution of a query is three step process.

1. Parsing
2. Optimization
3. Execution

1. Parsing-When a user writes query that query hits the server process and server process bring the query in into shared pool(SGA).The Server process generates Hash which is nothing but hexadecimal value for SQL statement text .If this query is previously executed then SQL-Area exists otherwise it is created. Parsing is made up of three parts

- **Syntax check-** SQL engine checks the syntax of the SQL statement. A Parse tree is generated.
- **Semantics check-** The parsed tree is Input to the Algebriser which performs the semantics check after this step the statement is not available in previous form. Query processor tree is output of algebrizer. During semantic analysis whether the object say table and the column exists or not is checked.
- **Privileges check -**whether the user has rights over the object that is table or on the columns is checked.

2. Optimization-Optimizer is the software which is used to generate different scripts for execution of a query. The job of optimizer is to find best option to retrieve result based on cost and time. Query processor tree act as input for optimizer, it reads the relevant blocks data blocks in which are brought back into buffer cache. Server process is responsible for doing this.

3. Execution of a Query/Fetching- The Relational Engine requests the storage engine for the data and result is delivered to the user.

1.3 Definitions of Data

The word data comes from the Latin word datum which means “something given”. Datum is a singular form of data.

- **D**ata is defined as, collection of facts and figures.
- Data is a collection of unorganized facts and figures that serves as a base for reasoning or calculation when processed and converted into information.

- Data can be a character or a symbol or digits on which operation can be performed by computer and it can be stored and transmitted.
  Example -List of phone numbers of student
- Data is describing units of observation like phone numbers. We can collect name, age, date of birth and weight data about a person.
- Data when processed is converted into information which is used for decision making.

1.4 Types of SQL Commands

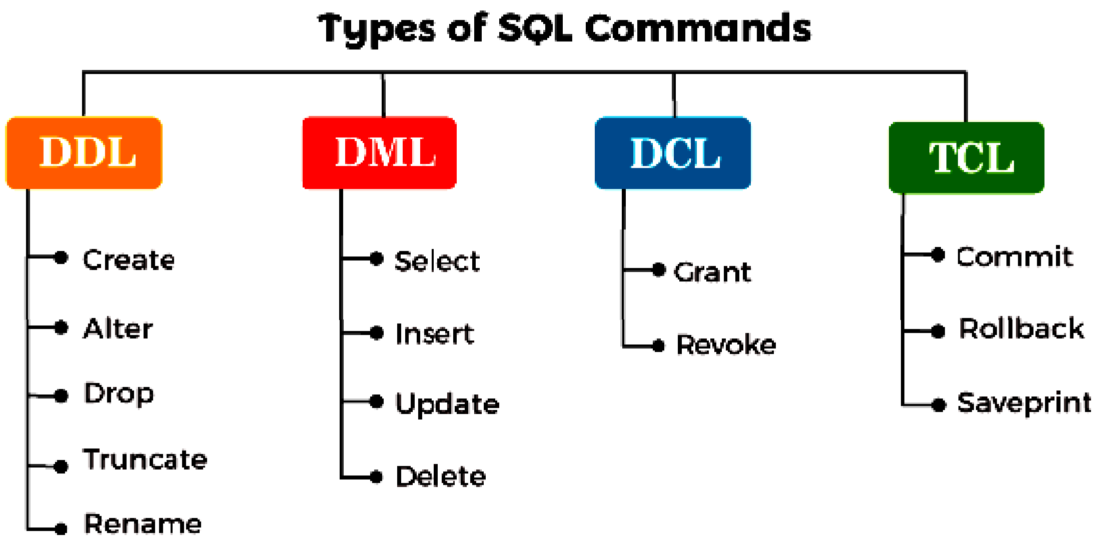


Fig. 1.2. Types of SQL Commands

There are four types of SQL Commands

a) “Data Definition Language”

b) “Data Manipulation Language”

c) “Data Control Language”

d) “Transaction Control Language”

1.5 SQL Data Types

The kinds of data that can be kept in each column of a database table are specified by SQL data types. These data kinds enhance data storage and retrieval while assisting in maintaining data integrity. An overview of the most often used SQL data types is provided below:

a) “**Numeric Data Types”:** Numeric data types are used to store numerical values.

- **“INT”:** This data type stores whole numbers without decimal points. It is typically used for counting or indexing purposes.
  Ex: age INT;
- **“DECIMAL(p, s):”** With "p" denoting the total number of digits and "s" denoting the number of digits following the decimal point, this data type contains fixed-point integers. For exact numerical numbers, like money amounts, it is helpful.
  Ex: salary DECIMAL(10, 2);
- **FLOAT:** This data type stores floating-point numbers, which are numbers that contain decimal points. It is used when precision is less critical, such as for scientific calculations.
  Ex: temperature FLOAT;

b) **“Character/String Data Types”:** Character data types store text strings.

- **“CHAR(n)”:** This data type stores fixed-length strings. The length “n” is specified when the table is created. It is used when the exact length of the string is known.
  Ex: country_code CHAR(2);
- **“VARCHAR(n)”:** This data type stores variable-length strings, where “n” defines the maximum length of the string. It is used when the length of string can vary.
  Ex: last_name VARCHAR(50);

c) **“Date and Time Data Types”:** Date and time data types store date and time values.

- **“DATE”:** This data type stores date values, typically formatted as “YYYY-MM-DD”. It is used to store dates without time information.
  Ex: birth_date DATE;
- **“TIME”:** This data type stores time values, typically formatted as “HH:MM:SS”. It is used to store time without date information.
  Ex: start_time TIME;
- **“DATETIME”:** This data type stores date and time values together, formatted as “YYYY-MM-DD HH:MM:SS”. It is used to store events that include both date and time.

Ex: appointment DATETIME;

- **"TIMESTAMP":** This data type stores date and time values and often includes automatic updating to the current date and time when a record is modified. It is useful for tracking changes.
  Ex: last_updated TIMESTAMP;

d) **"Binary Data Types":** Binary data types store binary data.

- **"BINARY(n)":** This data type stores fixed-length binary data, where "n" specifies the number of bytes.
  Ex: binary_data BINARY(16);
- **"VARBINARY(n)":** This data type stores variable-length binary data, with "n" indicating the maximum number of bytes.
  Ex: file_data VARBINARY(255);

e) **"Other Data Types":** Other specialized data types include:

- **"BOOLEAN":** This data type stores TRUE or FALSE values. It is used for boolean logic operations.
  Ex: is_active BOOLEAN;
- **"BLOB":** This data type stands for "Binary Large Object" and is used to store large binary data, such as "images or multimedia files".
  Ex: image BLOB;
- **"TEXT":** This data type stores large amounts of text data. It is used for storing long text strings, such as descriptions or articles.
  Ex: description TEXT;

1.6 SQL Operators and Expressions

In SQL, operators and expressions are fundamental components used to manipulate, compare, and retrieve data within database queries. Operators perform specific actions on data, while expressions combine operators and values to yield results.

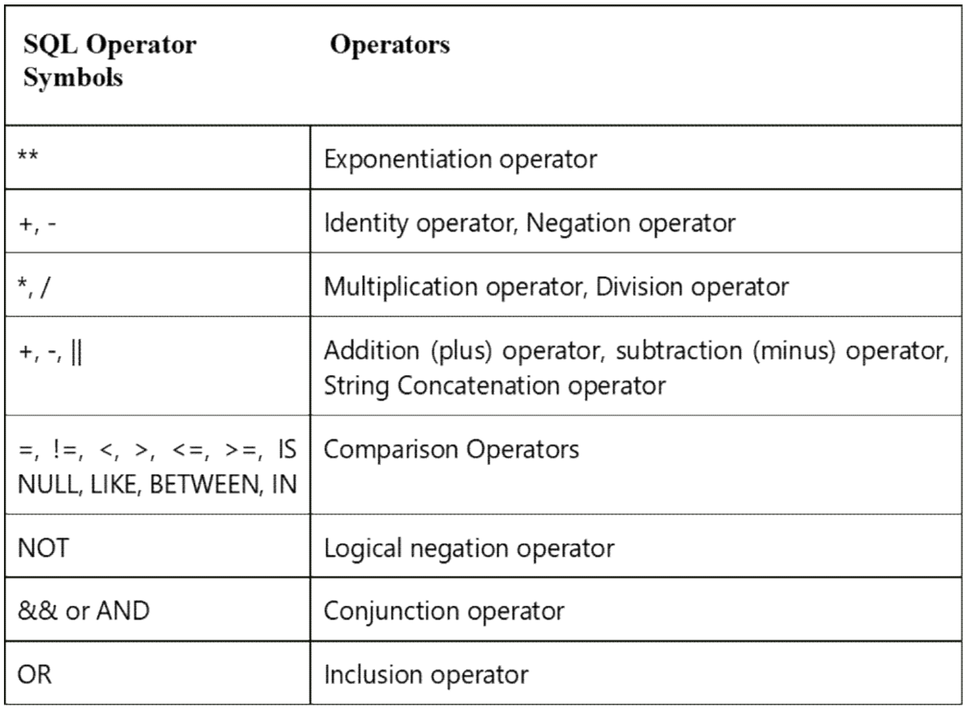
"What is SQL Operator?"

In SQL, reserved words and characters used in conjunction with a WHERE clause are known as operators. These operators can be classified into two types: “unary and binary”. A unary operator operates on a single operand to perform its operation, while a binary operator requires two operands to complete its function.

- **Syntax of Unary SQL Operator**: **“Operator SQL_Operand”**
- **Syntax of binary SQL Operator: “Operand1 SQL_Operator Operand2”**

Precedence of SQL Operator:

The precedence of SQL operators determines the order in which SQL evaluates the various operators within the same expression. Operators with higher precedence are evaluated before those with lower precedence. Understanding operator precedence ensures that expressions are calculated in the intended sequence.



SQL Operator Symbols	Operators
**	Exponentiation operator
+, -	Identity operator, Negation operator
*, /	Multiplication operator, Division operator
+, -, ||	Addition (plus) operator, subtraction (minus) operator, String Concatenation operator
=, !=, <, >, <=, >=, IS NULL, LIKE, BETWEEN, IN	Comparison Operators
NOT	Logical negation operator
&& or AND	Conjunction operator
OR	Inclusion operator

1.7 Types of Operator

SQL operators are categorized in the following categories:

1. “Arithmetic Operators”
2. “Comparison Operators”
3. “Logical Operators”
4. “Set Operators”

5. “Bit-wise Operators”
6. “Unary Operators”

The fundamental elements needed to carry out different actions on data inside a database are SQL operators. The various categories into which these operators might be divided depend on the kind of operation that they carry out.

a) “Arithmetic Operators”: Arithmetic operators are used to perform mathematical calculations on numeric data.

- “Addition (+): Adds two numbers.
- Subtraction (-): Subtracts one number from another.
- Multiplication (*): Multiplies two numbers.
- Division (/): Divides one number by another.
- Modulo (%): Returns the remainder of a division”.

b) Comparison Operators: Comparison operators are used to compare two values, and the result is typically a boolean value (true or false).

- “Equal to (=): Checks if two values are equal.
- Not equal to (<> or !=): Checks if two values are not equal.
- Greater than (>): Checks if the value on the left is greater than the value on the right.
- Less than (<): Checks if the value on the left is less than the value on the right.
- Greater than or equal to (>=): Checks if the value on the left is greater than or equal to the value on the right.
- Less than or equal to (<=): Checks if the value on the left is less than or equal to the value on the right”.

c) Logical Operators: Logical operators are used to combine multiple conditions in a SQL statement.

- “AND: Returns true if all conditions are true.
- OR: Returns true if at least one condition is true.
- NOT: Reverses the result of a condition”.

d) Bitwise Operators: On binary data, bitwise operators carry out bit-level operations.

- “Bitwise AND (&): Performs a bitwise AND operation.
- Bitwise OR (|): Performs a bitwise OR operation.
- Bitwise XOR (^): Performs a bitwise XOR operation.
- Bitwise NOT (~): Performs a bitwise NOT operation.
- Left Shift (<<): Shifts bits to the left.
- Right Shift (>>): Shifts bits to the right”.

e) Other Operators:

- “BETWEEN: Checks if a value is within a specified range.
- IN: Checks if a value matches any value in a list.
- LIKE: Searches for a specified pattern in a column.
- IS NULL / IS NOT NULL: Checks for null values”.

1.8 SQL Expressions

Combinations of values, operators, and functions that evaluate to a value are called expressions in SQL. They are essential to the manipulation and querying of databases. Here's a breakdown of different types of SQL expressions:

1. Scalar Expressions:

Scalar expressions yield a single value and can include columns, literals, operators, and functions.

- **Column Values:** Refer to specific column values.
  Ex: “SELECT salary FROM employees;”
- **Arithmetic Operations:** Perform calculations with arithmetic operators.
  Ex: “SELECT salary * 1.10 AS increased_salary FROM employees;”
- **String Operations:** Manipulate text values using string functions and operators.
  Ex: “SELECT CONCAT(first_name, ' ', last_name) AS full_name FROM employees;”

2. Boolean Expressions:

Boolean expressions evaluate to true or false and are commonly used in the WHERE clause for filtering.

- **Comparison Operators**: Compare values.

  Ex: "SELECT * FROM employees WHERE salary > 50000;"
- **Logical Operators:** Combine conditions using AND, OR, and NOT.

  Ex: "SELECT * FROM employees WHERE salary > 50000 AND position = 'Manager';"

3. String Expressions:

String expressions deal with operations on text data, such as concatenation and case conversion.

- **Concatenation**: Combine strings.

  Ex: "SELECT first_name || ' ' || last_name AS full_name FROM employees;"
- **String Functions:** Manipulate strings using functions like UPPER and LOWER.

  Ex: "SELECT UPPER(first_name) AS upper_name FROM employees;"

4. Date and Time Expressions:

Date and time expressions involve operations on date and time data.

- **Current Date and Time:** Get current date and time.

  Ex: "SELECT CURRENT_DATE;"
- **Date Arithmetic:** Perform calculations with dates.

  Ex: "SELECT hire_date + INTERVAL '1 year' AS next_year FROM employees;"

1.9 Introduction to SQL*Plus:

PL/SQL blocks, scripts, and SQL commands can all be run with SQL*Plus, an interactive and scriptable command-line tool for Oracle databases. It offers a strong environment for tasks related to database construction, management, and reporting. This is a summary of SQL*Plus:

Features of SQL*Plus:

1. Interactive SQL Queries: Users can enter SQL commands directly into the command-line interface to query and manipulate data in the database.

2. Scripting: SQL*Plus allows users to write and execute SQL scripts, which are files containing a sequence of SQL commands and PL/SQL blocks. This facilitates automation and batch processing of database tasks.

3. Formatting Output: Users can customize the format of query results using formatting commands, such as COLUMN, BREAK, and COMPUTE, to improve readability and presentation.

4. Spooling Output: SQL*Plus enables users to save query results to files on the operating system using the SPOOL command. This is useful for generating reports and exporting data.

5. PL/SQL Support: SQL*Plus provides support for executing PL/SQL blocks, allowing users to create and execute stored procedures, functions, and triggers directly from the command line.

1.10 Summary

Data is a collection of unprocessed facts and numbers that, when processed, become information. SQL offers a large variety of data types to represent different types of data. The following categories apply to SQL statements: DL, ML, DCL, DQL, and TCL. Scriptable and interactive interfaces for querying, scripting, and reporting are provided by SQL*Plus, an adaptable tool for working with Oracle databases. Because of its features, database administrators, developers, and analysts who work with Oracle Database systems can find it to be a useful tool. Managing Oracle databases can be made much more productive and efficient by knowing how to utilize SQL*Plus properly.

1.11 References

1. Data analysis using SQL by Gordon Linoff
2. The Applied SQL Data Analyticsby Benjamin Johnston, Matt Goldwasser, and Upom Malik
3. Learning SQL by Alan Beaulieu
4. Data Analytics: Data Science for Beginners, SQL Computer Programming for Beginners

Chapter 2:
Managing Tables

Learning Outcomes:

- Students will be able to understand working of all types of SQL statements with their commands set
- Students will be able to understand different commands and their functionality.
- Students will be able to apply different Constraints on Table.

Structure

2.1 Introduction

SQL commands are categorized as,

a) “Data Definition Language” (DDL)

b) “Data Manipulation Language” (DML)

c) “Data Control Language” (DCL)

d) “Transaction Control Language” (TCL)

e) “Data Query Language” (DQL)

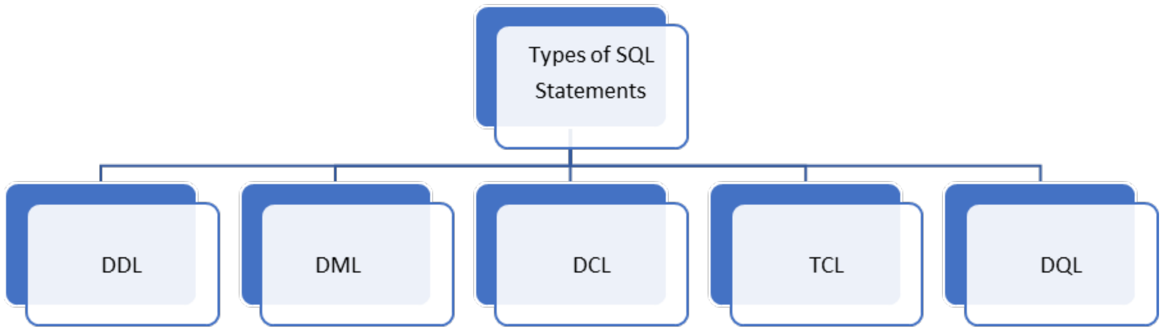


Fig.2.1 Type of SQL Statements

A variety of commands are available for executing operations on database objects depending on the type of SQL. These commands can alter an object's structure and create new ones as well as add, update, or remove data from them and remove the object from the database.

2.2 Creating Tables

Table is basic storage structure in RDBMS. Tables are two-dimensional matrix that contain number of rows and columns. CREATE TABLE command is used to create tables. This statement belongs to Data Definition Language.

If we want to create table following things should be fixed.

- Name of the Table.
- Number of columns in the table.
- Data type and the length of different columns of the table.
- Integrity constraints.

Rules for Naming the Table

- The table name must start with an alphabet .
- Maximum length for table name is 30 characters.
- Keyboards should not be used as table names .

- Table name should be unique and formed using alphabets, digits, special symbol, underscore, dollar and hash are allowed.

Syntax:

```
“CREATE TABLE table_name(
   column1 datatype (size),
   column2 datatype (size),
   column3 datatype (size),
   .
   .
  .
 columnN datatype (size),
   PRIMARY KEY( one or more columns )
);”
```

“Create Table”- This is the command verb which explains the RDBMS that user wants to create an object called table in the database, Integrity constraints can be specified in create table command.

“table_name”-Specifies the unique name of the new table to be created.

“column1”- Specifies name of first column.

“data_type”- Specifies type of data to be stored in specific column.

“size”- Defines the maximum amount of data that can be in each column of a table. A column can store a character value if, for instance, its size and data type are set to varchar and 50, respectively. The column's length will also be 50.

Example-

CREATE TABLE EMP(EMPNO NUMBER(5),ENAME VARCHAR2(50),SAL NUMBER(9,2));

2.3 Creating Tables from another Table-

A Table can be created from existing table using Create Table command with **“As Select”** option.

Syntax:

"CREATE TABLE TABLENAME(COLUMN1,COLUMN2,COLUMN3) AS SELECT (COLUMN1,COLUMN2,COLUMN3) FROM TABLE TABLENAME1;"

Example-

"CREATE TABLE NEWEMP
AS SELECT(EMPNO,ENAME,SALARY,DNO) FROM EMP;"

Here newemp with columns empno, ename, salary, dno will be created from table emp.

Describe command –Describes structure of table. This displays Name of Columns,Width, and constraints. So "**desc**" or "**describe**" command prints **structure** of table which has "**name**" of column, "**data-type**" of column and "**null**" which specify, that column can contain null values or not.

All of these features are defined at the time of **Creation** of table.

Syntax:

"DESC tablename;"

Example- DESC EMP;

2.4 Adding Columns

Alter Table command- The table's structure can be altered with this command. Using the Add option with the Alter Table command can be used to add new columns. The Modify option allows you to adjust the column's length and data type.

```
ALTER TABLE table_name
        ADD (Columnname_1  datatype(size),
        Columnname_2  datatype(size),
        …
        Columnname_n  datatype(size));
```

Example- "ALTER TABLE Stud ADD (rollno number(3),COURSE varchar(40));"

“ALTER TABLE-MODIFY”:

It is employed to change a table's current columns. It's also possible to change several columns at once. Reducing the size of a column could cause data loss.

“ALTER TABLE table_name
MODIFY column_name(size), column_type(size);”

Example- “ALTER TABLE Student MODIFY COURSE varchar(20);”

2.5 DROP COLUMN-

This is used to remove column in a table.

Syntax:

“ALTER TABLE table_name
DROP COLUMN column_name;”

Example –“ALTER TABLE Student DROP COLUMN COURSE;”

Drop table –This command is used to remove the objects from database the stucture of the table along with the data from the database is removed.

Syntax:

“DROP object object_name;
DROP TABLE table_name;”

“DROP TABLE:Command Verb
table_name: Name of the table to be deleted.”

Example-DROP TABLE EMP;

❖ **Knowledge Check 1**

State True or False

a. DDL commands are auto-committed.
b. Revoke command is used to add users.
c. Select is a command of Data Query Language.
d. Drop Table is used to remove table from the database.

Outcome Based Activity

Collect information about Truncate command.

2.6 Data Manipulation Language:

Databases can be modified using instructions in the Data Manipulation Language. It is in charge of all database modifications. DML commands do not automatically commit.

DML Commands –Following are the commands of DML category.

- UPDATE-This command modifies the data of table.
- DELETE-This command delete rows from table.
- INSERT-It adds a row in the existing table.

INSERT Command:

Syntax:

```
INSERT INTO <table_name>[(column_name1 , column_name2 )]
 VALUES (value1, value2);
```

-Specifying column names are optional.

Example:

```
INSERT INTO student
VALUES ('1', 'Ana', 'Solapur');
```

Inserting in one table using another table:

If the other table contains a set of fields that are necessary to fill the first table, you can use the "select" command to fill data into the table.

Syntax −

```
“INSERT INTO first_table_name [(column1, column2, ... columnN)]
   SELECT column1, column2, ...columnN
   FROM second_table_name
   [WHERE condition];”
```

Example -Insert into newstud select * from stud;

UPDATE Command:

The “Update” command is used to change existing records in table. This commands edits data from one or more records. It is also used to alter data which is already present in table.

Syntax:

```
“UPDATE <table_name>

SET <column_name = value>

WHERE condition;”
```

Example:

```
“UPDATE students
SET due_fees = 10000
WHERE stu_name = 'OM';”
```

DELETE Command:

To delete part or all of a table's records, use the "DELETE" command. The user will eliminate every entry if they fail to specify the "WHERE" criterion.

Syntax:

```
“DELETE FROM <table_name>
WHERE <condition>;”
```

Example:

```
“DELETE FROM students
WHERE rollno = 100;”
```

2.7 Data Control Language (DCL):

This is used to control access over the data that is control who can have access to with, who can log in to?, which privileges and rights user users have on the database

DCL Commands:

- **“Grant”-**This command gives privileges to user on database.
- **“Revoke”-**This command takes back the entitlement.

DCL commands are used to “enforce database security in a multiple user database environment”. “GRANT and REVOKE” are the two DCL commands. Privileges on a database object “can only be added or removed by the database administrator or the owner of the database item”.

➢ **“GRANT” Command:**

SQL “GRANT” is command used to provide privileges on database objects to users.

Syntax:

“GRANT privilege_name ON object_name TO {user_name |PUBLIC |role_name}
[WITH GRANT OPTION];”

- **“privilege_name** -Specifies privilege granted to user. Sample access rights are “CREATE, EXECUTE, and SELECT”.
- **object_name** –This is the name of database objects like TABLE, VIEW etc.
- **user_name** - This is the user name that should be given access rights.
- **PUBLIC** is used to grant access rights to all the users.
- **ROLES** are set of privileges bundled together.
- **WITH GRANT OPTION** - allows a user to grant access rights to other users”.

Example: “GRANT SELECT ON emp TO u1;”

Granting the "SELECT" permission on the "emp" table to the user "u1" can be enhanced by using the "WITH GRANT OPTION." This option allows "u1" to grant the "SELECT" privilege on the "emp" table to another user, such as "u2." If you subsequently revoke "SELECT" privilege from "u1," "u2" will still retain the "SELECT" privilege on the "emp" table.

➢ **“ REVOKE” Command:**

The REVOKE command “removes user access rights or privileges on database objects”.

Syntax:

“REVOKE privilege_name ON
object_name FROM {user_name |PUBLIC |role_name}”

Example: “REVOKE SELECT ON emp FROM u1;”

The following command will revoke the "SELECT" privilege on the "emp" table from user u1. When this privilege is revoked, u1 will no longer be able to perform "SELECT" operations on the table. However, if user u1 has been granted the "SELECT" privilege on the same table by multiple users, they will still retain the ability to perform "SELECT" operations until all those who granted the permission revoke it. Note that you can only revoke privileges that you have personally granted.

❖ **Privileges and Roles:**

➢ **Privileges:** The access “rights granted to a user on a database object are defined by their privileges”. Two categories of privileges exist.

1) “System privileges” – The system privileges allow user to “CREATE, ALTER, or DROP” database objects.

2) “Object privileges” – The object privileges allow user to “EXECUTE, SELECT, INSERT, UPDATE, or DELETE” data from database objects to which privileges apply.

The above rules also apply for ALTER and DROP system privileges.

➢ **Roles:**

A role is a set of access rights or privileges. Instead of granting privileges one by one we can grant Roles. As a result, by defining roles, you can give or take away users' privileges, which has the effect of giving or taking away privileges. Oracle has predefined system roles that we can use or construct our own. Some of privileges granted to system roles “CONNECT, RESOURCE, DBA”.

“CONNECT”: “CREATE TABLE, CREATE VIEW, CREATE SYNONYM, CREATE SEQUENCE, AND CREATE SESSION”.

“RESOURCE”: “CREATE PROCEDURE, CREATE SEQUENCE, CREATE TABLE, CREATE TRIGGER” etc. Restricting access to database items is the main use of the RESOURCE role.

“DBA”: All System Privilege’s Creating Roles:

Syntax:

“CREATE ROLE role_name

[IDENTIFIED BY password];”

Example: To create a role called "dev" with password as "pwd", code will be as follows:

“**CREATE ROLE DEV IDENTIFIED BY PWD”**

Using a role to "GRANT or REVOKE" privileges to users is more convenient than giving each user a privilege directly. If a role has a password associated with it, you must use that password to identify it when you "GRANT or REVOKE" capabilities to the role. A role's "GRANT or REVOKE" privilege is as follows.

Example: To grant “CREATE TABLE” privilege to a user by creating a testing role:

“CREATE ROLE test”

Second, grant a “CREATE TABLE” privilege to ROLE testing. We can add more privileges to “ROLE”.

“GRANT CREATE TABLE TO test;”

Third, grant the role to user.

“GRANT test TO u1;”

To revoke “CREATE TABLE” privilege from testing “ROLE”, you can write:

“REVOKE CREATE TABLE FROM test;”

Syntax:

“DROP ROLE role_name;”

Example: To drop a role called dev, you can write: **“DROP ROLE test;”**

2.8 Transaction Control Language (TCL)

Transaction Control Language “commands are used to handle transactions in the database. They are used to manage the changes made by DML-statements. TCL allows statements to be grouped together into logical transactions”. TCL commands can only use with DML commands like INSERT, DELETE and UPDATE only.

TCL Commands:

- **“COMMIT-** used to save all transactions to database.
- **ROLLBACK**- used to undo transactions that have not already been saved to the database.
- **SAVEPOINT**- used to roll transaction back to a certain point without rolling back entire transaction”.

  Syntax: “Savepoint Savepointname; Eg-Savepoint Sv1;”

- **Commit:**

A Transaction is logical unit of work that contains one or more SQL statement individual changes to the database tables are grouped into logical transactions put the database can save the changes or reject the changes. Insertions Deletions and update into the table is not permanent until and unless the work is explicitly committed to the database. When the work is completed the user can see the changes. There are three types of commits command available.

- Implicit commit
- Explicit commit
- Automatic commit

In implicit commit, Data Manipulation Language commands are automatically committed by giving the command commit. Implicit commit is done at the time of normal quitting.

In explicit commit if the changes during one transaction has to be committed we have to use the commit command explicitly.

Syntax:Commit;

Autocommit-If you set auto commit on then any changes are immediately save to the database after insert update and delete by default this auto commit is off.

- **Rollback-**

Rollback refers to the procedure used to undo changes made by data manipulation language commands insert update delete. To reverse the transactions up to the most recent commit, use this command.

Syntax:

"Rollback;

Rollback to savepoinname;

Rollback to sp1;"

This command will cancel effect of all DML commands executed after savepoint sp1.

Delete from employee where Departmentnumber = 10;

rollback;

This command will delete all the records of dept 10 and then cancels the deletion.

2.9 Data Query Language (DQL)

Data Query Language is used to fetch data from database. This is implemented with "Select" statement. Selection is in read only format.

SELECT: It is used to choose an attribute based on the circumstances specified in the "WHERE" clause, or even in its absence.

"SELECT column1,column2 FROM table_name ;"

"column1, column2": fields names of table

"table_name": From where we want to access.

All rows in the table with the fields column1 and column2 will be returned by this query.

- To fetch entire table or all fields in the table:

"SELECT * FROM table_name;"

• Fields to retrieve query Name, age, and Rollno from the Student table. If no condition is supplied, all rows will be retrieved.

“SELECT ROLLNO, NAME, AGE FROM Student;”

❖ **Knowledge Check 2:**

Fill in the blanks:

1. Transaction completes its execution is said to be_______.
2. We can have______ Primary keys in a table.
3. _________operator tests column for absence of data.
4. In ______________ cases a DML statement is not executed.

Outcome Based Activity

Create Table Book with appropriate attributes and add all table level constraints.

2.10 Constraints

Constraints on integrity guarantee correctness and consistency in relational databases. These limitations are guidelines that are applied to a table's columns in order to preserve data integrity.

- **Column-Level Constraints:** These constraints apply to a single column, ensuring that the data in that specific column adheres to certain rules.
- **Table-Level Constraints:** These constraints apply to the entire table, affecting one or more columns to enforce broader rules and relationships within the table.

  Referential Integrity (RI) is maintained through various types of integrity constraints, which include Primary Key, Foreign Key, Unique, Check, Null, and Default constraints. These constraints are essential for controlling the type of data entered into a table, thereby ensuring the data's accuracy and reliability. If an action attempts to violate any of these constraints, it will be aborted to preserve data integrity.

  Here are some commonly used constraints available in SQL:
- **“NOT NULL”** –This prevent entering null value in a column. Column with not null constraint can never contain a null value.
- **“UNIQUE”** – This makes sure that “all values in a column are different”. A unique constraint is a rule that prevents duplicate values in one or more columns in a table.

- **“PRIMARY KEY”** - A primary key uniquely identifies each row in a table. It combines the properties of NOT NULL and UNIQUE constraints to ensure that no two rows can have the same primary key value, and that no primary key value can be null. A primary key can be composed of one or more columns, enforcing uniqueness across the specified columns. This key can be defined using either column-level constraints or table-level constraints. Importantly, a table can have only one primary key.
- **“FOREIGN KEY”** - A foreign key is used to establish this connection, linking data from one table (the foreign table) to a base table. The purpose of the foreign key constraint is to maintain referential integrity. It can be defined at either the table level or the column level. The attribute that serves as a foreign key in one table is the primary key in another table.
- **“CHECK”** - makes ensuring that every value in the column meets a predetermined requirement.
- **“DEFAULT”** - Sets a column's default value in the event that no value is given. If user skips entering value for this column default value will appear in all rows.

➢ Constraints are of two types “table level constraints” and “column level constraint”.

“Column level Constraints” -These constraints are specified at column level a constraint given at column level is known as column level constraints it cannot refer to multiple columns it refers only that column where the constant is specified

Example-Column level constraint

```
CREATE TABLE STUD(ROLLNO NUMBER(5) PRIMARY KEY,
SNAME VARCHAR(20) NOT NULL,
PHONE NUMBER(10)UNIQUE,
CITY VARCHAR(10) DEFAULT ‘SOLAPUR’,
BR_ID NUMBER(5) CONSTRAINT FK REFERENCE BRANCH(BR_ID),
AGE NUMBER(3) AGE INT(3) CHECK (AGE>=17))
);
```

“Table level constaint”- Table level constraints are given at table level.They can refer to more than one column of the table.primary key constraint if it composite primary key constraint can be defined using table level constraints.

Example-Table level constraint

```
CREATE TABLE STUD(ROLLNO NUMBER(5),
SNAME VARCHAR(20),
PHONE NUMBER(10),
CITY VARCHAR(10) ,
BRID NUMBER(5) ,
CONSTRAINT PK PRIMARY KEY(ROLLNO),
CONSTRAINT NN NOT NULL(SNAME),
CONSTRAINT FK FOREIGN KEY(BRID) REFERENCE BRANCH(BRID),
CONSTRAINT UQ UNIQUE (PHONE) );
```

Constraints can be added to the existing table with the help of Alter Table command.

Syntax:

```
          “ALTER TABLE table_name
          ADD CONSTRAINT MyUniqueConstraint UNIQUE(column1, column2...);

ALTER TABLE EMP
ADD CONSTRAINT DF DEFAULT CITY= ‘PUNE’;”
```

We can drop the constraint from the existing table using alter table with drop option

```
“ALTER TABLE EMP
 DROP DEFAULT;
Or
ALTER TABLE EMP
 DROP DF;”
```

2.11 Summary

- SQL is a powerful language, and its statements are categorized into the following categories: "DDL (Data Definition Language), DML (Data Manipulation Language), DCL (Data Control Language), DQL (Data Query Language), and TCL (Transaction Control Language)".
- "Integrity constraints" are rules that ensure the "accuracy and consistency" of data within a relational database.
- In relational databases, data integrity is maintained through referential integrity. Various integrity constraints include "Primary Key, Foreign Key, Not Null, Check, Default, and Unique".
- A PRIMARY KEY uniquely identifies each row in a table by combining the properties of NOT NULL and UNIQUE.
- The Primary Key constraint ensures the unique identification of rows in a table. It can consist of multiple columns and can be set using both column-level and table-level constraints.
- A referential constraint or referential integrity constraint is "a logical rule about the values in one or more columns in one or more tables that uniquely identifies a row in another table".
- A Foreign Key establishes a link between data in two tables, enforcing referential integrity by referencing the data in a foreign table from a base table.

2.12 Self-Assessment Questions

1. Explain DDL commands.
2. Explain DML commands.
3. Explain DCL commands.
4. Explain TCL commands.
5. Explain DQL commands.
6. Explain List Constraints.

2.13 References

1. Geeks for Geeks
2. Data analysis using SQL by Gordon Linoff

3. The Applied SQL Data Analytics by Benjamin Johnston, Matt Goldwasser, and Upom Malik
4. Learning SQL by Alan Beaulieu
5. The Applied SQL Data Analytics - Quick, Interactive Approach to Learning Analytics with SQL, 2nd Edition by Upom Malik, Matt Goldwasser, Benjamin Johnston
6. Data Analytics: Data Science for Beginners, SQL Computer Programming for Beginners, Statistics for Beginners by Matt Foster

Chapter 3:
SQL BASICS FOR ANALYTICS

Learning Outcomes:

- Students will be able to understand various forms of Select statements.
- Students will be able to understand about how operators can be used with Select commands.
- Students will be able to understand using Between, Distinct, In, Like with Select Statements.

Structure

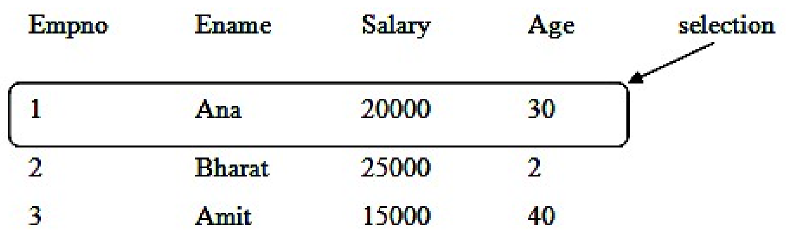
3.1 Introduction

The relational database can be queried using the SQL language. Using a SQL statement to retrieve data from a database is known as querying the database. We require SQL Data Query Language for this. To retrieve data from a database, utilize data query language. The SELECT statement is among the most often used SQL commands. The result can be shown in read-only formats by selecting different operators.

3.2 Select Statement

The SELECT statement is used to query the database and get specific data based on predefined criteria. Data from a database table can be retrieved using the select statement. There are two variations of select statements.

- **Projection operation** Specific properties defined with select are selected by the projection process. Because the project operation partitions the relation or table vertically, it is known as vertical partitioning.
  Syntax : "Select column1, column2…. Column n from table;"
  This is called as project operation here column 1 to column n are the attributes of the tables what we want to fetch will be displayed .
- **Selection operation**: Tuples that meet a specified condition are chosen using the choose method. The rows from a table that meet a particular condition are chosen in a select process.
  Syntax:
  Select * from tablename;
  Example- select * from emp;
  Selection Diagram:



Empno	Ename	Salary	Age
1	Ana	20000	30
2	Bharat	25000	2
3	Amit	15000	40

Projection Diagram:

Select empno,ename from emp;

Empno	Ename	
1	Ana	Projection
2	Bharat	
3	Amit	

Select Statement Syntax:

“SELECT [DISTINCT | ALL]{*|select_list}
FROM {table_name[alias]|view_name}
[{table_name[alias]|view_name}]...
[WHERE condition]
[GROUP BY condition_list]
[HAVING condition]
[ORDER BY {column_name| column_# [ASC | DESC] } ...”

“The **SELECT** is command verb and needed to SELECT OR PROJECT DATA.”
“The **FROM** clause is mandatory it specifies one or more tables or views to retrieve data to be shown in output.”
“The **WHERE** clause is optional and it specifies the condition and fetches the data satisfying given condition.”
“The **GROUP BY** clause is optional. It arranges data into groups using column or columns on which groups are formed.”
“The optional **HAVING** clause specifies conditions on which groups to include in a result table. The groups are specified by the **GROUP BY** clause.”
“The **ORDER BY** clause is optional. It sorts results on one or more columns in ascending or descending order.”

Syntax:
“SELECT column1, column2, column N FROM table_name;”
Column1, Column2... are the attributes from a table from where values are to be fetched.

- **Example** -Select empno, ename from emp;

This command will display empno and ename column of emp table.
SELECT * FROM table_name;

- **Example**- “Select * from emp;”

This will display all columns and all rows from emp table.

3.3 The WHERE Clause:

To provide a criteria for data extraction from one or more tables, use the **WHERE** clause. Certain rows from the table are returned in accordance with the specified criterion. To retrieve only the necessary records, we should filter the entries using the WHERE clause. In addition to the SELECT statement, the WHERE clause is also utilized in the UPDATE and DELETE statements, among other statements.

Syntax:

```
“SELECT column1, column2, column N FROM table_name
        WHERE [condition];”
```

A condition can be given using the Arithmetic, Logical operators.

“Arithmetic operator”- Arithmetic operations on numerical operands or table characteristics can be carried out with the use of arithmetic operators. The addition (+), subtraction (-), multiplication (*), and division (/) operators are arithmetic operations.

Operator	**Meaning**
+	Addition
-	Subtraction
*	Multiplication
/	Division
% (Modulo)	Returns remainder of a division..

In SQL, Operator Precedence also applies to arithmetic operators. The division and multiplication operators are evaluated before the addition and subtraction operators in an arithmetic phrase that has several operators. The expression is evaluated from left to right when the operators have the same amount of precedence.

Arithmetic operation in SQL SELECT statement:

“SELECT <Expression>[arithmetic operator]<expression>...

FROM [table_name]
WHERE [expression];”

We can use arithmetic operator as shown below to add two attributes Salary and Commission from emp table to get Net salary.

- Example- SELECT Salary+Commision from Emp;

This Select Command Calculates total price before discount and after discount for each ordered item by multiplying attributes Unit price and Quantity and display heading regular price.

“SELECT OrderID, ProductID,
UnitPrice*Quantity "Regular Price",
UnitPrice*Quantity-Discount "Price After Discount"
FROM order;”

Here we Select Stock,Stock%3 from product table using arithmetic operator %.

- SELECT Stock, Stock%3 "Modulo by 3" FROM products;

We can use arithmetic operators with select to carry out any operation.

- Example-Select 11/4 from Dual;
-

Relational Operators:

=	Equal
>	Greater than
<	Less than
>=	Greater than or equal
<=	Less than or equal
<>,!=	**Not equal to**

Syntax:

- “SELECT * FROM Products WHERE Price = 170;”

Here all the columns from product tables are selected but those rows where price is equal to 170 are displayed in output.

- SELECT * FROM Products WHERE Price > 170;

Here all the columns from product tables are selected but those rows where price is greater than 170 are displayed in output.

- SELECT * FROM Products WHERE Price < 300;

Here all the columns from product tables are selected but those rows where price is less than 300 are displayed in output.

- SELECT * FROM Products WHERE Price <= 300;

Here all the columns from product tables are selected but those rows where price is less than equal to 300 are displayed in output.

- SELECT * FROM Products WHERE Price <> 700;

Here all the columns from product tables are selected but those rows where price not equal to 700 are displayed in output.

3.4 The "SELECT DISTINCT" statement:

Rows with only different values are returned using the "SELECT DISTINCT" query. We wish to list distinct values in a column of a table that frequently has a lot of duplicate entries.

Syntax:

"SELECT DISTINCT column1, column2 FROM table_name;"

- Example- "SELECT DISTINCT Country FROM Customers;"

- **Between… And:** This operator is used to select a range of values inclusive of between and operator .The result-set contain all the values matching values within range of value1 and value2.

- Example- "SELECT * FROM Products
WHERE Price BETWEEN 40 AND 40;"

- **Like operator-** Like operator is used to retrieve the rows where particular pattern is present. Like operator use two wildcard characters for pattern matching. % and? Are the two characters used with Like operator.

% -This operator represent 0,1 or more characters.

_ Represents single character.

- SELECT * FROM Customers

WHERE City LIKE 'S%';

This command will select all cities starting with S. %will represent any number of characters after S.

➢ SELECT * FROM Customers

WHERE City LIKE 'P_ _ _ ';

This command will select all cities starting with P and will have three more characters represented by three underscore symbols.

- **In operator -**This is a logical operator which checks whether specified value exists in set of values.

➢ SELECT * FROM Customers

WHERE City IN ('Pune','Nagpur');

Here only those customers with city pune or Nagpur will get selected.

Using Column Alias Different headings can be specified in select output .By default column name appears as headers.

Select CatId, CatName From Categories; ******

❖ **Knowledge Check 1:**

State True or False:

1. _ represent single character in Like operator.
2. Distinct avoids duplicate values.
3. In is similar to or.
4. Select has two forms selection and projection.

Outcome Based Activity

Select Name of employees having salary>=4000

3.5 The “AND”, “OR” and “NOT” Operators:

Conditions in the "WHERE" clause can be combined using the "AND," "OR," and "NOT" operators.

- The "AND and OR operators filter records based on multiple conditions”.
- The "AND operator shows a record if all the conditions joined by "AND" are TRUE”.

- The "OR operator shows a record if any of the conditions joined by "OR" is TRUE”.
- The "NOT operator shows a record if the condition(s) is NOT TRUE”.
-

“AND” Syntax:

“SELECT column1, column2
FROM table_name
WHERE condition1 AND condition2 AND condition3 ...;
SELECT * FROM Customers
WHERE Country='India' AND City='Mumbai';”

“OR” Syntax:

“SELECT column1, column2, ...
FROM table_name
WHERE condition1 OR condition2 OR condition3 ...;”

“NOT” Syntax:

“SELECT column1, column2, ...
FROM table_name
WHERE NOT condition;”

OR Examples-

➢ “SELECT * FROM Customers
WHERE City='Solapur' OR City='Pune’;”

This following SQL statement selects the all fields from "Customers" where city is "Solapur" OR "Pune":

➢ “SELECT * FROM Customers
WHERE Country='India' OR Country='Shrilanka';”

This SQL statement selects all fields from "Customers" where country is "India" OR "Shrilanka":

NOT Example-

This SQL statement selects all fields from "Customers" where country is “NOT” "Pakistan":

➢ “SELECT * FROM Customers
WHERE NOT Country='Pakistan';”

Combining “AND”, “OR” and “NOT” example:

This SQL statement selects all fields from "Customers" where country is "India" AND city must be "Pune" OR "Solapur":

Example:

➢ “SELECT * FROM Customers
WHERE Country='India' AND (City='Solapur' OR City='Pune');”

This SQL statement selects all fields from "Customers" where country is NOT "Germany" and NOT "USA":

Example:

➢ “SELECT * FROM Customers
WHERE NOT Country='Germany' AND NOT Country='USA';”

❖ **Knowledge Check 2:**

State True or False:

1. In and operator all condition has to be true.
2. Not negates the condition.
3. Between and operator is used to select a range of values inclusive of between and operator
4. Alias is other name given to attributes.

Outcome Based Activity

Select all the cities start with A and 4 character long.

3.6 Summary

- The SELECT statement in SQL is versatile and can handle multiple tasks. It retrieves data from a database table based on specified criteria.

- Arithmetic operators such as addition (+), subtraction (-), multiplication (*), and division (/) perform mathematical operations either on numeric operands or attributes of tables within SQL queries.
- Relational operators like >, <, >=, <=, <> are also used within SELECT statements to establish conditions in the WHERE clause.
- Multiple conditions can be combined using the AND and OR operators in SQL. The NOT operator negates a condition, reversing its logic.
- SQL SELECT queries can utilize various clauses such as DISTINCT, IN, BETWEEN, and LIKE to refine and filter the results based on specific criteria.

3.7 Self-Assessment Questions

1. Write Select commands using arithmetic operators.
2. Write Select commands using Relational operators.
3. Explain Use of AND, OR, NOT with Select.
4. Explain Like operator
5. Explain Distinct and Between and operator commands
6. Write command to select all columns from student table
7. Write command to select name column from student table
8. Write select command using % and?

3.8 References

1. Geeks for Geeks, W3School
2. Data analysis using SQLby Gordon Linoff
3. The Applied SQL Data Analyticsby Benjamin Johnston, Matt Goldwasser, and Upom Malik
4. Learning SQL by Alan Beaulieu
5. The Applied SQL Data Analytics - Quick, Interactive Approach to Learning Analytics with SQL, 2nd Edition by Upom Malik, Matt Goldwasser, Benjamin Johnston
6. Data Analytics: Data Science for Beginners, SQL Computer Programming for Beginners, Statistics for Beginners by Matt Foster

Chapter 4:
AGGREGATE FUNCTIONS

Learning Outcomes:

- Students will be able to understand concept of Aggregate functions
- Students will be able to understand and have clear Understanding about Working of Group by with Select commands.
- Students will be able to know Using Having Condition with Group by Statements.

Structure:

4.1 Introduction:

Aggregate functions in SQL are essential tools in business analytics as they take multiple rows as input and consolidate them into a single output, providing valuable insights for top management. SQL provides numerous aggregate functions such as SUM, AVG, COUNT, MAX, and MIN, which calculate totals, averages, counts, and maximum or minimum values from specified columns or rows in a database table.

4.2 Aggregate functions:

Aggregate functions are alternatively referred to as group functions because they operate on sets of values. These functions are designed to accept multiple rows as input and yield a single value as output. In business contexts, top management often focuses on summarized or aggregated data. Aggregate functions play a crucial role in producing summarized insights from database tables, consolidating large datasets into meaningful summaries that aid decision-making and analysis.

Common aggregate functions include:

- “Count”
- “Average (arithmetic mean)”
- “Maximum”
- “Minimum”
- “Sum”
- “Stddev”

1. **COUNT()** : This function is used to Count the number of rows from a table. COUNT function work with all data type. Count can take column or columns from table or * as argument.

Count() can take following forms.

- **COUNT(*):**This function counts all the rows of a table, including those with null values. COUNT(*) returns the total number of rows in the specified table. It considers duplicates and counts rows regardless of whether they contain null values. COUNT(*) does not require parameters and does not support DISTINCT. Unlike other aggregate functions, COUNT(*)* operates independently of specific column information. Its purpose is solely to return the total row count of a table, including all rows, whether they are duplicates or contain null values.
- **COUNT(COLUMN_NAME):**This function will count the number of not-null values from column.
- **COUNT(DISTINCT COLUMN_NAME):** This function will count number of distinct values from column specified.

Syntax:

“COUNT(*)

or

COUNT([ALL|DISTINCT] expression) ”

“ALL-Applies the aggregate function to all values.”

“DISTINCT-Specifies that COUNT returns the number of unique non null values.”

Example:

- **“Select COUNT(*)** from Employee;”

This statement will count no of employees from Employee table.

- **“Select COUNT(*)** from Employee where deptno=10;”

This statement will count no of employees from deptno 10.

- **“Select COUNT(Empno) from Employee ;”**

This statement will count number of employees from Employee table.

- **“SELECT COUNT(DISTINCT Job) FROM EMP;”**

This statement will count no of different jobs from Emp table.

2. **SUM()-**This function will sum up all the values from the column. The column should be numeric**.** Sum function is used to calculate the sum of all selected columns. It works on numeric fields only. The DISTINCT keyword that allows us to omit duplicates from our results. Return unique and not null values.

Syntax:

“SUM()

or

SUM([ALL|DISTINCT] expression)”

The DISTINCT keyword that allows us to omit duplicates from our results.

Example:

“SELECT SUM(SALARY) FROM EMP;”

This statement will sum up the salaries from from all rows of emp table.

Example: SUM() with WHERE

“SELECT SUM(COST) FROM PRODUCT

WHERE Ord_Qty>30;”

This command will sum up the cost from all rows where Ord_Qty is greater than 30.

3. **AVG() function-**

The AVG function calculates the average value of numeric data types in SQL. When applied, it considers all non-null values for computation. It divides the sum of values in a specified column by the total number of records to derive the average.

To refine its operation, the DISTINCT keyword forces the AVG() function to only consider unique values, ensuring each distinct value contributes equally to the average calculation. Conversely, the ALL keyword, which is the default behavior, instructs the function to include all values in the calculation without any filtering based on uniqueness.

Syntax:

"AVG()

or

AVG([ALL|DISTINCT] expression)"

Example:

"SELECT AVG(COST) By FROM PRODUCT_MAST;"

This command will sum up the cost attribute from Product_mast table and divide it with no of rows to get average.

4. MAX () Function-

The maximum value in the column that is supplied as an argument is found using the MAX() function. All data types—alphanumeric, date, and numeric—are supported. The greatest value among all the chosen values in a column is found using this function.

Syntax:

"MAX()

or

MAX([ALL|DISTINCT] expression)"

Example – "Write a query to find the maximum salary in employee table."

"Select MAX(salary) from Employee;"

Example:

"SELECT MAX(RATE)

FROM PRODUCT_MAST;"

5. MIN() Function:

MIN function is used to find the minimum value of a certain column. This function determines the smallest value of all selected values of a column.

Syntax:

"MIN()

or

MIN([ALL|DISTINCT] expression)"

Example:

"SELECT MIN(RATE)

FROM PRODUCT_MAST;"

6. STDDEV()-

The standard deviation of the column sent in as an argument is found using the STDDEV function. As an illustration to find the salary standard deviation in the Employee table, write a query.

Choose STDDEV (pay) from the Employee

4.3 GROUP BY Clause-

The "GROUP BY" Clause is used to summaries data in Association with aggregate function group by helps to produce summary reports of data. **"GROUP BY"** Clause is used with Select statement of SQL for arranging similar data into groups based on columns. We can use group functions for arranging data if the rows in a table have similar values then we can arrange them in groups using group by groups are formed on specific column .Any column specified in select should be used as group by expression or should be argument of group function.

Syntax:

"SELECT column1, group_function_name(column2)"

"FROM table_name"

"WHERE condition"

"GROUP BY column1, column2"

"HAVING condition"

“ORDER BY column1, column2;”

❖ **Knowledge Check 1:**

Outcome Based Activity 1

Use all Group functions in single query.

4.4 HAVING Clause:

When sorting rows by grouping, it's possible to control which groups appear in the output by applying restrictions. The HAVING clause specifically filters results after using GROUP BY. The **“WHERE”** clause determines initial conditions before grouping. Conditions applied to columns help identify portions of the final result set for each group. It's unnecessary to combine aggregate functions like COUNT(), SUM(), etc., with the WHERE clause in such cases. Instead, the HAVING clause is required. Any non-aggregated attribute mentioned in the HAVING clause must also be included in the GROUP BY clause.

Syntax:

“SELECT column1, Group_function_name(column2)
FROM table_name
WHERE condition
GROUP BY column1, column2
HAVING condition
ORDER BY column1, column2;”

Example: “Select count(EId),Country from Emp Group by Country Having count(EId)>30;”
This command will give number of employees from different country having more than 30 employees

Examples:

- “select job, deptno,count(*) from emp group by job;”

This is invalid because column 2 i.e. deptno is not in group by .This command should be like,

- “select job, deptno,count(*) from emp group by job, deptno;”

Find the average salary and number of analyst and developer from each department.

- “select Deptno, avg(salary) from employee where job in (‘administrator’,’ programmer’) group by Deptno”

Write a query to display minimum salary and highest salary of each department

- “select average Deptno, min(sal) max(sal) 0from employee group by Deptno;”

Compute the, minimum maximum salary of those groups of employees where job is equal to manager or accountant

- “select avg(sal), Max(sal),job from employee where job in (‘manager’ or ‘accountant’) group by job;”

Find out find out difference between minimum salary and maximum salary earned by a person

- “select Max (salary)- Min (salary) from employee where Deptno= 10;”

Find difference between average earning salary of Department number 30 and Department number 40

- “select avg(salary)– average(salary) from employee Where Deptno= 30 and Deptno=40;”

Examples:

(Having)-

- “SELECT NAME, SUM(SALARY) FROM Employee
  GROUP BY NAME
  HAVING SUM(SALARY)>23000;”

- **“SELECT** DEPTNO, COUNT(EMPNO) **FROM** Employee
  GROUP BY DEPTNO
  HAVING COUNT(EMPNO)>4;”

❖ **Knowledge Check 2:**

Fill in the Blanks:

1.___________values is ignored by group functions.

2.____________is a built in aggregate function in SQL finds maximum value from a column.

3.___________ functions take multiple rows as input.

Outcome Based Activity -2:

Write a query to form Groups on Course name field in student table also count no of student enrolled for each course.

4.5 Summary.

- Aggregate functions, alternatively known as group functions, analyze sets of values by processing multiple rows as input and producing a single output value. Common examples include AVG, SUM, COUNT, MIN, and MAX.
- The GROUP BY clause in SQL is utilized alongside aggregate functions to summarize data, facilitating the creation of summarized reports from a dataset.
- Within a SELECT statement, the GROUP BY clause organizes data into groups based on specified columns, enabling the aggregation of similar data.
- The HAVING clause works in conjunction with GROUP BY to filter and restrict the result set of grouped data based on specified conditions.

4.6 Self-Assessment Questions

1. Explain Aggregate functions with examples.
2. Explain Group by clause.
3. Explain Use of Having clause.
4. Explain Various form of Count()
5. Write Syntax of Group By
6. Write Syntax of Having

4.7 References

1. Geeks for Geeks, W3School
2. Data analysis using SQL by Gordon Linoff
3. "The Applied SQL Data Analytics by Benjamin Johnston, Matt Goldwasser, and Upom Malik"
4. Learning SQL by Alan Beaulieu
5. The Applied SQL Data Analytics - Quick, Interactive Approach to "Learning Analytics with SQL, 2nd Edition by Upom Malik, Matt Goldwasser, Benjamin Johnston"
6. Data Analytics: Data Science for Beginners, SQL Computer Programming for Beginners, Statistics for Beginners by Matt Foster

Chapter 5:
FUNCTIONS

Learning Outcomes:

- Students will learn built in String functions
- Students will learn built in Numeric functions
- Students will learn built in Date functions
- Students will learn built in Conversion functions
-

Structure

5.1 STRING FUNCTIONS

String functions are for string manipulations and string functions process on input string and return string or numeric value:

1. "ASCII(str)"

Gives back the string str's leftmost character's numerical value. 0 is returned if the string is empty. If string is NULL, returns NULL. For the characters which numeric values between "0" and "255", ASCII() is functional.

```
"SQL> SELECT ASCII('2') FROM DUAL ;
ASCII('2')
50
SQL> SELECT ASCII('dx') FROM DUAL;
ASCII('dx')
100 (ascii value for d)"
```

2. “INSTR(str,substr)”

Gives back the location of substring substr's initial occurrence in string “str”. The string and substring are specified as arguments .

```
“SQL> SELECT INSTR('foobarbar','bar') FROM DUAL;
 INSTR('foobarbar','bar')”
```

3. “LEFT(str,len)”

Returns the string str's leftmost n characters, or NULL if none of the arguments are NULL.

```
 “SQL> SELECT LEFT('foobarbar',5);
 LEFT('foobarbar',5)
 Fooba”
```

4. “LENGTH(str)”

Gives back the string str's length in bytes. Multiple bytes are counted in a multi-byte character. This indicates that LENGTH() returns 10 for a string that has five two-byte characters.

```
 “SQL> SELECT LENGTH('text') FROM DUAL;
 LENGTH('text')”
```

5 “LOWER(str)”

Gives back the string str with every letter converted to lowercase based on the mapping of the current character set.

```
 “SQL> SELECT LOWER('SQL')
FROM DUAL
Sql”
```

6. “LPAD(str,len,padstr)”

Returns the string str after it has been left-padded to a length of len characters using the string padstr. The return value is truncated to len characters if str is longer than len.

```
 “SQL> SELECT LPAD('hi',4,'??') FROM DUAL;
| LPAD('hi',4,'??')
|??hi”
```

7. “LTRIM(str)”

Returns string str after removing the leading space character.

“SQL> SELECT LTRIM(' barbar') FROM DUAL;

LTRIM(' barbar')

Barbar”

8. “**MOD()”**

It returns remainder of n divided by m.

Syntax: “SELECT MOD(17, 4);”

Output: 2

9. “PI()”

It returns value of PI displayed with 6 decimal places.

Syntax: “SELECT PI();”

Output: 3.141593

10. “SQRT()”

It returns square root of a number.

Syntax: “SELECT SQRT(25);”

Output: 5

11. “SUBSTRING (str FROM pos FOR len)”

This function extracts a substring from the input string str, starting at the specified position pos. If the len parameter is provided, the function returns a substring that is len characters long, beginning from pos. The versions using FROM follow standard SQL syntax. Additionally, pos can be a negative value, indicating that the substring starts pos characters from the end of the string, rather than from the beginning. This negative positioning for pos applies universally across all variations of the function.

“SQL> SELECT SUBSTR('ABCDEFG',-5,4) "Substring" FROM DUAL;

Substring

CDEF”

```
“SQL> SELECT SUBSTRING('foobarbar' FROM 4) from Dual;
 SUBSTRING('foobarbar' FROM 4)
 barbar
SQL> SELECT SUBSTRING('Quadratically',5,6) from Dual;
 SUBSTRING('Quadratically',5,6)
 Ratica”
```

12. “TRIM([{BOTH | LEADING | TRAILING} [remstr] FROM] str)”

TRIM([remstr FROM] str):

This function processes a string str to remove occurrences of remstr from both the beginning and/or end of the string. By default, if the specifiers BOTH, LEADING, or TRAILING are not explicitly provided, BOTH is assumed. The remstr parameter is optional; if omitted, spaces will be removed instead of a specific substring.

```
“SQL> SELECT TRIM('  bar   ');
 TRIM('  bar   ')
 bar
SQL> SELECT TRIM(LEADING 'x' FROM 'xxxbarxxx');
 TRIM(LEADING 'x' FROM 'xxxbarxxx')
 barxxx

SQL> SELECT TRIM(BOTH 'x' FROM 'xxxbarxxx') FROM DUAL;
 TRIM(BOTH 'x' FROM 'xxxbarxxx')
 bar

SQL> SELECT TRIM(TRAILING 'xyz' FROM 'barxxyz') FROM DUAL;
 TRIM(TRAILING 'xyz' FROM 'barxxyz')
 Barx”
```

|

13. “UPPER(str)”

Gives back the string str with every character converted to uppercase based on the mapping of the current character set.

"SQL> SELECT UPPER('number') FROM DUAL;"

5.2 NUMERIC FUNCTIONS:

Numerical values are returned by numeric functions, which accept numbers as input. The following are the numerical functions:

1. "ABS(X)"

"This function returns the absolute value of X. Where X is input value to the function.

```
SQL> SELECT ABS(5) FROM DUAL;
5
SQL> SELECT ABS(-20) FROM DUAL;
20"
```

2. "CEILING(X)"

The smallest integer value that is not less than X is returned by these functions.

```
 "SQL> SELECT CEILING(3.46) FROM DUAL;
4"
SQL> SELECT CEIL(-6.43) FROM DUAL;
-6"
```

```
"SELECT CEIL(123.45) AS x, CEIL(32) AS y, CEIL(-123.45) AS z
RESULT: x = 124, y = 32, z = -123"
```

3. "FLOOR(X)"

The greatest integer value that is not bigger than X is returned by this function.

```
 "SQL>SELECT FLOOR(6.55) FROM DUAL;
 FLOOR(6.55)
6"
```

.

4. “POWER(X,Y)”

These two functions return the value of X raised to the power of Y.

```
“SQL> SELECT POWER(3,3) FROM DUAL;
 POWER(3,3)
26”
```

5. “ROUND(X,D)”

This function returns X, rounded to the nearest integer. The function returns X with D decimal places rounded when a second argument, D, is provided. If D is not positive, all digits to the right of the decimal point will be removed. Look at this example.

```
 “SQL>SELECT ROUND(5.693793) FROM DUAL;
 ROUND(5.693793)
6”
“SQL>SELECT ROUND(5.693793,2) FROM DUAL;
 ROUND(5.693793,2)
5.69”
```

```
“ROUND(7.2) returns 7
ROUND(7.6) returns 9”
```

The binary (two parameter) version requires two arguments: “the number to be rounded and a positive or negative integer that allows you to specify how many spaces the value is rounded to”. The binary variant always returns a double: “Positive second arguments specify the number of places that must be returned following the decimal point”. As an example:

“ROUND(123.4566, 3) returns 123.456”

Negative second arguments indicate the number of places that must be returned before the decimal point. As an example:

```
 “ROUND(123.4, -3) returns 0
   ROUND(1234.56, -3) returns 1000”
```

6. “TRUNCATE(X,D)”

The value of X shortened to D decimal places is returned by this function. The decimal point is eliminated if D is 0. D number of values in the integer portion of the value are trimmed if D is negative.

```
“SQL>SELECT TRUNCATE(6.536432,2) FROM DUAL;
TRUNCATE(6.536432,2)
6.53”
```

It returns “the number n with m fewer decimal places. If m = 0, the result has neither a decimal point nor a fractional portion”.

The input's decimal (non-integral) portion is eliminated in its unary (one argument) version. As an example:

“SELECT TRUNC(3.14159265) AS x FROM DUAL”

RESULT: x = 3

You can specify how many spaces the number is truncated at in the binary (two parameter) version. A double is always returned by the binary version. As an illustration:

“SELECT TRUNC(3.14159265, 3) AS yFROM DUAL”

RESULT: y = 3.141

7. “SQRT(X)”

“This function returns the non-negative square root of X.”

```
“SQL>SELECT SQRT(49) FROM DUAL;
 SQRT(49)
6”
```

8. “SIGN(X)”

“This function returns the sign of X (negative, zero, or positive) as -1, 0, or 1.”

```
“SQL>SELECT SIGN(-4.66) FROM DUAL;
 SIGN(-4.66)
-1
```

SQL>SELECT SIGN(0) FROM DUAL;
 SIGN(0)
0
SQL>SELECT SIGN(4.65) FROM DUAL;
 SIGN(4.65)|
1”

9. “MOD(N,M)”-

“This function returns the remainder of N divided by M. Consider the following example:”
“SQL>SELECT MOD(29,3) FROM DUAL;
 MOD(29,3)
2”

10. “GREATEST(n1,n2,n3,..........)”

“The **GREATEST()** function returns the greatest value in the set of input parameters (n1, n2, n3, and so on). **GREATEST()** function return the largest number from a set of numeric values:”
“SQL>SELECT GREATEST(3,5,1,7,33,99,34,55,66,43) FROM DUAL;
 GREATEST(3,5,1,7,33,99,34,55,66,43)
99”

11. “LEAST(N1,N2,N3,N4,......)”

“The **LEAST()** function is the revese of the **GREATEST()** function. Its purpose is to return the least-valued item from the value list (N1, N2, N3, and so on).”
“SQL>SELECT LEAST(3,5,1,7,33,99,34,55,66,43) FROM DUAL;
 LEAST(3,5,1,7,33,99,34,55,66,43)
1”

12. “STDDEV(expression)”

"The **STD()** function is used to return the standard deviation of expression. This is equivalent to taking the square root of the **VARIANCE()** of expression. The following example computes the standard deviation of the **PRICE** column in our **CARS** table:"

```
"SQL>SELECT STD(PRICE) STD_DEVIATION FROM CARSFROM DUAL;
 STD_DEVIATION
6650.2146"
```

13. "ACOS(X)"

"This function returns the arccosine of X. The value of X must range between -1 and 1 or NULL will be returned. Consider the following example:"

```
"SQL> SELECT ACOS(1) FROM DUAL;
 ACOS(1)"
```

14. "ASIN(X)"

"The **ASIN()** function returns the arcsine of X. The value of X must be in the range of -1 to 1 or NULL is returned."

```
"SQL> SELECT ASIN(1) FROM DUAL;
 ASIN(1)
1.5606963266949"
```

15. "ATAN(X)"

"This function returns the arctangent of X."

```
"SQL> SELECT ATAN(1) FROM DUAL;
 ATAN(1)
0.67539716339645"
```

16. "FORMAT(X,D)"

"The **FORMAT()** function is used to format the number X in the following format: ###,###,###.## truncated to D decimal places. The following example demonstrates the use and output of the **FORMAT()** function:"

```
"SQL>SELECT FORMAT(423423234.65434453,2) FROM DUAL;
```

FORMAT(423423234.65434453,2)
|423,423,234.65”

17. “SIN(X)”

“This function returns the sine of X. Consider the following example:”

“SQL>SELECT SIN(90) FROM DUAL;
SIN(90)
0.793996”

18. “COS(X)”

“This function returns the cosine of X. The value of X is given in radians.”

“SQL>SELECT COS(90) FROM DUAL;
COS(90)
-0.44706361612916”

19. “TAN(X)”

“This function returns the tangent of the argument X, which is expressed in radians.”

“SQL>SELECT TAN(45) FROM DUAL;
TAN(45)
1.619665”

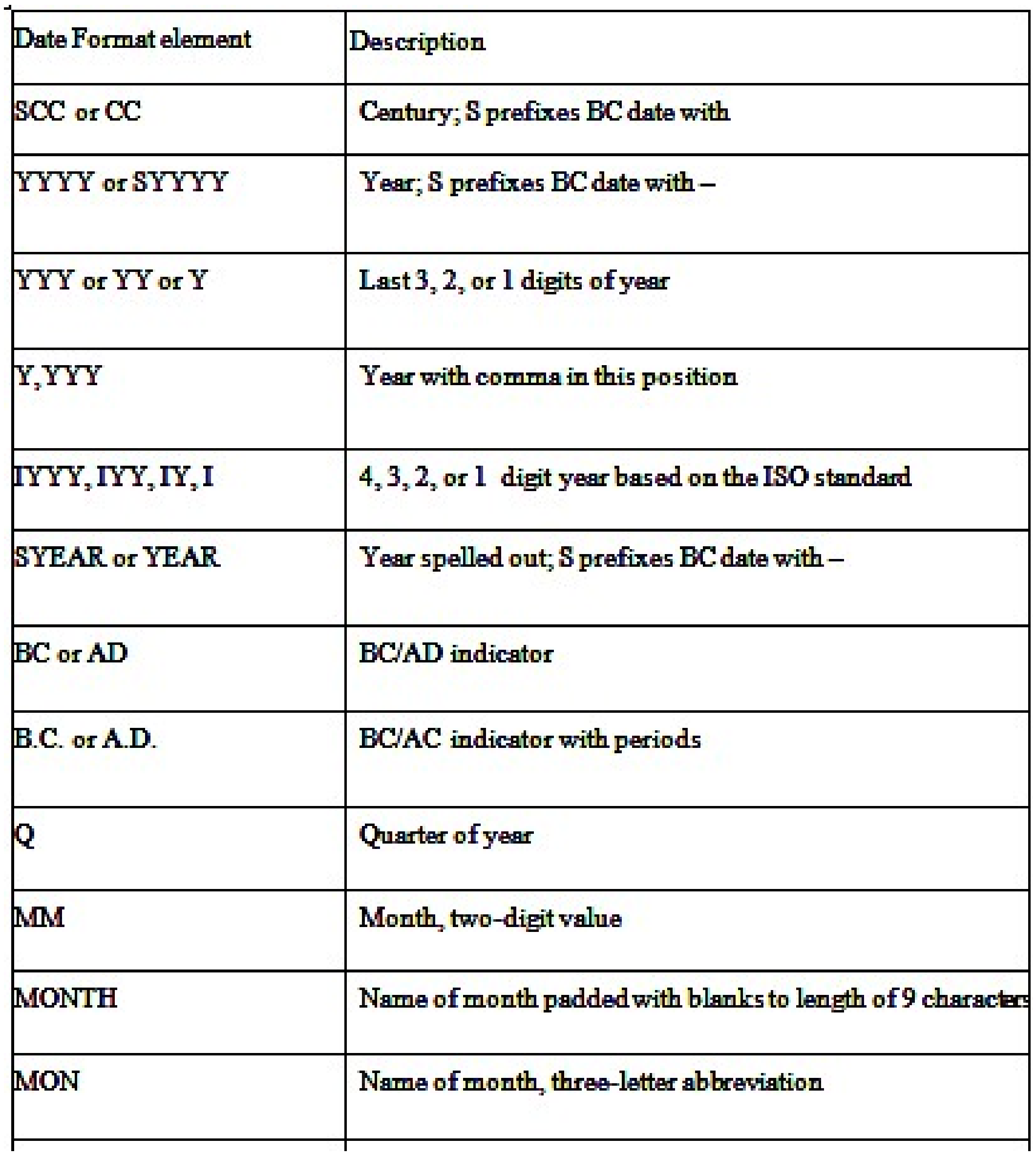
20. “CONV(N,from_base,to_base)”

“The purpose of the CONV() function is to convert numbers between different number bases. The function returns a string of the value N converted from from_base to to_base. The minimum base value is 2 and the maximum is 36. If any of the arguments are NULL, then the function returns NULL. Consider the following example, which converts the number 5 from base 16 to base 2:”

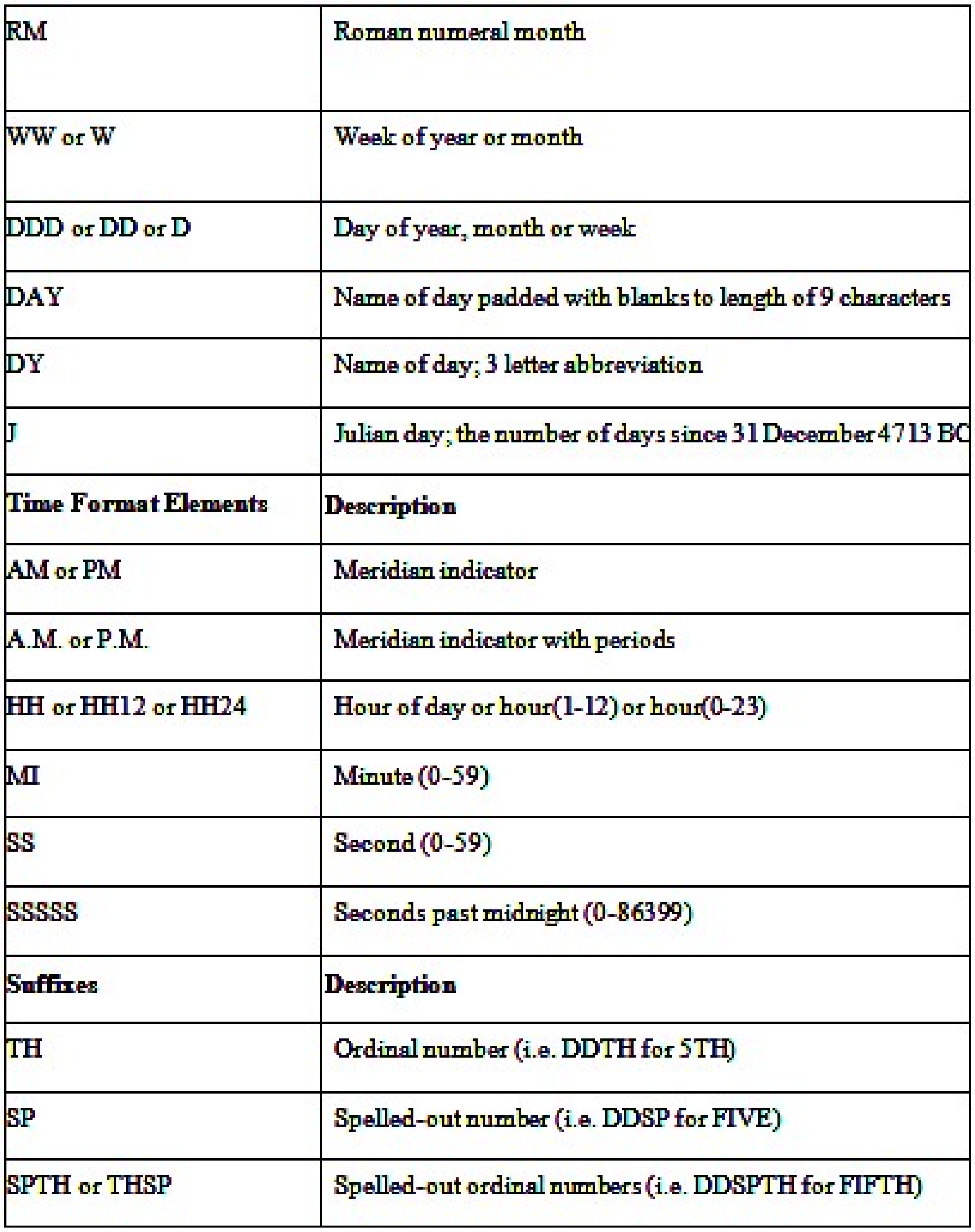
“SQL> SELECT CONV(5,16,2) FROM DUAL;
CONV(5,16,2)
101”

5.3 Date functions

DATE is a datatype that store the “month, day, year, century, hours, minutes,” and “seconds”.



Date Format element	Description
SCC or CC	Century; S prefixes BC date with
YYYY or SYYYY	Year; S prefixes BC date with –
YYY or YY or Y	Last 3, 2, or 1 digits of year
Y,YYY	Year with comma in this position
IYYY, IYY, IY, I	4, 3, 2, or 1 digit year based on the ISO standard
SYEAR or YEAR	Year spelled out; S prefixes BC date with –
BC or AD	BC/AD indicator
B.C. or A.D.	BC/AC indicator with periods
Q	Quarter of year
MM	Month, two-digit value
MONTH	Name of month padded with blanks to length of 9 characte
MON	Name of month, three-letter abbreviation



RM	Roman numeral month
WW or W	Week of year or month
DDD or DD or D	Day of year, month or week
DAY	Name of day padded with blanks to length of 9 characters
DY	Name of day; 3 letter abbreviation
J	Julian day; the number of days since 31 December 4713 BC
Time Format Elements	**Description**
AM or PM	Meridian indicator
A.M. or P.M.	Meridian indicator with periods
HH or HH12 or HH24	Hour of day or hour(1-12) or hour(0-23)
MI	Minute (0-59)
SS	Second (0-59)
SSSSS	Seconds past midnight (0-86399)
Suffixes	**Description**
TH	Ordinal number (i.e. DDTH for 5TH)
SP	Spelled-out number (i.e. DDSP for FIVE)
SPTH or THSP	Spelled-out ordinal numbers (i.e. DDSPTH for FIFTH)

1. “ADD_MONTHS”-

This function Returns a date value after adding ‘n’ months to the date ‘x’.

Syntax: “ADD_MONTHS (date, n)”

Eg.- “select ADD_MONTHS ('16-Sep-71', 3) from dual 16-Dec-71”

ADD_MONTHS adds a full month to the date. The month_shift argument allows you to enter a fractional value, but ADD_MONTHS will always round to the next full integer to the closest zero.

Examples:

- select “ADD_MONTHS ('27-FEB-2005', 1.5) from dual

31-Mar-2005”

We also can use negative values which gives previous dates.

- select “ADD_MONTHS ('27-FEB-2005', -1) from dual

31-Jan-2005”

2. “Last_day”-

It is used to determine the number of days remaining in a month from the date ‘x’ specified. LAST_DAY (x) The LAST_DAY function returns the date of the last day of the month for a given date.

- “select LAST_DAY ('01-Jun-16') from dual

30-Jun-2016”

3. “Next_day”-

The date of the first weekday named by day that is later than date is returned by NEXT_DAY. The argument day must be a day of the week in the date language; it can be either the complete name or an abbreviated. The return type is always DATE.

- “NEXT_DAY (x, week_day) Returns the next date of the ‘week_day’ on or after the date ‘x’ occurs.”
- “select NEXT_DAY ('01-Jun-07', 'Wednesday') from dual

04-JUN-07”

4. “Months_Between” –

“This function returns the number of months between dates x1 and x2.”

The MONTHS_BETWEEN function determines how many months have passed between two dates and outputs the difference as a numerical value.

Rules:

1) “If date1 comes after date2, then MONTHS_BETWEEN returns a positive number.
2) If date1 comes before date2, then MONTHS_BETWEEN returns a negative number.

3) If date1 and date2 both fall on the last day of their respective months, then MONTHS_BETWEEN returns a whole number

4)If date1 and date2 are in different months and at least one of the dates is not a last day in the month, MONTHS_BETWEEN returns a fractional number. The fractional component is calculated on a 31-day month basis and also takes into account any differences in the time component of date1 and date2."

Examples:

- "select MONTHS_BETWEEN ('29-FEB-2016', '31-MAR-20')   from dual

-1"

- "select MONTHS_BETWEEN ('31-MAR-1995', '27-FEB-1994') from dual

13"

- "select MONTHS_BETWEEN ('31-JAN-2006', '10-MAR-2006') from dual

-1.3225706"

- "SELECT MONTHS_BETWEEN(TO_DATE('02-02-1995','MM-DD-YYYY'),
- TO_DATE('01-01-1995','MM-DD-YYYY')) "Months"FROM DUAL;

Months

1.03225706"

5. "Round"-

When a format mask is used, the ROUND function rounds a date value to the closest date. With the exception of working with dates, it is identical to the normal numeric ROUND function, which rounds an integer to the nearest number of specified accuracy.

Examples

- "Select ROUND (TO_DATE ('12-MAR-2016'), 'MONTH') from dual;
  01-MAR-2016
- Select ROUND (TO_DATE ('16-MAR-2016'), 'MONTH') from dual;
  01-APR-2016
- select ROUND (TO_DATE ('01-MAR-2006'), 'YYYY') from dual;
  01-JAN-2006
- select ROUND (TO_DATE ('01-SEP-2006'), 'YEAR') from dual;
  01-JAN-2007"

6. “Arithmetic With date”:

With the Oracle Date datatype, we may execute a wide range of arithmetic operations. For a consequent date value, we can add or remove a number from the date. To find the number of days between two dates, we can subtract one from the other. By dividing the total number of hours by 24, we can add hours to a date.

Addition and Subtraction examples:

```
“SQL> select sysdate, sysdate+1/24, sysdate +1/1440, sysdate + 1/76400 from dual;
SYSDATE             SYSDATE+1/24        SYSDATE+1/1440      SYSDATE+1/76400
01-Jul-2016 06:32:12 01-Jul-2016 06:32:12 01-Jul-2016 06:33:12 01-Jul-2016 06:32:13”
```

5.4 CONVERSION FUNCTIONS:

Conversion functions in SQL transform a value from one data type to another using the syntax datatype1 TO datatype2, where datatype1 is the input type and datatype2 is the desired output type. When SQL queries encounter mismatched data types, SQL handles the conversion either implicitly or explicitly.

Implicit type conversion occurs automatically when SQL converts one data type to another based on system requirements. For example, numeric data can be converted to character data and vice versa, if necessary. Implicit conversion is only performed when the data types involved are compatible and the system can recognize and perform the conversion accurately. It cannot proceed with invalid or incorrectly provided data types.

Explicit type conversion, on the other hand, is when programmers manually convert data from one type to another within a query. This method gives developers more control over how data is transformed and ensures compatibility between data types as needed in complex SQL operations.

“Explicit Data Type Conversion”:

The programmer has the option to directly change data types between different forms if they so want. SQL routines are available specifically for this use. The following SQL functions are −

1. “TO_CHAR”

“This function is used to explicitly convert a number or date data type to char.”

Syntax:

“TO_CHAR(number,format,parameters)”

This method converts a number to a character using the precise format specified by the syntax. Decimal characters, group separators, and other customizations can be made using the options.

Example:

“SELECT TO_CHAR(sysdate,“Month DD,YYYY”) FROM DUAL;”

“This returns the system date in the form of 31 July, 2017”

“SELECT TO_CHAR (12345, '99999D99') FROM DUAL;”

This returns the 12345.00

2. “TO_NUMBER”

To specifically convert a string to a number, use this function. TO_NUMBER displays an error if the string that has to be converted doesn't contain any numeric characters.

Syntax:

“TO_NUMBER(number,format,parameters)”

This function converts the supplied string to a number using the precise format specified by the syntax. To convert a string to a number, you can choose the format and parameters.

Example:

“Select TO_NUMBER (‘353.67’) from dual;”

“This returns the string 353.67 in numerical format.”

3. “TO_DATE”

“This function takes character values and returns the output in the date format.”

Syntax:

“TO_DATE(number, format, parameters)”

This function converts the supplied string to a number using the precise format specified by the syntax.

“SELECT TO_DATE(‘2017/06/31’,’yyyy/mm/dd’) FROM DUAL;”

“This takes the values in character format and returns them in date format as specified.”

5.5 Summary.

- Functions serve as fundamental components in any programming language, including SQL.

- SQL offers a wide range of functions across various categories for manipulating data effectively.
- Categories of SQL functions include Numeric, String, Date, Conversion, Miscellaneous, and Aggregate functions.
- String functions in SQL are specifically designed for manipulating strings, processing input strings, and returning either string or numeric values.
- The INSERT function in SQL returns the position of the first occurrence of a substring (substr) within a string (str), specified as arguments.
- The SUBSTRING function in SQL retrieves a substring from a string (str) starting at a specified position (pos). Versions of this function with a length (len) argument return a substring of a specified length.
- The TRIM function in SQL removes specified prefixes or suffixes (remstr) from a string (str). If not specified, spaces are removed by default unless otherwise indicated (BOTH, LEADING, or TRAILING).
- Implicit data type conversion in SQL occurs automatically within the system to convert data from one type to another as needed.
- Explicit data type conversion in SQL allows programmers to manually convert data from one form to another based on specific requirements or preferences.

5.6 Self-Assessment Questions

1. Write Syntax of Round function
2. Write Syntax of Trunc function
3. Write syntax Upper function
4. Write syntax To_Char function
5. Explain 5 Numeric functions with example
6. Explain 5 Date functions with example
7. Explain 5 Conversion functions with example
8. Explain 5 string functions with example

5.7 References

1. Geeks for Geeks, W3School
2. Oracle.com
3. Data analysis using SQL by Gordon Linoff
4. “The Applied SQL Data Analytics by Benjamin Johnston, Matt Goldwasser, and Upom Malik”
5. Learning SQL by Alan Beaulieu
6. “The Applied SQL Data Analytics - Quick, Interactive Approach to Learning Analytics with SQL, 2nd Edition by Upom Malik, Matt Goldwasser, Benjamin Johnston”
7. Data Analytics: Data Science for Beginners, SQL Computer Programming for Beginners, Statistics for Beginners by Matt Foster

Chapter 6:
JOINS

Learning Outcomes:

- Students will learn concept of SQL JOINS.
- Students will learn different Types of JOINS & their Working.
- Students will learn SET Operators

Structure

6.1 Introduction

Joins, as the name suggests, are a feature provided by Structured Query Language (SQL) to combine rows from two or more tables based on a common column. This operation links tables by matching values in specified columns. Joins are essential for retrieving data from multiple tables simultaneously in a Relational Database Management System (RDBMS).

When crafting a join query in a database, the operation involves combining data from different tables using specific criteria. Typically, columns being joined must share compatible data types to ensure accurate matching and efficient query execution.

Join columns often include key columns such as primary keys or foreign keys, ensuring relational integrity between tables. Null values do not match during joins, meaning rows with null values in the join column are excluded from the result set. Well-designed joins enhance data retrieval accuracy and efficiency in relational databases.

Facts about Joins:

- When retrieving data from multiple tables in SQL, Joins provide the mechanism to combine rows based on shared columns.
- SQL Joins merge rows from specified tables in a SELECT query using columns that have matching values.
- SQL offers various types of joins including “Equi Join, Non-Equi Join, Left Outer Join, Right Outer Join, Full Outer Join, Self Join, Inner Join, and Cartesian Product”.
- Joins are executed using operators like =, >, <, >=, <=, !=, BETWEEN, LIKE, etc., to specify the relationship between columns.
- To avoid ambiguity when columns share the same name across tables, SQL requires specifying table names before column names (e.g., table1.column1, table2.column2).

Syntax:

“SELECT table1.column1, table2.column2...
FROM table1,table2
WHERE table1.commonfield = table2.commonfield;”

Here Column1, Column2 are the columns from the tables1 and table2 respectively.

table1.commonfield , table2.commonfield are the columns specified in where clause on which join is based.

Example:

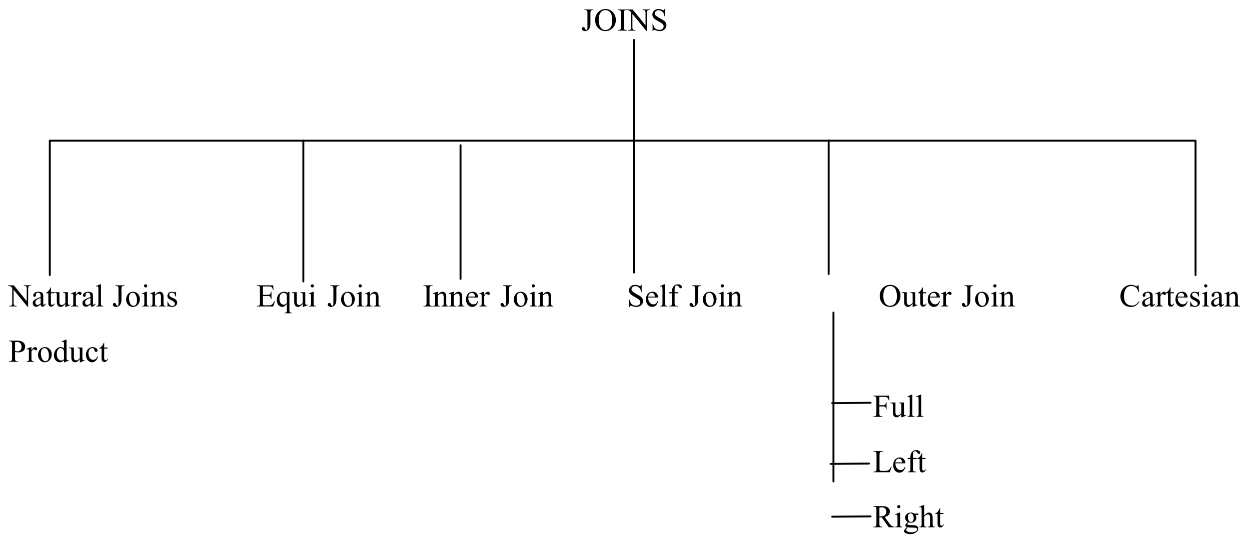
“Select emp.eno,emp.ename,dept.dname from emp,dept where emp.dno=dept.dno;”

6.2 Types of Joins

SQL Joins are categorized into following types.

- Natural
- Equi or Inner Joins
- Self Joins
- Outer joins
    - “Left Outer join

  - ▪ Right Outer Join
  - ▪ Full Outer Join”
- Cartesian Product



- **Natural Join-**

In Natural Join “two tables based on same attribute name and datatypes output table will contain all the columns of both table but same columns are selected once avoiding duplication of columns”. If Table1(column1,column2) and Table2(column2, column3) Then natural join gives column1, column2, column3.

SYNTAX:

“SELECT * FROM table1 NATURAL JOIN table2;”

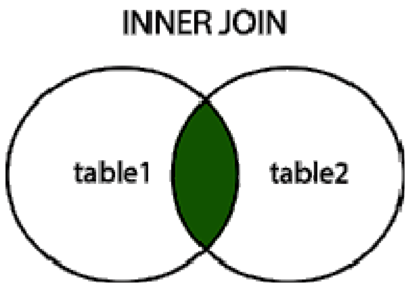
Example:

Select eno,Doj,Sal from emp
Natural join Backup;

This will select all the columns which are common in two table avoiding the duplicates and the remaining column from tables.

- **INNER JOIN Or EQUIJOIN:**

An INNER JOIN, also known as an EQUIJOIN, creates a new table by combining column values from two tables based on a specified join condition. This type of join compares each row from the first table (table1) with every row from the second table (table2) to identify and retrieve all rows that satisfy the join condition.



Syntax:

“SELECT Table1.Column1,Table1.Column2,Table2.Column1,....

INNER JOIN Table2

ON Table1.sameColumnName = Table2.sameColumnName;”

Example:

SELECT Emp.EmpID, Emp.Ename, dept.DID, dept.dName

FROM Emp

INNER JOIN Dept ON Emp.dID=dept.dID

- **Non Equi Join**:

If join is established using operators other than =,like,Between,greater etc.

Then it is non Equi join.

Example-

“SELECT stud.name, rec.id, rec.city

FROM stud, rec

WHERE Stud.id < Rec.id ;”

- **Outer Join**

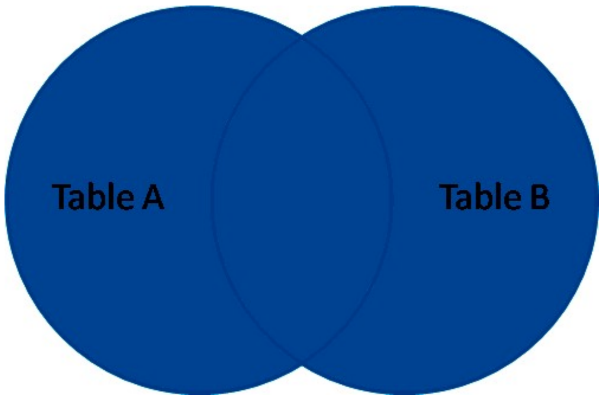
In natural and equijoins, only the rows that satisfy the join condition appear in the output. However, in outer joins, rows that do not match the condition can also exist in the result set. When performing a join operation, the result set typically includes all rows that match the specified condition from the involved tables.

Outer join operations extend this functionality by including rows from either the left, right, or both tables in the result set, even if they do not satisfy the join condition. Outer joins are categorized into the following types based on which table's rows are included regardless of matching conditions:

- FULL OUTER JOIN
- LEFT OUTER JOIN
- RIGHT OUTER JOIN

FULL OUTER JOIN

Full Join or the Full Outer Join outputs all rows from both the tables that is left(Table1) and the right(Table2) table not bothering about the condition matched or unma**tched**.



Syntax:

“SELECT Table1.Column1,Table1.Column2, Table2.Column1, ...

FROM Table1

FULL JOIN Table2

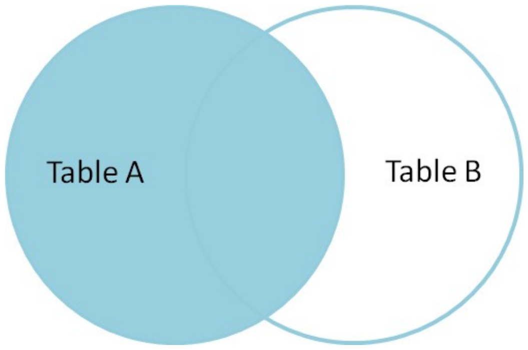
ON Table1.MatchingColumnName = Table2.MatchingColumnName;”

Example:

```
“SELECT Emp.Ename, Projects.ProjectID
FROM Emp
FULL JOIN Projects
ON Emp.EmpID = Projects.EmpID;”
```

LEFT OUTER JOIN:

The LEFT JOIN, also known as the LEFT OUTER JOIN, retrieves all records from the left table and those that meet a specified condition from the right table. For records in the left table that have no corresponding matches in the right table, the result set will include NULL values for the right table's columns. This means that all rows from the left table are returned, regardless of whether there are matches in the right table.



Syntax:

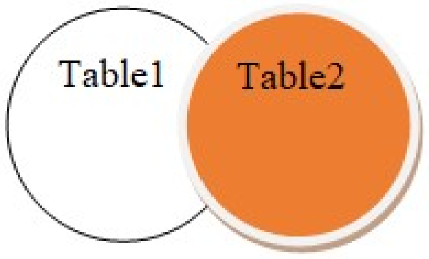
```
"SELECT Table1.Column1,Table1.Column2,Table2.Column1,....
FROM Table1
LEFT JOIN Table2
ON Table1.MatchingColumnName = Table2.MatchingColumnName;"
```

Example:

```
"SELECT Emp.Ename, Projects.ProjectID, Projects.ProjectName
FROM Emp
LEFT JOIN
ON Emp.EmpID = Projects.EmpID ;"
```

RIGHT OUTER JOIN:

The RIGHT JOIN, also known as the RIGHT OUTER JOIN, retrieves all records from the right table and the records from the left table that meet the specified condition. If there are no matching records in the left table, the result set will include NULL values for those rows. Essentially, it returns every row from the right table, regardless of whether a corresponding match exists in the left table.



Syntax:

“SELECT Table1.Column1,Table1.Column2,Table2.Column1,....

FROM Table1

RIGHT JOIN Table2

ON Table1.MatchingColumnName = Table2.MatchingColumnName;”

Example:

1 “SELECT Emp.Ename, Projects.ProjectID, Projects.ProjectName

2 FROM Emp

3 RIGHT JOIN

4 ON Emp.EmpID = Projects.EmpID;”

- **Self-Join:**

A SELF JOIN is a join where a table is joined with itself. This means each row in the table is combined with itself, a process often referred to as Autojoin. When a table appears more than once in the FROM clause of a join query, using the same table name multiple times can lead to ambiguity errors. To prevent this, table aliases can be employed. Table aliases provide alternate names for the table within the query, helping to clarify and distinguish the different instances of the table.

We list employees and their manager’s name –

Mgr could be foreign key into manager table, but it has to be a ‘foreign key’ in emp table itself.

Syntax:

“SELECT s1.col_name, s2.col_name...

FROM table1 s1, table1 s2

WHERE s1.common_col_name = s2.common_col_name;”

Example-:

“select e.ename,m.ename from emp m,emp e where e.mgr=m.empno;”

- **Cross Join**

A CROSS JOIN is a type of join where each row from one table is combined with every row from another table. When a WHERE condition is applied, a CROSS JOIN functions similarly to an INNER JOIN, filtering the results based on the specified condition. Without a WHERE condition, a CROSS JOIN produces a Cartesian product, resulting in all possible combinations of rows from the involved tables.

Syntax:

```
“SELECT table1.column1, table2.column2...
FROM  table1, table2 [, table3 ]”
```

```
“EG > SELECT  ID, NAME, AMOUNT, DATE
  FROM CUSTOMERS, ORDERS;”
```

Every row from the customer table is connected with every row from the orders table to produce the Cartesian product because the join condition is lacking.

Joining Multiple Tables-

We can join three tables using join statement and this can be extended to n level.

We need minimum two where conditions and common field to achieve this.

```
“SELECT
  stud.name, course.name
FROM student
JOIN student_course
  ON stud.id = stud_course.stud_id
JOIN course
  ON course.id = stud_course.course_id;”
```

Here student and Student_course has student id in common and student and Course and student_course has course in common.

❖ **Knowledge Check 1**

State True or False

a. Join of a table to itself is self join.

b. INNER JOIN are also called as an EQUIJOIN

c. The Joining column should have same name in both the tables.

❖ **Outcome Based Activity 1**

Imagine structures of 3 tables and join them.

6.3 SET operators:

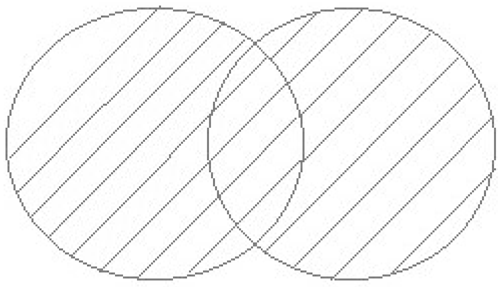
Set operations in SQL can be applied to the data in the tables. These are used to apply various conditions to the data in the table and obtain meaningful results.

There are three different types of SET operations.

1. UNION
2. INTERSECT
3. MINUS

UNION Operation

- The UNION set operator merges the results of two SELECT statements into a single result set, removing any duplicate rows so that each unique row appears only once. In contrast, the UNION ALL operator also combines the results but retains all duplicate rows in the final result.
- For a UNION operation to be applied, both SELECT statements must have the same number of columns with matching data types in the tables being combined.



Example:

The table1 contains(ID,Name),

ID	Name
1	Rahul
2	Amit

The table2 contains(ID,Name)

ID	Name
2	Amit
3	Ana

Union SQL query will be,

“SELECT * FROM table1

UNION

SELECT * FROM table2;”

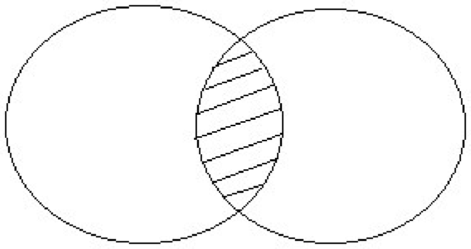
The result set table will look like,

ID	NAME
1	Rahul
2	Amit
3	Ana

“Union all operation is similar to Union. But it also shows the duplicate rows.”

INTERSECT:

The INTERSECT operation is used to combine the results of two SELECT statements, returning only the records that are common to both. For the INTERSECT operation to work, the number of columns and their data types in the SELECT statements must be the same..



Example:

Intersect query will be:

“SELECT * table1

INTERSECT

SELECT * FROM table2;”

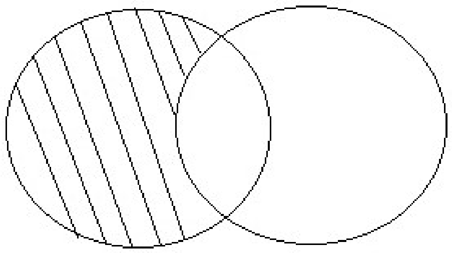
The result set table will look like

ID	NAME
2	Amit

MINUS

The MINUS set operator in SQL removes the results of the second query from the output if those results also appear in the first query's output. Both INTERSECT and MINUS operations yield unduplicated results.

The MINUS operation combines the results of two SELECT statements, returning only the rows that are present in the first query but not in the second. This operation ensures there are no duplicate rows in the final result, and the data is sorted in ascending order.



Example:

Minus query:

“SELECT * FROM table1

MINUS

SELECT * FROM table2;”

The result set table will look like,

ID	NAME
1	Rahul

Outcome Based Activity

Write difference between Natural and inner joins

6.4 Summary:

- In a Relational Database Management System (RDBMS), data from different tables can be collected using joins in queries. Columns in these tables can be joined if they have the same data types. Effective joins require that the join columns have compatible data types.
- Join columns are typically key columns such as primary keys or foreign keys. Null values do not participate in joins. SQL offers various types of joins, including Equi Join, Non-Equi Join, Left Outer Join, Right Outer Join, Full Outer Join, Self Join, Inner Join, and Cartesian Product.
- The joining operation in SQL is facilitated by operators such as =, >, <, >=, <=, !=, BETWEEN, and LIKE.
- SQL also provides set operations that can be applied to table data to derive meaningful results under different conditions.
- Three primary types of set operations in SQL are: **UNION**, **INTERSECT**, **MINUS**

6.5 Self-Assessment Questions

1. Explain Natural Join with Example
2. Explain Outer Join and its type with Example
3. Explain self-Join with Example
4. Explain Inner Join with Example
5. Explain set operators.
6. Write Syntax of Self Join
7. Write Rules for joining

8. Write syntax Union operation
9. Write syntax Intersect operation
10. Write syntax Minus operation

6.6 References

1. https://www. https://www.tutorialspoint.com/
2. https://docs.oracle.com
3. Geeks for Geeks, W3School
4. Data analysis using SQL by Gordon Linoff
5. The Applied SQL Data Analytics by Benjamin Johnston, Matt Goldwasser, and Upom Malik
6. Learning SQL by Alan Beaulieu
7. “The Applied SQL Data Analytics - Quick, Interactive Approach to Learning Analytics with SQL, 2nd Edition by Upom Malik, Matt Goldwasser, Benjamin Johnston”
8. Data Analytics: Data Science for Beginners, SQL Computer Programming for Beginners, Statistics for Beginners by Matt Foster

Chapter 7:
VIEWS & INDEXES

Learning Outcomes:

- Students will learn concept of SQL VIEWS &INDEXES
- Students will learn different types of VIEWS &INDEXES
- Students will learn advantages of VIEWS & INDEXES

Structure

7.1 Introduction:

In a “Relational Database Management System” (RDBMS), data access is controlled and not all information is available to every user. Access to data is granted based on user permissions. An SQL view is a named query stored in database, acting as a virtual table. This virtual table can include all rows or a subset of rows from one or more tables, depending on the specified conditions.

The view itself is created by a query, and tables from which a view derives its data are called **“base tables”.** Views do not store data physically; they exist as queries stored in the database.

When data is retrieved through a view, it dynamically fetches data from the base tables. Some operations, particularly certain update operations, might not be applicable to all types of views.

Uses of Views

- Simplify Data Retrieval: Views make data retrieval easier by encapsulating complex queries.
- Enhance Security: Views can restrict access to specific data, thereby improving security.
- Combine Multiple Tables: It can join multiple tables into a single virtual table for easier data manipulation.
- Aggregate Data: Views can summarize data, acting as aggregated tables.
- Hide Data Complexity: it can hide the complexity of underlying data structures, presenting a simpler interface to the user.
- Efficient Storage: Views occupy minimal storage space since only their definition is stored, not the actual data.
- User-Specific Data Presentation: Views can organize data in a manner tailored to specific user needs.
- Restrict Data Access: Views can limit what data users can see and modify, effectively showing only a part of the data.
- Generate Reports: Views can aggregate and summarize data from various tables, facilitating report generation.

7.2 Creating Views:

We can create Database views using "CREATE VIEW" statement. Views can be created from a "single table, multiple tables, or another view". A user needs to have the required system privilege in order to create a view.

Syntax:

```
"CREATE  OR REPLACE FORCE VIEW view_name
 AS SELECT column1, column2.....
FROM table_name
WHERE [condition]
WITH CHECK OPTION
WITH READ ONLY;"
```

"REPLACE" will replace the view with if specified name already exists, this query will replace it with new definition.

"FORCE" keyword is used for creating a view forcefully. If force keyword is used then a View is created even if "base table" does not exist. After creating a force View if we create "base table" the view will be automatically updated..

Here Viewname is the unique name given to view

Column1,column2… are the columns selected from base table or tables.

VIEW WITH CHECK OPTION:

Using "WITH CHECK OPTION" in "CREATE VIEW" statement make sure all "UPDATE" and "INSERT"s satisfy condition in view definition.

"UPDATE or INSERT" returns an error if they don't meet the criterion.

Example

```
"CREATE VIEW CUSTOMERS_VIEW AS
SELECT name, age
FROM  CUSTOMERS
WHERE age IS NOT NULL
WITH CHECK OPTION;"
```

Since the view is defined by data that does not include a "NULL" value in the AGE column, the "WITH CHECK OPTION" in this instance should prevent the entry of any "NULL" values in the view's AGE column.

WITH READ ONLY OPTION:

If we create view WITH READ ONLY option then data manipulation i.e. **insert,update** is not possible.

```
"CREATE VIEW clerk (id_number, person, department, position)
   AS SELECT empno, ename, deptno, job
        FROM emp
        WHERE job = 'CLERK'
  WITH READ ONLY;"
```

“CREATE VIEW studentView AS
SELECT NAME, ADDRESS
FROM Student
WHERE STU_ID < 50;”

A view named studentView is created with name,address fields and have data where student_id is<50.

How to display data of view:

SELECT * FROM StudentView;

It will show all rows from the view.

View Using Multiple table:

Multiple tables can be included in your “SELECT” statement in a manner very similar to how they are used in a typical “SQL SELECT” query.

Example: You may generate a view from numerous tables by just include them in “SELECT” command.

In the example provided, two tables, Student_Detail and Student_Marks, are used to create a view called MarksView.

Query:

1. CREATE VIEW MarksView AS
2. SELECT Student_Detail.id,Student_Detail.NAME, Student_Detail.ADDRESS,
3. “Student_Mark.MARKS”
4. “FROM Student_Detail, Student_Mark”
5. “WHERE Student_Detail.ID = Student_Marks.id;”

UPDATE, INSERT and DELETE on views

SQL views can be updatable, allowing data to be inserted or modified in base table. The “SELECT” statement defined within a view determines whether or not it is updatable. A view cannot be specifically marked as updatable with a special clause.

To ensure a view is updatable, its definition must be straightforward and must not include aggregate functions such as “SUM”, “AVG”, “MAX”, “MIN”, or “COUNT”. Additionally, the view should not involve any grouping, “DISTINCT”, or “JOIN” clauses, as these also render the view non-updatable.

Updating a View

A view is updatable only if–

- No DISTINCT clause be present in the SELECT.
- No aggregate functions.
- No set operators.
- No “ORDER BY” clause.
- “FROM” clause should not contain multiple tables.
- The “WHERE” clause should not contain subqueries
- No GROUP BY or HAVING.

Let’s create a view that is updatable:

```
“CREATE VIEW V AS
SELECT
  deptId, dName, MgrId, Date
FROM Dept;
```

The SELECT * will show all rows from view:

```
SELECT * FROM V;”
UPDATE V
  SET dname = “account” where deptno=10;
```

Inserting Rows into a View:

Data rows can be inserted to a view. The “INSERT” command is subject to the same guidelines as the “UPDATE” command.

```
“CREATE VIEW studentView AS
SELECT NAME, ADDRESS
FROM Student;
Insert into studentView  values(‘ana’,’11,central road pune’);”
```

“CREATE VIEW CUSTOMERS_VIEW AS

SELECT name, age

FROM CUSTOMERS

WHERE age IS NOT NULL

WITH CHECK OPTION;”

In this case, we cannot insert rows into “CUSTOMERS_VIEW” because not all “NOT NULL” columns are included in the view. If all “NOT NULL” columns were included, you could insert rows into a view similarly to how you insert them into a table.

Deleting Rows into a View:

Rows of data can be deleted from a view. The DELETE command is subject to same guidelines as the UPDATE and INSERT commands.

Following is an example to delete a record having AGE = 22.

“SQL > DELETE FROM CUSTOMERS_VIEW

WHERE age =22;”

This would ultimately delete a row from the “base table” CUSTOMERS and the same would reflect in the view itself.

Unupdatable view:

“create or replace view avg_view as

select course,avg(evaluation) as avg_eval

from registrations

group by course;”

This view contains aggregate function so un updatable.

Complex views:

Complex views can be built using multiple base tables and can include features such as:

- Join conditions

- A “GROUP BY” clause

- An “ORDER BY” clause

Direct Data Manipulation Language (DML) operations cannot be performed on complex views. To enable DML operations on these views, you need to write “INSTEAD OF” triggers, which instruct Oracle on how to apply the changes to the base table(s).
Examples:

```
“CREATE VIEW complex_view AS
  SELECT emp.empno, emp.ename, emp.job, emp.deptno, dept.dname, dept.loc
  FROM emp, dept
 WHERE emp.deptno = dept.deptno;”
```

Deleting View:
The Drop View statement can be used to remove a view.
Syntax
“DROP VIEW view_name;”
Example:
To remove the View MarksView, we can do the following action:

“DROP VIEW MarksView;”

7.3Advantages of view-:

- Simplified Data Retrieval: Views encapsulate complex queries, making data retrieval easier and more straightforward.
- Enhanced Security: By restricting access to specific data, views help improve security.
- Simplified Data Representation: Views can join multiple tables into a single virtual table, simplifying data representation.
- Data Aggregation: Views can summarize and aggregate data, making it easier to analyze.
- Hidden Complexity: Views can hide the complexity of the underlying database schema from users, presenting a simplified interface.
- Efficient Storage: Since views store only the query and not the actual data, they require minimal storage space.
- Custom Data Structuring: Views can structure data in a way that meets specific user or application requirements.

- Controlled Data Access: Views can limit what data users can see and modify, showing only a subset of the data.
- Facilitate Reporting: Views can aggregate and present data from various tables, which is useful for generating reports.

Knowledge Check 1

State True or False

1. Views exist virtualy.
2. A View is updatable if it contains aggregate function
3. View enforces security.
4. Drop view removes the view definition.

Outcome Based Activity

Create a view based on multiple table.

7.4 INDEX:CONCEPT

In a database, "an index is a data structure that improves the speed of data retrieval operations on a table at the cost of additional storage space". Indexes are created on columns to allow quick access to the rows. They work similarly to the index in a book, which helps you quickly locate specific information without scanning the entire book.

Types of Indexes

- "Primary Index": Automatically created when a primary key is defined. Ensures that each row can be uniquely identified.
- "Unique Index": Ensures that all the values in indexed column are unique.
- "Non-Unique Index": Improves the performance of queries but does not enforce uniqueness.
- "Composite Index": Created on multiple columns, improving the performance of queries involving these columns.
- "Clustered Index": Determines physical order of data in table and is typically created on primary key.
- 'Non-Clustered Index": does not change data's physical order; instead, it searches the database and then produces a distinct object that points back to the original rows.

Indexes are created in a database using the "CREATE INDEX" statement. This statement allows you to name the index, specify the table, and choose the column or columns to index. Additionally, you can indicate whether the index should be in ascending or descending order.
Indexes can also be unique, denoted by the "UNIQUE" type. A unique index ensures that there are no duplicate entries in the column or combination of columns on which the index is created. This prevents the insertion of duplicate values, maintaining data integrity.
Using the "CREATE INDEX" statement, you can efficiently manage data retrieval and enforce data integrity constraints within your database system.

CREATE INDEX Command-

“CREATE INDEX index_name ON table_name;”

“Single-Column Indexes”:

A single-column index is created on only one table column.

“CREATE INDEX index_name
ON table_name (column_name);”

Eg.-“CREATE INDEX ind_name
ON emp (ename);”

Unique Indexes:

In addition to preserving data integrity, unique indexes are employed to increase performance. No duplicate values can be added to a table while using a unique index.

Syntax :

“CREATE UNIQUE INDEX index_name
on table_name (column_name);”

Eg.-“create unique index date_ui on dates (trunc (calendar_date));”

Composite Indexes:

A composite index is set on two or more columns of a table.

Syntax

“CREATE INDEX index_name
on table_name (column1, column2);”

“CREATE INDEX ind_nmadd
on student (name, add);”

When deciding whether to create a single-column index or a composite index, consider the columns frequently used in query WHERE clauses as filter conditions. If only one column is used frequently, opt for a single-column index. However, if two or more columns are frequently used together in WHERE clauses, a composite index is preferable.

Implicit indexes are automatically generated by the database server upon certain actions. For instance, indexes are automatically created for primary key and unique constraints. When defining a column as a primary key, it automatically becomes an implicit index.

By leveraging single-column and composite indexes based on query requirements, you can significantly enhance query performance and database efficiency.

The DROP INDEX Command:

You can use the SQL "DROP" command to drop an index. When removing an index, caution should be used because the performance could get better or worse.

“DROP INDEX index_name;”

There are scenarios when using indexes might not be advisable, despite their intended performance benefits. To decide when using indexes should be reevaluated, take into account the following guidelines:

- Small Tables: Avoid creating indexes on small tables, as the overhead of maintaining and using indexes might outweigh the performance gains.

- Frequent Batch Updates or Inserts: Tables that undergo frequent and large batch updates or insert operations might experience performance degradation due to the overhead of maintaining indexes.

- **High Number of NULL Values:** Indexes should be avoided on columns with a high number of NULL values since indexing NULLs doesn't provide significant benefits and may consume unnecessary storage space.
- **Frequently Manipulated Columns:** Avoid indexing columns that are frequently manipulated (e.g., updated) since each modification operation on indexed columns requires additional maintenance overhead.

By carefully evaluating these factors, you can make informed decisions about when to use indexes to optimize query performance and when to refrain from using them to prevent potential performance issues.

7.5 Advantages of Indexes:

1. Faster Queries: They speed up data retrieval by quickly locating relevant rows.
2. Efficient Sorting: Indexes facilitate efficient sorting of data.
3. Improved Searches: They enhance search query performance by reducing scanning overhead.
4. Data Integrity: Unique indexes prevent duplicate entries, ensuring data integrity.
5. Optimized Joins: Indexes accelerate join operations between tables.
6. Space Efficiency: They optimize disk space by facilitating quicker data access.
6. Constraint Support: Indexes help enforce “primary key, unique, and foreign key” constraints.

By utilizing indexes effectively, database performance and efficiency can be significantly improved, ultimately enhancing the overall user experience.

Index Disadvantages

1. Increased Storage: Indexes consume additional storage space.
2. Overhead: Maintaining indexes incurs overhead during data modification operations.
3. Complexity: Managing indexes can introduce complexity, particularly in large databases.
4. Performance Impact: Poorly chosen or excessive indexes can degrade performance, especially for write-intensive operations.
5. Null Values: Indexes may not be beneficial for columns with high numbers of NULL values.
6. Small Tables: Indexes on small tables may not provide significant performance benefits and can even be detrimental.

6. Frequent Updates: Tables with frequent batch updates or inserts may suffer performance degradation due to index maintenance overhead.

Knowledge check 2

Outcome Based Activity

Draw structure of index table.

7.6 Summary.

SQL views are predefined SQL queries stored in the database under specific names, functioning as virtual tables. They can display all rows of a table or selected rows based on specified conditions, and can be created from one or multiple tables. Views are based on base tables, which are the tables from which they derive their data. They are stored as queries in the database and do not physically contain rows.

On the other hand, database indexes, similar to an index in a book, expedite data retrieval by pointing to specific data in tables. Indexes are created using the "CREATE INDEX" statement, specifying table, column(s) to index, and whether index is ascending or descending. Unique indexes prevent duplicate entries in indexed columns or combinations of columns.

In summary, views provide a structured way to access data stored in tables, while indexes facilitate quick data retrieval by pointing to specific records in tables. Both play crucial roles in database management, optimizing data access and query performance.

7.7 Self-Assessment Questions

1. Write Syntax of view based on single table
2. Write Syntax of view based on multiple table
3. Remove a view
4. Insert rows in view
5. Write advantages of Indexing
6. Write types of Index
7. Explain Views with Example

8. Explain DML operations on view with Example
9. Explain advantages of View.
10. Explain With check option,force and with read only.
11. When view is updatable.
12. Explain Indexes

7.8 References

1. Geeks for Geeks, W3School
2. Data analysis using SQL by Gordon Linoff
3. The Applied SQL Data Analytics by Benjamin Johnston, Matt Goldwasser, and Upom Malik
4. Learning SQL by Alan Beaulieu
5. Data Analytics: Data Science for Beginners, SQL Computer Programming for Beginners, Statistics for Beginners by Matt Foster

Chapter 8:
WINDOWS FUNCTIONS

Learning Outcomes:

- Students will learn about the SQL Window Functions.
- Students will learn concept of Windows Function.
- Students will learn Window Functions in SQL.

Structure:

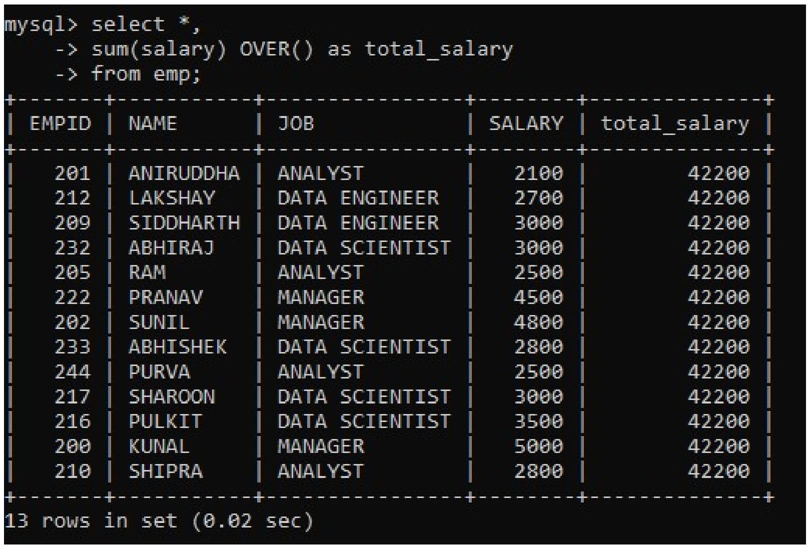
8.1 Introduction:

Window functions in SQL are instrumental for data analysis and predictive analytics, performing calculations across sets of table rows similar to aggregate functions. Unlike aggregate functions, window functions maintain the individual identities of rows without consolidating them into a single output. They are defined as SQL functions and are characterized by the presence of an OVER clause. This clause is essential for specifying how the window function should operate over a set of rows. Additionally, window functions may include a FILTER clause, offering further flexibility in data manipulation. These functions are initiated with the "OVER" clause and are configured using three key concepts:

- "window partition (PARTITION BY) - Form rows partitions"
- "window ordering (ORDER BY) - Defines sequence of rows in every window"
- "window frame (ROWS) - Defines the window by use of an offset from the specified row"

Format of Window functions: When performing calculations, rows retain their original identities and each row yields the computed result.

Window Function's Over Clause For instance, here is how it would appear if I were to show each employee's total compensation along with each row value:



```
mysql> select *,
    -> sum(salary) OVER() as total_salary
    -> from emp;
+-------+-----------+----------------+--------+--------------+
| EMPID | NAME      | JOB            | SALARY | total_salary |
+-------+-----------+----------------+--------+--------------+
|   201 | ANIRUDDHA | ANALYST        |   2100 |        42200 |
|   212 | LAKSHAY   | DATA ENGINEER  |   2700 |        42200 |
|   209 | SIDDHARTH | DATA ENGINEER  |   3000 |        42200 |
|   232 | ABHIRAJ   | DATA SCIENTIST |   3000 |        42200 |
|   205 | RAM       | ANALYST        |   2500 |        42200 |
|   222 | PRANAV    | MANAGER        |   4500 |        42200 |
|   202 | SUNIL     | MANAGER        |   4800 |        42200 |
|   233 | ABHISHEK  | DATA SCIENTIST |   2800 |        42200 |
|   244 | PURVA     | ANALYST        |   2500 |        42200 |
|   217 | SHAROON   | DATA SCIENTIST |   3000 |        42200 |
|   216 | PULKIT    | DATA SCIENTIST |   3500 |        42200 |
|   200 | KUNAL     | MANAGER        |   5000 |        42200 |
|   210 | SHIPRA    | ANALYST        |   2800 |        42200 |
+-------+-----------+----------------+--------+--------------+
13 rows in set (0.02 sec)
```

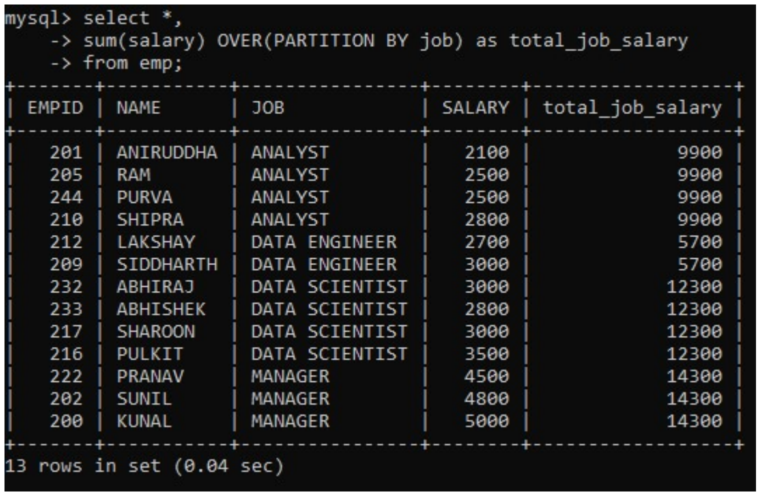
On rows, a window function is applied. Either aggregate functions or non-aggregate functions can be utilized with it.

Syntax:

"window_function_name(<expression>) OVER ( )"

➢ PARTITION BY:

When using "OVER" clause, "PARTITION BY" clause is used. It divided rows into several partitions based on function of window.



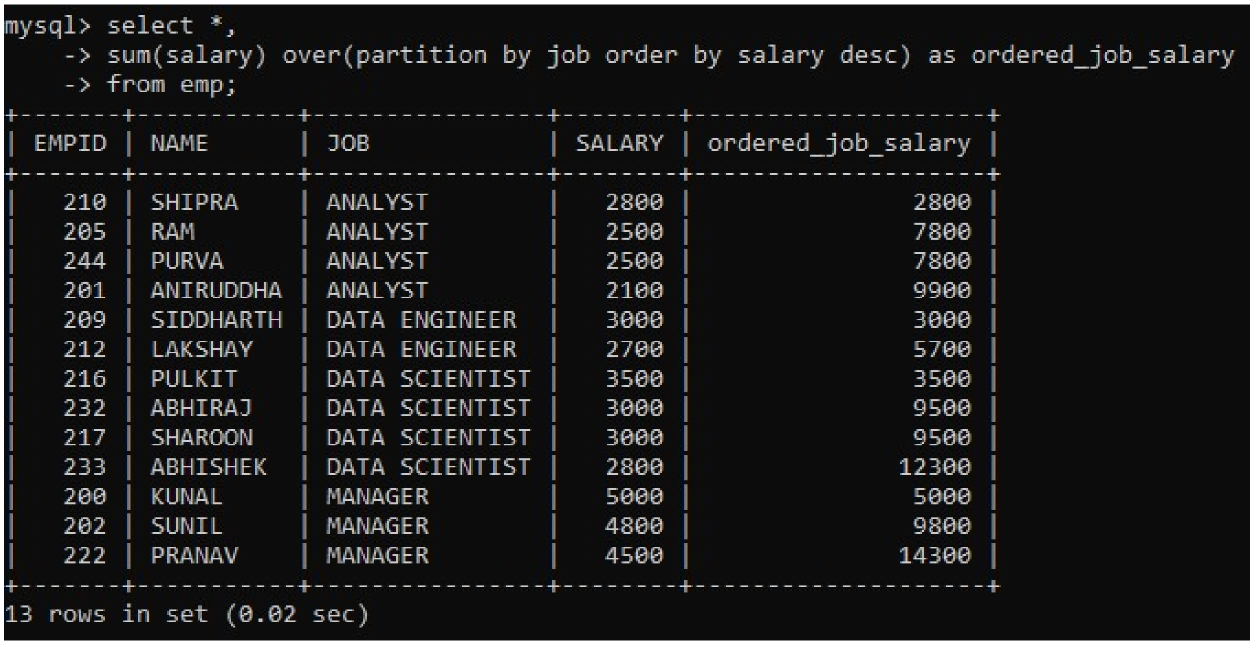
```
mysql> select *,
    -> sum(salary) OVER(PARTITION BY job) as total_job_salary
    -> from emp;
+-------+-----------+----------------+--------+------------------+
| EMPID | NAME      | JOB            | SALARY | total_job_salary |
+-------+-----------+----------------+--------+------------------+
|   201 | ANIRUDDHA | ANALYST        |   2100 |             9900 |
|   205 | RAM       | ANALYST        |   2500 |             9900 |
|   244 | PURVA     | ANALYST        |   2500 |             9900 |
|   210 | SHIPRA    | ANALYST        |   2800 |             9900 |
|   212 | LAKSHAY   | DATA ENGINEER  |   2700 |             5700 |
|   209 | SIDDHARTH | DATA ENGINEER  |   3000 |             5700 |
|   232 | ABHIRAJ   | DATA SCIENTIST |   3000 |            12300 |
|   233 | ABHISHEK  | DATA SCIENTIST |   2800 |            12300 |
|   217 | SHAROON   | DATA SCIENTIST |   3000 |            12300 |
|   216 | PULKIT    | DATA SCIENTIST |   3500 |            12300 |
|   222 | PRANAV    | MANAGER        |   4500 |            14300 |
|   202 | SUNIL     | MANAGER        |   4800 |            14300 |
|   200 | KUNAL     | MANAGER        |   5000 |            14300 |
+-------+-----------+----------------+--------+------------------+
13 rows in set (0.04 sec)
```

defining window function syntax for partition of rows:

"window_function_name(<expression>) OVER (<partition_by_clause>)"

Arranging Rows within Partitions

We are aware that the ORDER BY clause can be used to organize rows in a table. As a result, we must add ORDER BY clause to the OVER clause in order to arrange rows within each partition.



```
mysql> select *,
    -> sum(salary) over(partition by job order by salary desc) as ordered_job_salary
    -> from emp;
+-------+-----------+----------------+--------+--------------------+
| EMPID | NAME      | JOB            | SALARY | ordered_job_salary |
+-------+-----------+----------------+--------+--------------------+
|   210 | SHIPRA    | ANALYST        |   2800 |               2800 |
|   205 | RAM       | ANALYST        |   2500 |               7800 |
|   244 | PURVA     | ANALYST        |   2500 |               7800 |
|   201 | ANIRUDDHA | ANALYST        |   2100 |               9900 |
|   209 | SIDDHARTH | DATA ENGINEER  |   3000 |               3000 |
|   212 | LAKSHAY   | DATA ENGINEER  |   2700 |               5700 |
|   216 | PULKIT    | DATA SCIENTIST |   3500 |               3500 |
|   232 | ABHIRAJ   | DATA SCIENTIST |   3000 |               9500 |
|   217 | SHAROON   | DATA SCIENTIST |   3000 |               9500 |
|   233 | ABHISHEK  | DATA SCIENTIST |   2800 |              12300 |
|   200 | KUNAL     | MANAGER        |   5000 |               5000 |
|   202 | SUNIL     | MANAGER        |   4800 |               9800 |
|   222 | PRANAV    | MANAGER        |   4500 |              14300 |
+-------+-----------+----------------+--------+--------------------+
13 rows in set (0.02 sec)
```

rows are partitioned according to job category shown by JOB column. SALARY column has descending order and the *“ordered_job_salary”* column shows the running total of the job category

Syntax:

“window_function_name(<expression>) OVER (<partition_by_clause><order_clause>)”

Windowing syntax

“SUM(durationseconds)”, looks like any other aggregation. Adding “OVER” designates it as a window function. You could read the above aggregation as "take the sum of durationseconds *over* the entire result set, in order by start_time."

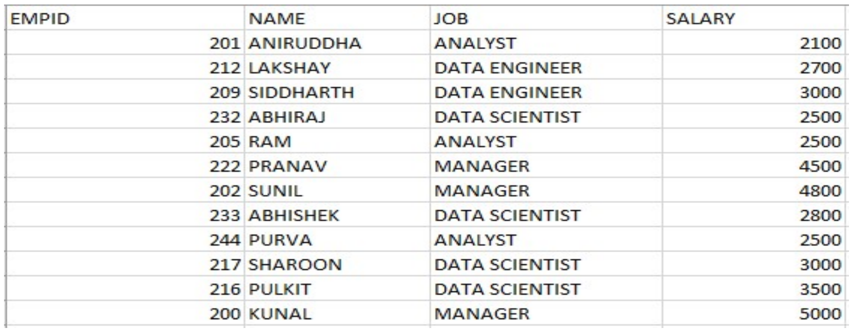
use PARTITION BY

```
SELECTstartterminal,
durationseconds,    SUM(duration_seconds) OVER
        (PARTITIONBYstart_terminalORDERBYstart_time)
ASrunning_total
FROMbikeshare
WHEREstart_time<'2010-01-07'
```

By using start_terminal, this command groups and arranges the query. Start_time is used to arrange each value of start_terminal, and the running total is the sum of duration_seconds for current row and all previous rows.

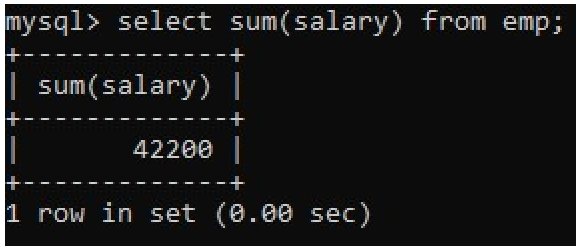
Dataset:

Assume that there is a business that keeps the following records on an employee: name, position, and pay.

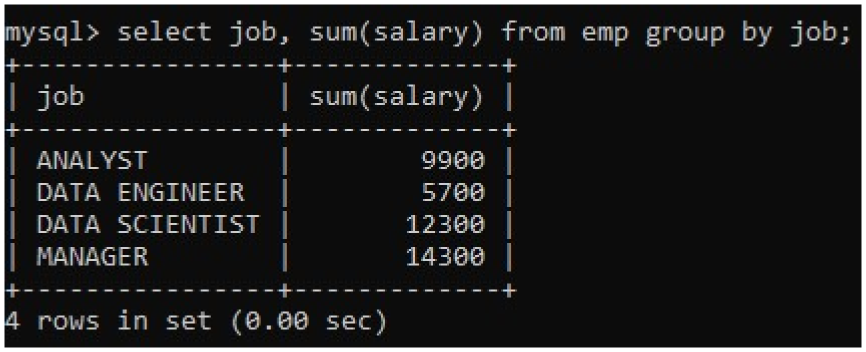


EMPID	NAME	JOB	SALARY
201	ANIRUDDHA	ANALYST	2100
212	LAKSHAY	DATA ENGINEER	2700
209	SIDDHARTH	DATA ENGINEER	3000
232	ABHIRAJ	DATA SCIENTIST	2500
205	RAM	ANALYST	2500
222	PRANAV	MANAGER	4500
202	SUNIL	MANAGER	4800
233	ABHISHEK	DATA SCIENTIST	2800
244	PURVA	ANALYST	2500
217	SHAROON	DATA SCIENTIST	3000
216	PULKIT	DATA SCIENTIST	3500
200	KUNAL	MANAGER	5000

Aggregate functions lag at:
to determine total wage of every worker in the organization. Apply the SUM() aggregate function to SALARY column.


```
mysql> select sum(salary) from emp;
+-------------+
| sum(salary) |
+-------------+
|       42200 |
+-------------+
1 row in set (0.00 sec)
```

To determine total salary of workers for each job type write,


```
mysql> select job, sum(salary) from emp group by job;
+----------------+-------------+
| job            | sum(salary) |
+----------------+-------------+
| ANALYST        |        9900 |
| DATA ENGINEER  |        5700 |
| DATA SCIENTIST |       12300 |
| MANAGER        |       14300 |
+----------------+-------------+
4 rows in set (0.00 sec)
```

When we aggregate a collection of rows together, aggregate functions provide us with a single value. However, these functions are not easily applicable to the latter queries. to preserve the unique identities of the distinct rows, something that functions are unable to accomplish. To achieve the same, use Window functions.

two types of window functions;

- “Built-in function” -row_number, dense_rank, lag, and lead;
- “Aggregation” - usual aggregation function.

Window function find the cumulative sums, moving average, ranking etc

8.2 Window keyword:

“CREATE TABLE tab1(x INTEGER PRIMARY KEY, y TEXT);
INSERT INTO tab1 VALUES (10, 'aaa'), (2, 'ccc'), (3, 'bbb');

```
   x   | y   | row_number
-----------------------
--   1 | aaa | 1
--   2 | ccc | 3
--   3 | bbb | 2
--
```

SELECT x, y, row_number() OVER (ORDER BY y) AS row_number FROM t0 ORDER BY x;”

The row_number() window function assigns consecutive integers to each row in order of the "ORDER BY" clause within the window-defn . Note that this does not affect the order in which results are returned from the overall query. The order of the final output is still governed by the ORDER BY clause attached to the SELECT statement (in this case "ORDER BY x").

Named window-defn clauses may also be added to a SELECT statement using a WINDOW clause and then referred to by name within window function invocations. For example, the following SELECT statement contains two named window-defs clauses, "win1" and "win2":

“SELECT x, y, row_number() OVER **win1**, rank() OVER **win2**
FROM t0
WINDOW **win1** AS (ORDER BY y RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW),
win2 AS (PARTITION BY y ORDER BY x)
ORDER BY x;”

The “WINDOW” clause, when one is present, comes after any “HAVING” clause and before any ORDER BY.

If we want to find running total.

“SELECT CID,TITLE,FNAME,LNAME,GENDER,COUNT(*)OVER w AS TCUSTOMER OVER (PARTION BY GENDER ORDER BY CID AS TCUSTOMER SUM(CASE WHEN TITLE IS NOT NULL THEN 1 ELSE 0 END)
OVER(PARTION BY GENDER ORDER BY CID) AS TCUSTOMER TITLE FROM CUSTOMER ORDER BY CID”

This query is tedious to write so we can simplify it by window clause.

```
“SELECT  CID,TITLE,FNAME,LNAME,GENDER,COUNT(*)OVER  w  AS  TCUSTOMER
SUM(CASE WHEN TITLE IS NOT NULL THEN 1 ELSE 0 END) over w AS TCUSTOMER
TITLE  FROM CUSTOMER
WINDOW W AS(PARTION BY GENDER ORDER BY CID) ORDER BY CID;”
```

This query give same result but we don’t have to write long partition by and order by query for each window function. Instead we write window w

For window function, there is a significant keyword “over” . Once you notice “over” within a query, you can tell there is a window function with simplicity solve such complex problems. The running total is the most useful illustration of this:

```
“SELECTstart_terminal,
duration_seconds,
SUM(duration_seconds) OVER
        (PARTITIONBYstart_terminalORDERBYstart_time)
ASrunning_total
FROM tutorial.dc_bikeshare_q1_2012
WHEREstart_time<'2012-01-07'

SELECTduration_seconds,
SUM(duration_seconds) OVER (ORDERBYstart_time) ASrunning_total
FROM tutorial.dc_bikeshare_q1_2012”
```

This query creates an aggregation (running_total) without using GROUP BY.

Knowledge Check 1

State True or False

a. Window partition groups rows into partitions.

b. window ordering defines the order or sequence of rows.

c. Likerow_number, dense_rank, lag, are Built-in window functions;

Outcome Based Activity

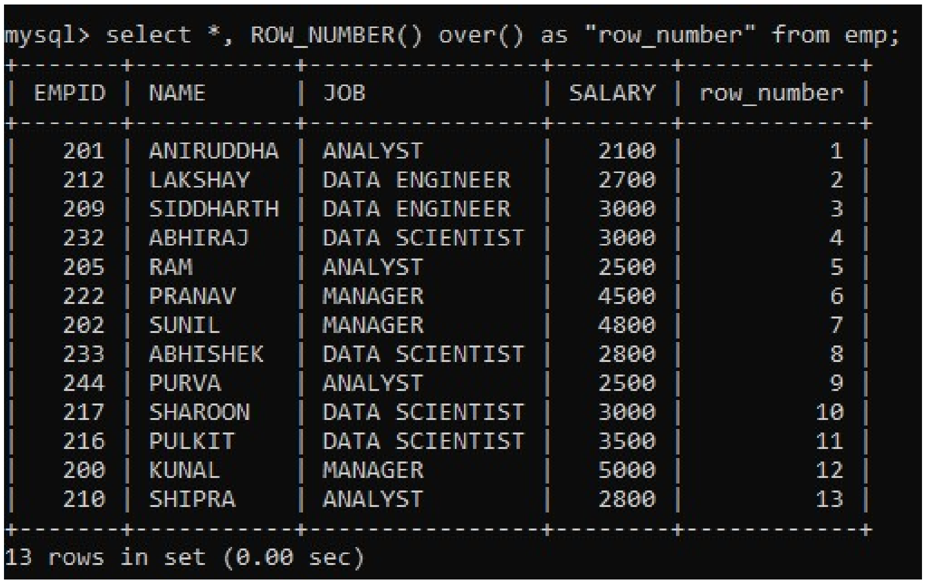
Find and list few windows functions with their functionality.

8.3Window Functions:

ROW_NUMBER()

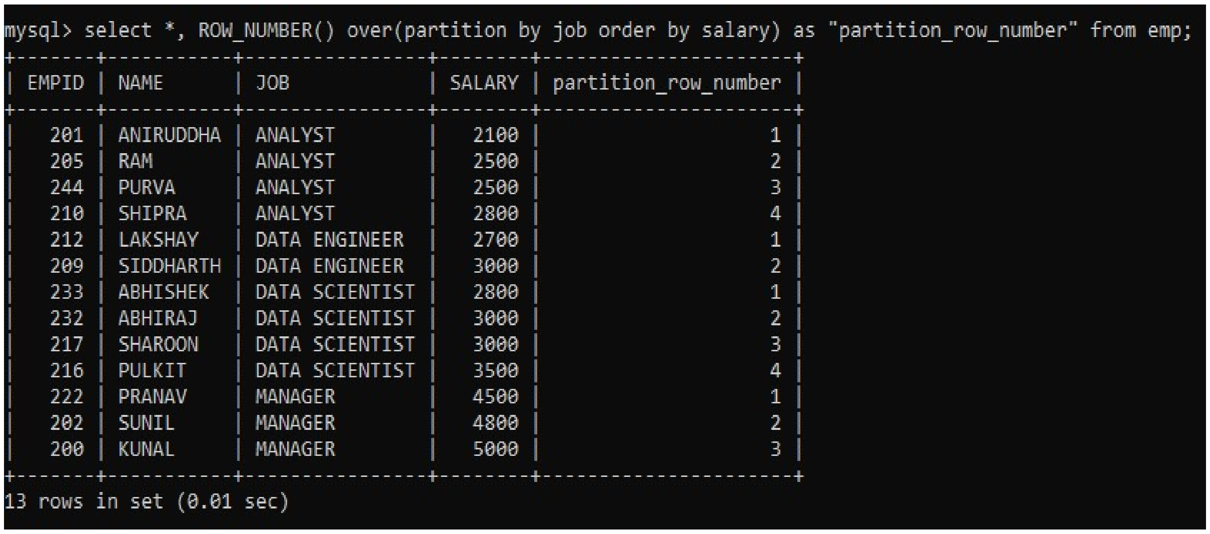
The number of a given row is displayed via “ROW_NUMBER()”. Starting at 1, it counts the rows in accordance with the window statement's “ORDER BY” clause.

If we don’t have sequential order of the rows, **ROW_NUMBER()** can be used togive a unique s number to each row of the table.



```
mysql> select *, ROW_NUMBER() over() as "row_number" from emp;
+-------+-----------+----------------+--------+------------+
| EMPID | NAME      | JOB            | SALARY | row_number |
+-------+-----------+----------------+--------+------------+
|   201 | ANIRUDDHA | ANALYST        |   2100 |          1 |
|   212 | LAKSHAY   | DATA ENGINEER  |   2700 |          2 |
|   209 | SIDDHARTH | DATA ENGINEER  |   3000 |          3 |
|   232 | ABHIRAJ   | DATA SCIENTIST |   3000 |          4 |
|   205 | RAM       | ANALYST        |   2500 |          5 |
|   222 | PRANAV    | MANAGER        |   4500 |          6 |
|   202 | SUNIL     | MANAGER        |   4800 |          7 |
|   233 | ABHISHEK  | DATA SCIENTIST |   2800 |          8 |
|   244 | PURVA     | ANALYST        |   2500 |          9 |
|   217 | SHAROON   | DATA SCIENTIST |   3000 |         10 |
|   216 | PULKIT    | DATA SCIENTIST |   3500 |         11 |
|   200 | KUNAL     | MANAGER        |   5000 |         12 |
|   210 | SHIPRA    | ANALYST        |   2800 |         13 |
+-------+-----------+----------------+--------+------------+
13 rows in set (0.00 sec)
```

It starts from 1 and so on.

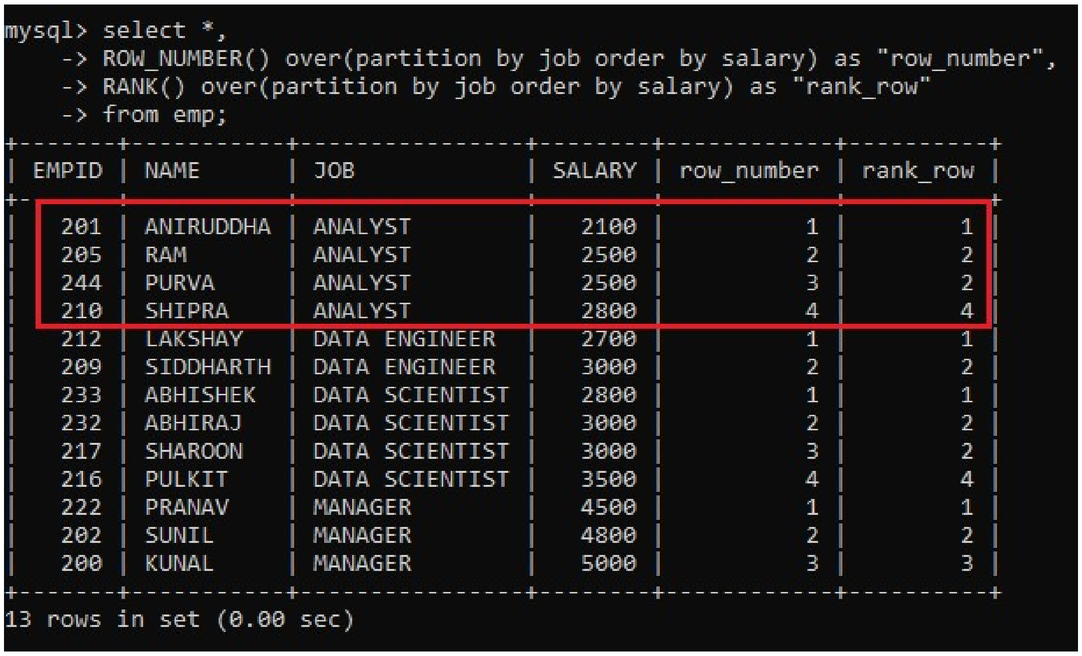


```
mysql> select *, ROW_NUMBER() over(partition by job order by salary) as "partition_row_number" from emp;
+-------+-----------+----------------+--------+----------------------+
| EMPID | NAME      | JOB            | SALARY | partition_row_number |
+-------+-----------+----------------+--------+----------------------+
|   201 | ANIRUDDHA | ANALYST        |   2100 |                    1 |
|   205 | RAM       | ANALYST        |   2500 |                    2 |
|   244 | PURVA     | ANALYST        |   2500 |                    3 |
|   210 | SHIPRA    | ANALYST        |   2800 |                    4 |
|   212 | LAKSHAY   | DATA ENGINEER  |   2700 |                    1 |
|   209 | SIDDHARTH | DATA ENGINEER  |   3000 |                    2 |
|   233 | ABHISHEK  | DATA SCIENTIST |   2800 |                    1 |
|   232 | ABHIRAJ   | DATA SCIENTIST |   3000 |                    2 |
|   217 | SHAROON   | DATA SCIENTIST |   3000 |                    3 |
|   216 | PULKIT    | DATA SCIENTIST |   3500 |                    4 |
|   222 | PRANAV    | MANAGER        |   4500 |                    1 |
|   202 | SUNIL     | MANAGER        |   4800 |                    2 |
|   200 | KUNAL     | MANAGER        |   5000 |                    3 |
+-------+-----------+----------------+--------+----------------------+
13 rows in set (0.01 sec)
```

Here, we have divided the rows according to the JOB column and arranged them in order of the employee's salary.

2. “Rank vs Dense_Rank”

The “**RANK()**” function, ranks rows within their partition on given condition.

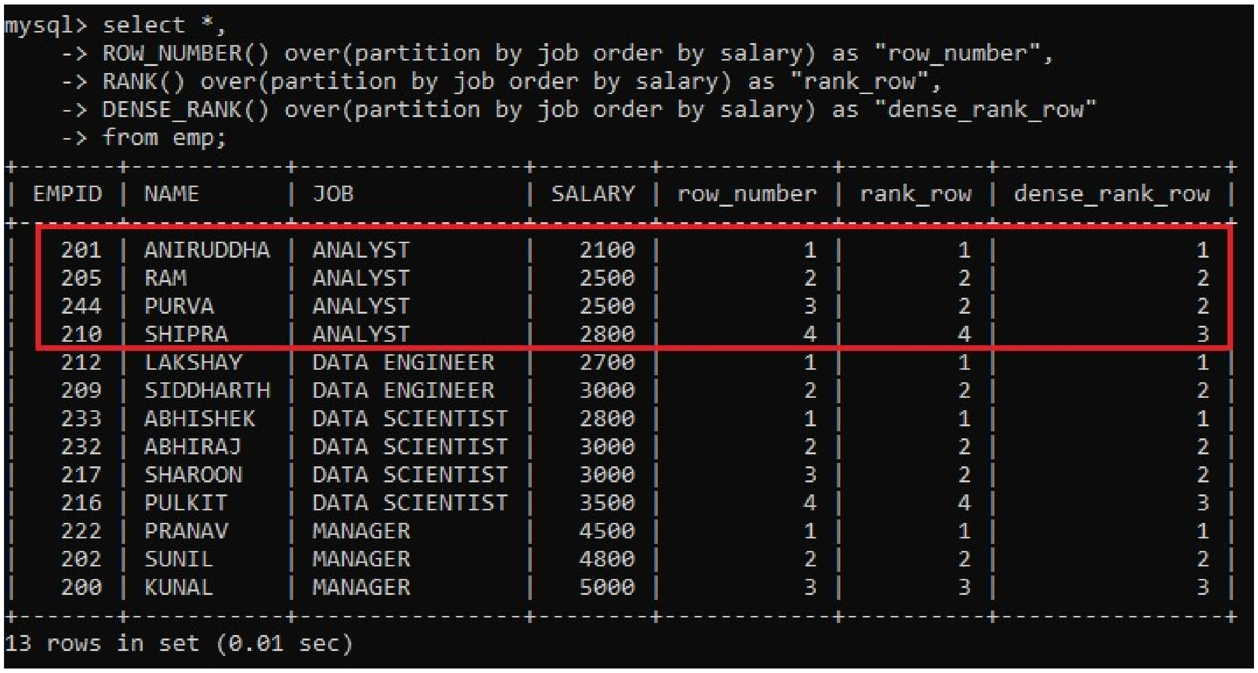


```
mysql> select *,
    -> ROW_NUMBER() over(partition by job order by salary) as "row_number",
    -> RANK() over(partition by job order by salary) as "rank_row"
    -> from emp;
+-------+-----------+----------------+--------+------------+----------+
| EMPID | NAME      | JOB            | SALARY | row_number | rank_row |
+-------+-----------+----------------+--------+------------+----------+
|   201 | ANIRUDDHA | ANALYST        |   2100 |          1 |        1 |
|   205 | RAM       | ANALYST        |   2500 |          2 |        2 |
|   244 | PURVA     | ANALYST        |   2500 |          3 |        2 |
|   210 | SHIPRA    | ANALYST        |   2800 |          4 |        4 |
|   212 | LAKSHAY   | DATA ENGINEER  |   2700 |          1 |        1 |
|   209 | SIDDHARTH | DATA ENGINEER  |   3000 |          2 |        2 |
|   233 | ABHISHEK  | DATA SCIENTIST |   2800 |          1 |        1 |
|   232 | ABHIRAJ   | DATA SCIENTIST |   3000 |          2 |        2 |
|   217 | SHAROON   | DATA SCIENTIST |   3000 |          3 |        2 |
|   216 | PULKIT    | DATA SCIENTIST |   3500 |          4 |        4 |
|   222 | PRANAV    | MANAGER        |   4500 |          1 |        1 |
|   202 | SUNIL     | MANAGER        |   4800 |          2 |        2 |
|   200 | KUNAL     | MANAGER        |   5000 |          3 |        3 |
+-------+-----------+----------------+--------+------------+----------+
13 rows in set (0.00 sec)
```

Use of “ROW_NUMBER()”, give sequential number. “RANK()”, we get same rank for rows with same value.

Similar-valued rows receive the same rank, and the rank that comes after it bypasses the missing rank. If we were asked to retrieve the "top N distinct" values from a table, this would not provide accurate results.

The “**DENSE_RANK()”** function is similar to “RANK()” except a difference, it doesn’t skip any ranks when ranking rows.



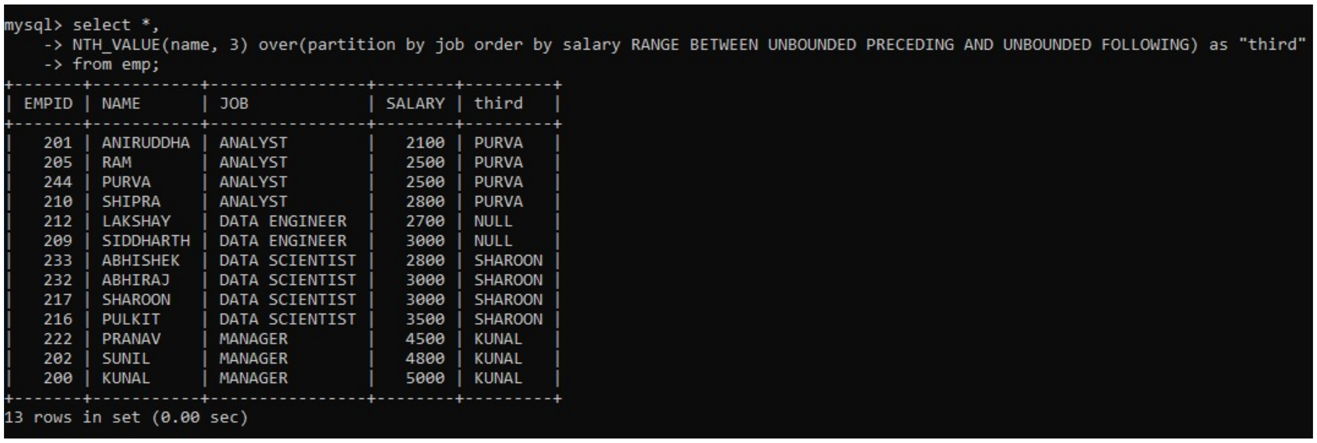
```
mysql> select *,
    -> ROW_NUMBER() over(partition by job order by salary) as "row_number",
    -> RANK() over(partition by job order by salary) as "rank_row",
    -> DENSE_RANK() over(partition by job order by salary) as "dense_rank_row"
    -> from emp;
+-------+-----------+----------------+--------+------------+----------+----------------+
| EMPID | NAME      | JOB            | SALARY | row_number | rank_row | dense_rank_row |
+-------+-----------+----------------+--------+------------+----------+----------------+
|   201 | ANIRUDDHA | ANALYST        |   2100 |          1 |        1 |              1 |
|   205 | RAM       | ANALYST        |   2500 |          2 |        2 |              2 |
|   244 | PURVA     | ANALYST        |   2500 |          3 |        2 |              2 |
|   210 | SHIPRA    | ANALYST        |   2800 |          4 |        4 |              3 |
|   212 | LAKSHAY   | DATA ENGINEER  |   2700 |          1 |        1 |              1 |
|   209 | SIDDHARTH | DATA ENGINEER  |   3000 |          2 |        2 |              2 |
|   233 | ABHISHEK  | DATA SCIENTIST |   2800 |          1 |        1 |              1 |
|   232 | ABHIRAJ   | DATA SCIENTIST |   3000 |          2 |        2 |              2 |
|   217 | SHAROON   | DATA SCIENTIST |   3000 |          3 |        2 |              2 |
|   216 | PULKIT    | DATA SCIENTIST |   3500 |          4 |        4 |              3 |
|   222 | PRANAV    | MANAGER        |   4500 |          1 |        1 |              1 |
|   202 | SUNIL     | MANAGER        |   4800 |          2 |        2 |              2 |
|   200 | KUNAL     | MANAGER        |   5000 |          3 |        3 |              3 |
+-------+-----------+----------------+--------+------------+----------+----------------+
13 rows in set (0.01 sec)
```

In this case, within each partition, every rank is separate and increases consecutively. It has not missed any ranks within a partition, in contrast to the "RANK()" function.

1. “Nth_Value”

To get the nth value from a window frame for an expression, use “NTH_VALUE(expression, N)” window function.

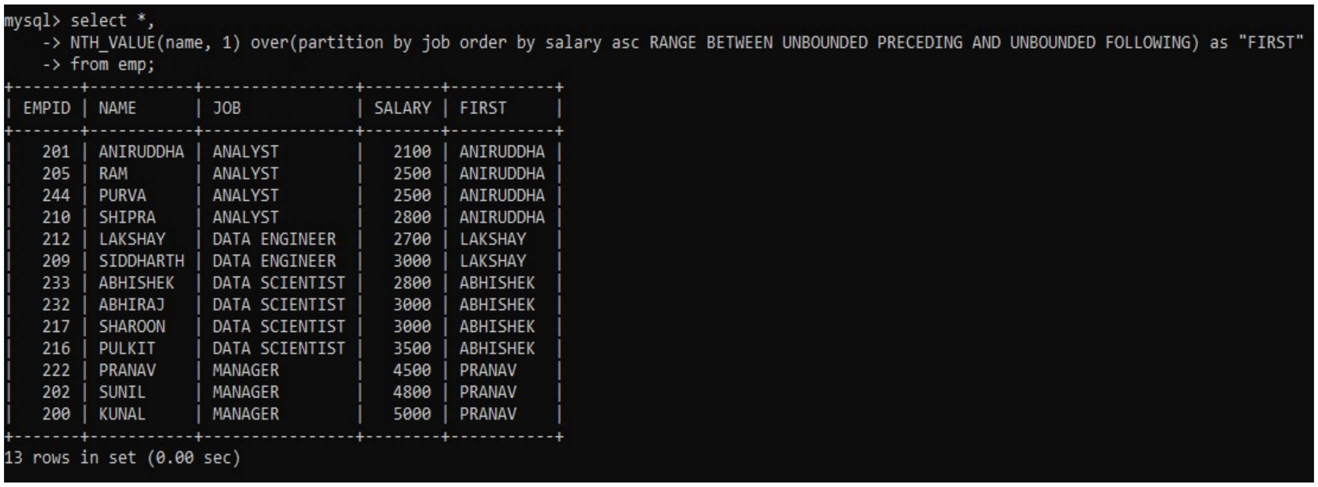
To find the third-highest pay in each JOB category, divide the rows according to the JOB column using the NTH_VALUE function. Then, sort the rows within the partitions in decreasing salary order.



```
mysql> select *,
    -> NTH_VALUE(name, 3) over(partition by job order by salary RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) as "third"
    -> from emp;
+-------+-----------+----------------+--------+---------+
| EMPID | NAME      | JOB            | SALARY | third   |
+-------+-----------+----------------+--------+---------+
|   201 | ANIRUDDHA | ANALYST        |   2100 | PURVA   |
|   205 | RAM       | ANALYST        |   2500 | PURVA   |
|   244 | PURVA     | ANALYST        |   2500 | PURVA   |
|   210 | SHIPRA    | ANALYST        |   2800 | PURVA   |
|   212 | LAKSHAY   | DATA ENGINEER  |   2700 | NULL    |
|   209 | SIDDHARTH | DATA ENGINEER  |   3000 | NULL    |
|   233 | ABHISHEK  | DATA SCIENTIST |   2800 | SHAROON |
|   232 | ABHIRAJ   | DATA SCIENTIST |   3000 | SHAROON |
|   217 | SHAROON   | DATA SCIENTIST |   3000 | SHAROON |
|   216 | PULKIT    | DATA SCIENTIST |   3500 | SHAROON |
|   222 | PRANAV    | MANAGER        |   4500 | KUNAL   |
|   202 | SUNIL     | MANAGER        |   4800 | KUNAL   |
|   200 | KUNAL     | MANAGER        |   5000 | KUNAL   |
+-------+-----------+----------------+--------+---------+
13 rows in set (0.00 sec)
```

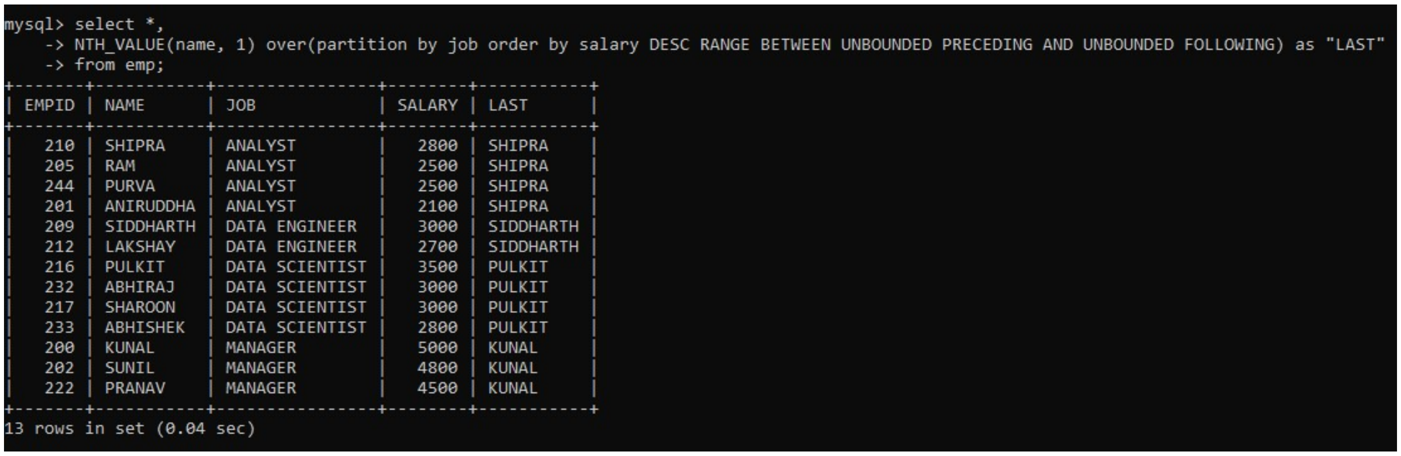
The frame clause specifies subset of a partition that a window function uses to compute value for current row. When a window function is applied, it considers all preceding and following rows relative to current row within frame. Most window functions operate over entire partition regardless of frame clause. However, NTH_VALUE() is an exception as it can operate on frames within a partition.

To output first value from each partition.



```
mysql> select *,
    -> NTH_VALUE(name, 1) over(partition by job order by salary asc RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) as "FIRST"
    -> from emp;
+-------+-----------+----------------+--------+-----------+
| EMPID | NAME      | JOB            | SALARY | FIRST     |
+-------+-----------+----------------+--------+-----------+
|   201 | ANIRUDDHA | ANALYST        |   2100 | ANIRUDDHA |
|   205 | RAM       | ANALYST        |   2500 | ANIRUDDHA |
|   244 | PURVA     | ANALYST        |   2500 | ANIRUDDHA |
|   210 | SHIPRA    | ANALYST        |   2800 | ANIRUDDHA |
|   212 | LAKSHAY   | DATA ENGINEER  |   2700 | LAKSHAY   |
|   209 | SIDDHARTH | DATA ENGINEER  |   3000 | LAKSHAY   |
|   233 | ABHISHEK  | DATA SCIENTIST |   2800 | ABHISHEK  |
|   232 | ABHIRAJ   | DATA SCIENTIST |   3000 | ABHISHEK  |
|   217 | SHAROON   | DATA SCIENTIST |   3000 | ABHISHEK  |
|   216 | PULKIT    | DATA SCIENTIST |   3500 | ABHISHEK  |
|   222 | PRANAV    | MANAGER        |   4500 | PRANAV    |
|   202 | SUNIL     | MANAGER        |   4800 | PRANAV    |
|   200 | KUNAL     | MANAGER        |   5000 | PRANAV    |
+-------+-----------+----------------+--------+-----------+
13 rows in set (0.00 sec)
```

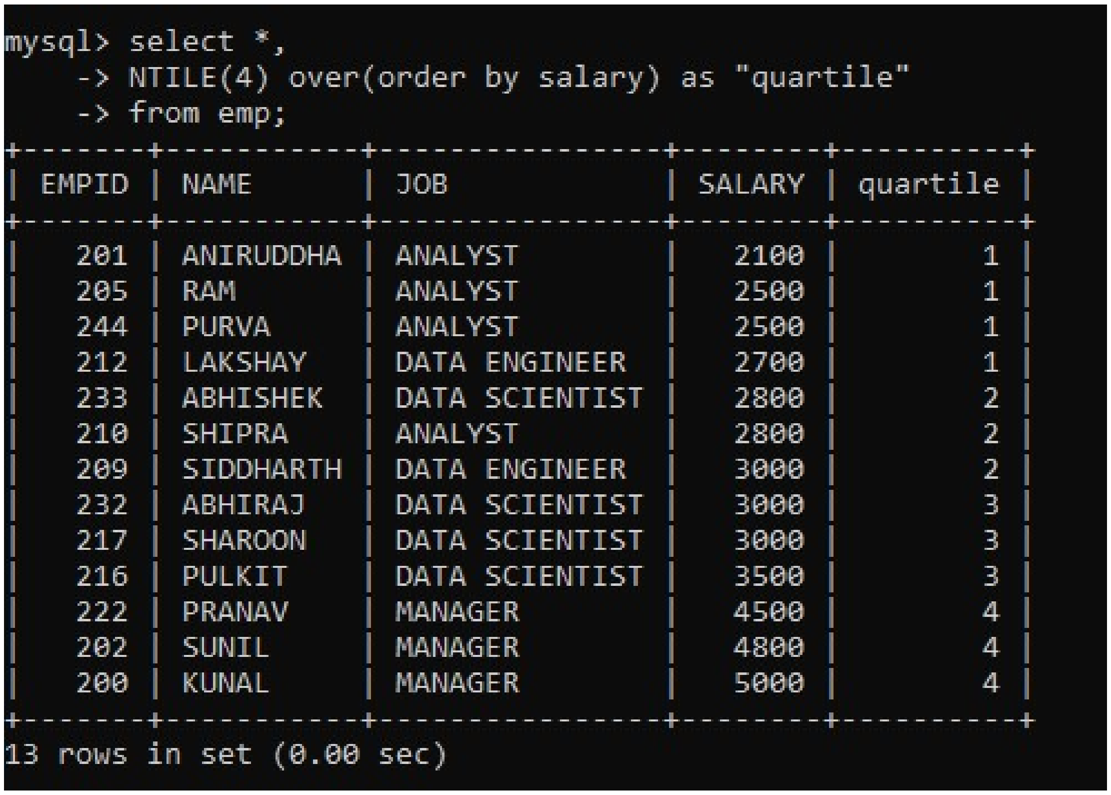
Similarly, the “LAST_VALUE()” function can be used to find last value within each partition, using rows ordered in descending order.



```
mysql> select *,
    -> NTH_VALUE(name, 1) over(partition by job order by salary DESC RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) as "LAST"
    -> from emp;
+-------+-----------+----------------+--------+-----------+
| EMPID | NAME      | JOB            | SALARY | LAST      |
+-------+-----------+----------------+--------+-----------+
|   210 | SHIPRA    | ANALYST        |   2800 | SHIPRA    |
|   205 | RAM       | ANALYST        |   2500 | SHIPRA    |
|   244 | PURVA     | ANALYST        |   2500 | SHIPRA    |
|   201 | ANIRUDDHA | ANALYST        |   2100 | SHIPRA    |
|   209 | SIDDHARTH | DATA ENGINEER  |   3000 | SIDDHARTH |
|   212 | LAKSHAY   | DATA ENGINEER  |   2700 | SIDDHARTH |
|   216 | PULKIT    | DATA SCIENTIST |   3500 | PULKIT    |
|   232 | ABHIRAJ   | DATA SCIENTIST |   3000 | PULKIT    |
|   217 | SHAROON   | DATA SCIENTIST |   3000 | PULKIT    |
|   233 | ABHISHEK  | DATA SCIENTIST |   2800 | PULKIT    |
|   200 | KUNAL     | MANAGER        |   5000 | KUNAL     |
|   202 | SUNIL     | MANAGER        |   4800 | KUNAL     |
|   222 | PRANAV    | MANAGER        |   4500 | KUNAL     |
+-------+-----------+----------------+--------+-----------+
13 rows in set (0.04 sec)
```

4. Ntile

To divide rows within a partition into a specific number of groups, “NTILE()” function is used. This function returns group number for each row within partition. Example: Determine quartile for every row according to employee’s salary:

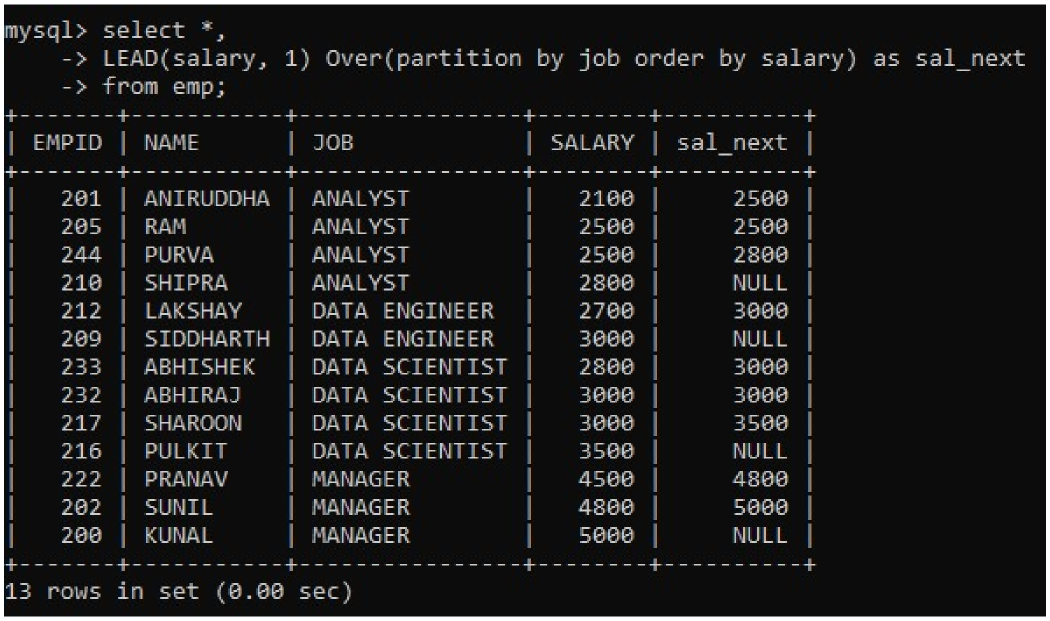


```
mysql> select *,
    -> NTILE(4) over(order by salary) as "quartile"
    -> from emp;
+-------+-----------+----------------+--------+----------+
| EMPID | NAME      | JOB            | SALARY | quartile |
+-------+-----------+----------------+--------+----------+
|   201 | ANIRUDDHA | ANALYST        |   2100 |        1 |
|   205 | RAM       | ANALYST        |   2500 |        1 |
|   244 | PURVA     | ANALYST        |   2500 |        1 |
|   212 | LAKSHAY   | DATA ENGINEER  |   2700 |        1 |
|   233 | ABHISHEK  | DATA SCIENTIST |   2800 |        2 |
|   210 | SHIPRA    | ANALYST        |   2800 |        2 |
|   209 | SIDDHARTH | DATA ENGINEER  |   3000 |        2 |
|   232 | ABHIRAJ   | DATA SCIENTIST |   3000 |        3 |
|   217 | SHAROON   | DATA SCIENTIST |   3000 |        3 |
|   216 | PULKIT    | DATA SCIENTIST |   3500 |        3 |
|   222 | PRANAV    | MANAGER        |   4500 |        4 |
|   202 | SUNIL     | MANAGER        |   4800 |        4 |
|   200 | KUNAL     | MANAGER        |   5000 |        4 |
+-------+-----------+----------------+--------+----------+
13 rows in set (0.00 sec)
```

To divide the rows into different numbers of groups and calculate the NTILE for different partitions.

5. Lead and Lag

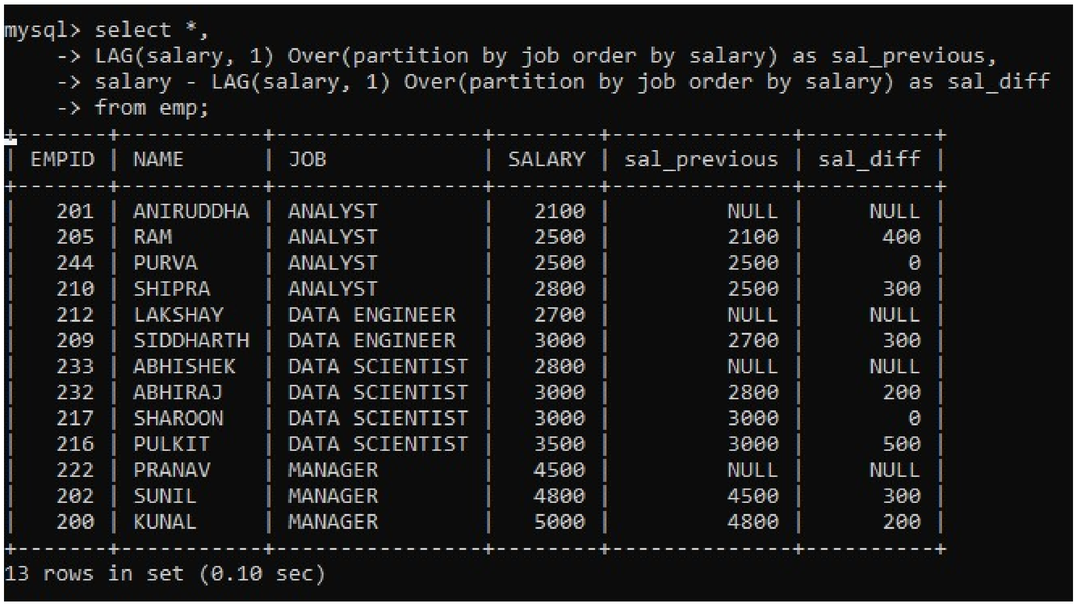
to analyze the data by comparing the value of the current row to that of the previous and next rows. For this reason, the window functions "LEAD() and LAG()" are present.



```
mysql> select *,
    -> LEAD(salary, 1) Over(partition by job order by salary) as sal_next
    -> from emp;
+-------+-----------+----------------+--------+----------+
| EMPID | NAME      | JOB            | SALARY | sal_next |
+-------+-----------+----------------+--------+----------+
|   201 | ANIRUDDHA | ANALYST        |   2100 |     2500 |
|   205 | RAM       | ANALYST        |   2500 |     2500 |
|   244 | PURVA     | ANALYST        |   2500 |     2800 |
|   210 | SHIPRA    | ANALYST        |   2800 |     NULL |
|   212 | LAKSHAY   | DATA ENGINEER  |   2700 |     3000 |
|   209 | SIDDHARTH | DATA ENGINEER  |   3000 |     NULL |
|   233 | ABHISHEK  | DATA SCIENTIST |   2800 |     3000 |
|   232 | ABHIRAJ   | DATA SCIENTIST |   3000 |     3000 |
|   217 | SHAROON   | DATA SCIENTIST |   3000 |     3500 |
|   216 | PULKIT    | DATA SCIENTIST |   3500 |     NULL |
|   222 | PRANAV    | MANAGER        |   4500 |     4800 |
|   202 | SUNIL     | MANAGER        |   4800 |     5000 |
|   200 | KUNAL     | MANAGER        |   5000 |     NULL |
+-------+-----------+----------------+--------+----------+
13 rows in set (0.00 sec)
```

SALARY from subsequent row in each partition, sorted by salary using "LEAD" function, is displayed in a new column. Each partition's final row contains a null value since there isn't another row for it to acquire data from.

LAG function:



```
mysql> select *,
    -> LAG(salary, 1) Over(partition by job order by salary) as sal_previous,
    -> salary - LAG(salary, 1) Over(partition by job order by salary) as sal_diff
    -> from emp;
+-------+-----------+----------------+--------+--------------+----------+
| EMPID | NAME      | JOB            | SALARY | sal_previous | sal_diff |
+-------+-----------+----------------+--------+--------------+----------+
|   201 | ANIRUDDHA | ANALYST        |   2100 |         NULL |     NULL |
|   205 | RAM       | ANALYST        |   2500 |         2100 |      400 |
|   244 | PURVA     | ANALYST        |   2500 |         2500 |        0 |
|   210 | SHIPRA    | ANALYST        |   2800 |         2500 |      300 |
|   212 | LAKSHAY   | DATA ENGINEER  |   2700 |         NULL |     NULL |
|   209 | SIDDHARTH | DATA ENGINEER  |   3000 |         2700 |      300 |
|   233 | ABHISHEK  | DATA SCIENTIST |   2800 |         NULL |     NULL |
|   232 | ABHIRAJ   | DATA SCIENTIST |   3000 |         2800 |      200 |
|   217 | SHAROON   | DATA SCIENTIST |   3000 |         3000 |        0 |
|   216 | PULKIT    | DATA SCIENTIST |   3500 |         3000 |      500 |
|   222 | PRANAV    | MANAGER        |   4500 |         NULL |     NULL |
|   202 | SUNIL     | MANAGER        |   4800 |         4500 |      300 |
|   200 | KUNAL     | MANAGER        |   5000 |         4800 |      200 |
+-------+-----------+----------------+--------+--------------+----------+
13 rows in set (0.10 sec)
```

Two new columns have been added. The first one lists the salary from the previous row for each partition sorted by salary. The second column is useful for analysis because it shows the difference in salary between the previous row and the current row.

Outcome Based Activity

What is effect of the below query.

“SELECT
athleteNm,eventNm,
ROW_NUMBER() OVER() AS Row_Number
FROM Medals
ORDER BY Row_Number ASC;”

8.4 Summary:

- Window functions are used for data analysis and predictive analytics within SQL.
- Unlike aggregate functions, window functions perform calculations across a set of related rows without collapsing them into a single output row.
- The distinguishing feature of window functions is the presence of an OVER clause in their syntax.
- Functions without an OVER clause are considered ordinary aggregate or scalar functions.
- Some window functions may include a “FILTER” clause between function and “OVER” clause.
- Examples of window functions include ROW_NUMBER(), LEAD(), LAG(), NTILE(), and RANK(), which ranks rows within their respective partitions based on specified conditions.

8.5 Self-Assessment Questions

1. Discuss Windows function.
2. Disucss Lead and Lag.
3. Explain Row_no().
4. Explain Partition by.
5. Explain OVER clause.

6. Discuss Syntax of Partition by.
7. Discuss Syntax of over.

8.6 References

1. https://www.analyticsvidhya.com/blog/2020/12/window-function-a-must-know-
2. https://towardsdatascience.com/why-window-function-in-sql-is-so-important-that-you-should-learn-it-right-now-1264b6096a76
3. https://www. https://www.tutorialspoint.com/
4. https://docs.oracle.com
5. Geeks for Geeks, W3School
6. Data analysis using SQL by Gordon Linoff
7. The Applied SQL Data Analytics by Benjamin Johnston, Matt Goldwasser, and Upom Malik
8. Learning SQL by Alan Beaulieu
9. The Applied SQL Data Analytics - Quick, Interactive Approach to Learning Analytics with SQL, 2nd Edition by Upom Malik, Matt Goldwasser, Benjamin Johnston
10. Data Analytics: Data Science for Beginners, SQL Computer Programming for Beginners, Statistics for Beginners by Matt Foster

Chapter 9:
IMPORTING EXPORTING DATA

Learning Outcomes

- Students will learn Concept of Importing and Exporting Data.
- Students will be able to understand Working of Copy function.

Structure

9.1 Introduction:

The import and export of data involve the input and output of datasets, typically achieved by converting data from one format to another using automated processes. Data exports preserve data in its raw format. Importing and exporting data often entails transferring datasets from one application to another. For instance, in a relational database management system (RDBMS), data may need to be backed up or new data imported from another database.

Database administrators typically manage import and export operations, although SQL developers should possess knowledge of relevant commands. Data can be imported from or exported to various formats, and tools are available to facilitate these tasks. Here, we will discuss data importation from and exportation to two common and straightforward formats that do not require additional tools or programs.

1. “SQL format”
2. “CSV format”

Exporting in SQL format:

To export a database in PostgreSQL, you typically create a .sql file containing a series of PostgreSQL statements and commands. These files may also be compressed to reduce storage size.

Export the library database.

pg_dump [connection-option...] [option...] [dbname]

To dump database mydb to sql file db

“$ pg_dump mydb > db.sql”

pg_dump is a utility used to create backups of PostgreSQL databases. These backups can be generated in two primary formats: script or archive files. Script dumps are plain-text files that include SQL commands necessary to recreate the structure and data of the database.

Importing from SQL format:

import the library.sql

“$ createdb library”

We now have an empty library database, and table users do not exist in it.

“$ psql -d library < library.sql”

1. CSV format:

“CSV (Comma-Separated Values)” is widely used as a format for exchanging data. When exporting data to a CSV file from a database, it involves selecting a specific subset of data based on a query rather than dumping all available data. This allows for targeted extraction of relevant information into a structured CSV format.

```
“COPY (SELECT u.username, b.title FROM users u
INNER JOIN users_books ub ON (ub.user_id = u.id)
INNER JOIN books b ON (b.id = ub.book_id))
TO '/tmp/users_books.csv'
WITH CSV;
$ cat /tmp/users_books.csv
John Smith,My First SQL book
John Smith,My Second SQL book
Jane Smiley,My Second SQL book”
```

Give an “absolute file path when using the COPY statement”, otherwise it will not work.

9.2 COPY COMMAND

The COPY command facilitates transferring data between a file and a database table. When specifying a list of columns with COPY, only data from or to those specific columns is transferred. During a COPY FROM operation, if any columns in the table are not included in the column list, default values will be inserted into those columns.

SYNTAX:

```
"COPY table_name [ ( column_name [, ...] ) ]
    FROM { 'filename' | PROGRAM 'command' | STDIN }
    [ [ WITH ] ( option [, ...] ) ]
    [ WHERE condition ]

COPY { table_name [ ( column_name [, ...] ) ] | ( query ) }
    TO { 'filename' | PROGRAM 'command' | STDOUT }
    [ [ WITH ] ( option [, ...] ) ]

where option can be one of:

    FORMAT format_name
    FREEZE [ boolean ]
    DELIMITER 'delimiter_character'
    NULL 'null_string'
    HEADER [ boolean ]
    QUOTE 'quote_character'
    ESCAPE 'escape_character'
    FORCE_QUOTE { ( column_name [, ...] ) | * }
    FORCE_NOT_NULL ( column_name [, ...] )
    FORCE_NULL ( column_name [, ...] )
    ENCODING 'encoding_name'  "
```

"table_name": existing table name.

"column_name": If column list is not specified, all columns of table will be copied.

“Query”: A select, values command whose results are to be copied.

“Filename”: The input or output file name.

“STDIN”: Here input comes from client application.

“STDOUT”: Here output goes to client application.

“Boolean”: To enable the option, you can use TRUE, ON, or 1. To disable it, use FALSE, OFF, or 0. If no Boolean value is specified, TRUE is assumed by default.

“FORMAT”: Selects data format to be read or written: text, csv or binary. The default is text.

“OIDS”: Provides copying OID for every row.

“**DELIMITER”**: The character that separates columns within each row of a file varies by format. In text format, the default separator is typically a tab, whereas in CSV (Comma-Separated Values) format, it is a comma.

“**NULL”**: Specifies string that represents null value. The default is “\N (backslash-N)” in text format, and an unquoted empty string in CSV format.

“HEADER”: names of each column in the file.

“QUOTE”: Specifies the character used for quoting data values in CSV format. By default, double-quote is used. This character must be a single byte.

“ESCAPE”: The character that comes before a data character that matches the QUOTE value is specified by this setting. It ensures that the quotation character doubles if it exists in the data by default, mirroring the QUOTE value. This configuration applies exclusively when using CSV format and requires a single one-byte character.

“FORCE_QUOTE”: Requires that all non-NULL values in each designated column be quoted.

“FORCE_NOT_NULL”: In the context of the specified columns, values should not be compared with an empty string. By default, when null string is empty, empty values will be treated as zero-length strings instead of nulls, even if they aren't enclosed in quotes. This option can only be used in COPY FROM operations, and only in the case of CSV format.

Where specifies condition.

ENCODING specifies that file encoding in the **“*encoding_name*”**.

WHERE condition- A “**WHERE”** condition is an expression in SQL that evaluates to a boolean result. Rows that do not meet this condition will not be inserted into table. A row satisfies condition if substituting its actual values for any variables referenced in the condition results in a true Boolean value.

The COPY command in PostgreSQL facilitates data transfer between database tables and files in the file system. When specifying a file name with COPY, PostgreSQL reads from or writes to the designated file directly. This file must be accessible to the PostgreSQL user. Alternatively, specifying PROGRAM allows the server to execute a specified command, reading from its standard output or writing to its standard input. This command must be executable from the server's perspective and accessible to the PostgreSQL user. Using STDIN or STDOUT indicates data transmission through the connection between client and server.

Outputs:

On successful completion, a COPY command returns a command tag of the form

“COPY ***count”***

The “*count”* is the number of rows copied.

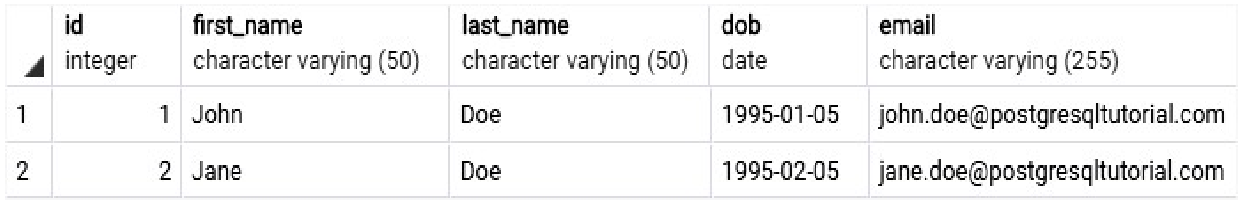
Few Facts:

1. COPY Command:
   - Function: The `COPY` command is used for bulk data transfer between files and database tables in PostgreSQL.
   - Permissions: Typically requires superuser or special permissions on the target table.
   - Performance: Optimized for efficient bulk data loading and unloading operations.
   - Direct Access: Operates directly on files accessible by the PostgreSQL server.
   - Syntax: Examples include `COPY table_name FROM 'filename'` for loading data into a table, and `COPY table_name TO 'filename'` for exporting data from a table.

2. \copy Command:
   - Function: The `\copy` command is a meta-command within the `psql` command-line tool for data transfer between files and database tables.

- Accessibility: Can be executed by database users with appropriate permissions on the target table.
- Flexibility: Operates within the `psql` client, enabling data transfer from client-side files.
- Syntax: Used in `psql` sessions with syntax like `\copy (SELECT * FROM table_name) TO 'filename'` to export query results to a file, or `\copy table_name FROM 'filename'` to import data into a table.

Example-
Consider table persons(id,First_name,last_name,dob,email)
Copy persons(first_name,last_name,dob,email) from ‘c:\sample\persons.csv
Delimiter ‘,’ csv header;
Select * from persons;



	id integer	first_name character varying (50)	last_name character varying (50)	dob date	email character varying (255)
1	1	John	Doe	1995-01-05	john.doe@postgresqltutorial.com
2	2	Jane	Doe	1995-02-05	jane.doe@postgresqltutorial.com

Specify table with column names after the “COPY” keyword. The order of columns must be same as ones in CSV file. for example:
“COPY sample_table_name
FROM 'C:\sampledb\sample_data.csv'
DELIMITER ','
CSV HEADER;”
Specify path to CSV file after “FROM” keyword in your PostgreSQL query. Since file is in CSV format, include appropriate “DELIMITER and CSV” clauses as needed. Use “HEADER” keyword to denote that CSV file includes a header row, although “COPY” command will disregard this header during data import.
For successful execution of the COPY command, ensure that the CSV file is accessible directly by the PostgreSQL server itself, rather than solely by the client application. This means the file must reside on a file system that PostgreSQL can access without intermediaries.

- **Export data using “COPY” statement:**

- The copy command is used to export table data to a CSV file.
- To export the persons table data to a CSV file named persondata.csv in the “C:\tmp folder”, use the following command:

“COPY persons TO 'C:\tmp\persondata.csv ' DELIMITER ',' CSV HEADER;”

PostgreSQL exports all data from all columns of “**persons”** table to “**persondata.csv”** file.

Let's say we wish to export information to a CSV file from a few columns in a table. The table name and column names are put after COPY keyword to do this. To exports data from **fname**, **email** columns of **persons** table to “**person_partial_db.csv”**

```
“COPY persons(first_name, email)
TO 'C:\tmp\persons2.csv' DELIMITER ',' CSV HEADER;”
```

In COPY statement, remove HEADER flag if you do not need to export header, which includes column names and table names. The command will be,

```
“COPY persons(email)
TO 'C:\tmp\personsemail.csv' DELIMITER ',' CSV;”
```

It indicates that database server computer, not your local computer, is where CSV file has to be located. Additionally, user that PostgreSQL server is running as must be able to write to CSV file.

- **Export data from a table to CSV file using “\copy” command:**

User should have rights to write to file, use “PostgreSQL” built-in command “**\copy” for the same.**

The “COPY” statement above is essentially executed by “\copy” command. Psql writes the CSV file instead of the server and moves data from server to your local file system. In order to utilize the “\copy” command, we must possess adequate privileges on your local machine. PostgreSQL super user rights are not necessary.

For instance, you can use “\copy” command from psql client to export all of data from the persons table into the personsclient.csv file.

“\copy (**SELECT * FROM** persons) **to**'C:\tmp\personsclient.csv' **with** csv”

This is how a “**COPY”** statement and “**\copy”** command to export data from a table to CSV files.

Knowledge check 2

State True or False

1. STDIN Specifies that input comes from the client application.
2. “FORCE_NOT_NULL” do not match the specified columns' values against the null string.
3. Format chooses between text, csv, or binary data formats to be read or written. Text is used by default.

Outcome Based Activity

What is effect of the above query.

```
“SELECT
  athlete,
  event,
ROW_NUMBER() OVER() AS Row_Number
FROM Summer_Medals
ORDER BY Row_Number ASC;”
```

9.3 Summary:

- Data import and export involve the transfer of datasets between applications.
- This process typically involves automated translation from one application's format to another's.
- Raw data exports often contain data in formats not directly readable by end-users.
- Importing and exporting data shares similarities with copying and pasting data between applications.
- The QUOTE clause in data handling specifies the character used for quoting data values, with the default being double-quote, a single one-byte character.

- The ESCAPE clause specifies the character used to escape instances where the QUOTE character appears within the data, defaulting to the same character as QUOTE to double it if it appears within the data.

9.4 Self-Assessment Questions

1. Explain Importing with example.
2. Explain Eexporting with example.
3. Explain copy command.
4. Define Import.
5. Define Export.
6. Discuss Syntax for copy.
7. Discuss Syntax for /copy.

9.5 References

1. https://www. https://www.tutorialspoint.com/
2. https://docs.oracle.com
3. Geeks for Geeks, W3School
4. Data analysis using SQL by Gordon Linoff
5. The Applied SQL Data Analytics by Benjamin Johnston, Matt Goldwasser, and Upom Malik
6. Learning SQL by Alan Beaulieu
7. The Applied SQL Data Analytics - Quick, Interactive Approach to Learning Analytics with SQL, 2nd Edition by Upom Malik, Matt Goldwasser, Benjamin Johnston
8. Data Analytics: Data Science for Beginners, SQL Computer Programming for Beginners, Statistics for Beginners by Matt Foster

Chapter 10:
Functions and Triggers

Learning Outcomes

- Students will learn basics and purpose of Functions
- Students will learn basics and purpose of Database Triggers
- Students will learn types of Triggers and uses of triggers.

Structure

10.1 Introduction:

Functions are subprograms that can optionally accept arguments and return values to the calling program. Events, on the other hand, are operations that trigger responses, often managed through triggers. A trigger is a statement declaring interest in specific events. Associating a function with a trigger enables the capturing and processing of these events. Row-level triggers execute for each affected row, while statement-level triggers execute once per SQL statement, also known as table-level triggers.

10.2 FUNCTIONS WITH AND WITHOUT ARGUMENT:

Functions are blocks of statements designed to perform specific tasks. They can return values upon completion. User-defined functions in SQL can accept parameters, execute calculations based on data retrieved via one or more “SELECT” statements, and directly return results to the invoking SQL code.

A crucial distinction is that user-defined functions must specify the return data type in their header section and explicitly include a return statement within the function body. Any code written after “RETURN” statement within the function body will not be executed.

Syntax “CREATE FUNCTION:

CREATE [**OR REPLACE**] **FUNCTION** function_name [parameters]**”**

“[(parameter_name [IN | OUT | IN OUT] type [, ...])]

RETURN return_datatype

{IS | AS}

BEGIN

< function_body >

END [function_name];”

Here in the syntax:

- **“Function_name”** is function name.
- **“[OR REPLACE]”** will replace existing function with new definition.
- The **optional parameter list** contains name, mode and types of the parameters.
- **“IN,OUT,INOUT” are different parameters modes.**
- The function must contain a “RETURN” statement.
- Function body contains the executable part.

Example : Function with argument:

“create” or replace “**function”** cube(n1 in number)

“return number

is

n3 number(7);

begin

n3 :=n1*n1*n1;

return n3;

end;

/

calling the function.

```
DECLARE
   n3 number(2);
BEGIN
   n3 := cube(5);
   dbms_output.put_line('Cube is: ' || n3);
END;
/”
```

Function without argument:

```
“CREATE OR REPLACE FUNCTION Allemp
RETURN number IS
   total number(2) := 0;
BEGIN
   SELECT count(*) into total
   FROM emp;
    RETURN total;
END;
/”
Function created.
```

Calling PL/SQL Function:

When a function is created, it contains a definition that specifies its functionality or what task it performs. To execute or use a function, you need to call it within your program. When a function is called, the program's control is transferred to that function to execute its defined task. After completing the task, control is returned back to the main program.

Calling a function involves specifying the function's name along with any arguments it requires. If the function returns a value, you can store this returned value in a variable for further use within your program.

Example-

```
“DECLARE
```

```
   c number(2);
BEGIN
   c := allemp();
   dbms_output.put_line('Total no. of emp: ' || c);
END;
/”
```

Following the above code's execution in SQL prompt, the following outcome will appear.

```
Total no. of emp: 4
```

1.

The parameter modes:

Parameter modes determine behaviors of parameters in SQL. There are three parameter modes: “IN, OUT, and INOUT”. If not explicitly specified, a parameter defaults to the IN mode.

- The IN mode

This is default parameter pass value to .This is read only. In parameter act like constant so new values can not be assigned to these parameters.

The function that follows locates a movie using its ID and returns its title:

- OUT parameter-**out parameter has to specify explicitly.This is Write only. *out* parameters act like uninitialized variables.Values must be assigned to them.**

Out parameters are included within the argument list and are returned as part of the result. They are particularly beneficial in functions that require multiple return values. To designate out parameters, you explicitly prefix the parameter name with the keyword 'out', like this:

- IN-OUT parameter-**This type mode is explicitly specified. This is Read and write both. Inout parameters act like an initialized variables.**

The inout mode combines both in and out modes in programming. This indicates that the caller can provide an argument to a function, which the function then modifies and returns with the updated value.

```
“DECLARE
   a number;
   b number;
   c number;
FUNCTION fMax(x IN number, y IN number)
RETURN number
IS
    z number;
BEGIN
   IF x > y THEN
      z:= x;
   ELSE
      Z:= y;
END IF;
   RETURN  z;
END;

BEGIN
   a:=23;
   b:=55;
   c := fMax(a, b);
   dbms_output.put_line(' Maximum of (23,55): '|| c);
END;
/”

  create or replace function “fun (a in number,b out number, c inout number” return number is
“begin
b:=a+10;
c:=c+a;
return(100);
end;
/”
Function created.
```

```
“declare
x number:=10;
y number:=20;
z number;
begin
z:=fun(x,z,y);
DBMS_OUTPUT.PUT_LINE('X'||X||'y'||y||'z'||z);
end;
/”
```

- Removing a user-defined function:

The basic syntax to remove a user-defined function is identical for all three databases:

“DROP FUNCTION <function_name>”

Similarly to stored procedures, Transact-SQL enables you to delete multiple functions using a single “DROP FUNCTION” statement.

Knowledge Check 1

Fill in the blanks

1. _______ is a declaration that you are interested in a certain event.
2. ___________ can accept parameters, perform specific calculations.
3. The parameter _______ decide the behaviors of parameters.

Outcome Based activity

List down the Library Functions which works with or without Argument.

10.3 TRIGGER:

A database trigger is a database object that can automatically invoke or execute actions in response to specific events, which are typically “DML (Data Manipulation Language)” commands like “INSERT, UPDATE, or DELETE”. These events are known as triggering events. Triggers are essentially stored procedures associated with these events.

Triggers cannot be explicitly called by users; instead, they are fired automatically when the defined event occurs. They can be enabled or disabled using the ALTER TRIGGER command.

When enabled, a trigger executes its defined actions when its triggering event happens. When disabled, it remains inactive and does not execute.

Triggers are created using "CREATE TRIGGER" statement and can be applied to tables, views, schemas, or the entire database. The timing of a trigger's execution can be specified as BEFORE or AFTER the triggering statement runs. Additionally, triggers can be configured to fire for each row affected by triggering statement (known as row-level triggers) or just once for entire statement (statement-level triggers).

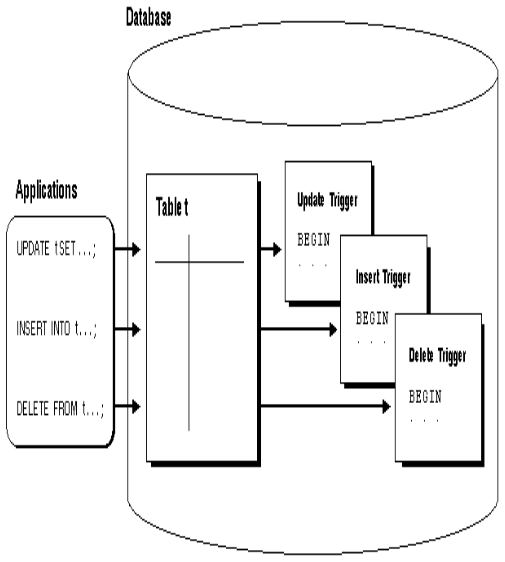


Figure 13.1 Database Triggers

CREATE TRIGGER:

" **CREATE** [OR REPLACE] **TRIGGER** trigger_name

{BEFORE | **AFTER** | **INSTEAD OF** }

{**INSERT** [OR] | **UPDATE** [OR] | **DELETE**}

[**OF** col_name]

ON table_name

[REFERENCING OLD **AS** o NEW **AS** n]

[**FOR** EACH ROW]

WHEN (condition)

DECLARE

Declaration-statements

BEGIN
  Executable-statements
EXCEPTION
  Exception-handling-statements
END; ”

“CREATE TRIGGER trigger_name”: It creates new trigger with trigger_name.

“[OR REPLACE]”: This will replace the existing trigger if triggername specified is same.

“{BEFORE | AFTER | INSTEAD OF}” : This is trigger timings tell when trigger will be executed. Before the DML . The “INSTEAD OF” clause is used for creating trigger on a view.

“{INSERT [OR] | UPDATE [OR] | DELETE}”: This specifies the DML operation.

“[OF col_name]”: This specifies column name that would be updated.

“[ON table_name]”: This specifies table name associated with trigger.

“[REFERENCING OLD AS o NEW AS n]”: This allows you to refer new and old values for various DML statements, like “INSERT, UPDATE, and DELETE”.

“[FOR EACH ROW]”: This indicates a row level trigger, meaning that each impacted row would have the trigger applied. If not, the trigger—also referred to as a table level trigger—will only run once when the SQL query is executed.

“WHEN (condition)”: This gives the rows that the trigger would fire a condition. This clause only applies to triggers at the row level.

Triggers can be broadly classified into “**Row Level”** and **“Statement Level”** triggers.

“Row-level triggers” execute once for each **“row”** in a transaction. “FOR EACH ROW” clause is present in “CREATE TRIGGER” command.

“Statement-level triggers” execute once for each transaction. A "statement-level trigger" on the Customer table, for instance, would only be triggered once if a single transaction added 500 rows to the table. For Statement-level triggers, the “CREATE TRIGGER” command lacks the "FOR EVERY ROW" clause.

Eg-

“ CREATE or REPLACE TRIGGER Before_Update_Stat_product
BEFORE UPDATE ON product

Begin

INSERT INTO table

Values('Before update, statement level',sysdate);

END; ”

“DML Triggers”: A DML trigger is designed for tables or views and activates in response to DELETE, INSERT, or UPDATE statements. Triggers can be categorized as either simple or compound based on their structure and functionality.

A “**simple DML trigger”** fires at exactly one of these timing points:

Before the triggering statement runs (**BEFORE** statement trigger)

After the triggering statement runs (**AFTER** statement trigger)

Before each row that the triggering statement affects (The trigger is called a **BEFORE** each row trigger)

After each row that the triggering statement affects (The trigger is called an **AFTER** each row trigger)

A **“compound DML trigger”** created on a table or view can activate at one or more of preceding timing points. These triggers are useful for coordinating actions across different timing points by sharing common data. When a row-level trigger fires, it can access specific data within row being processed using Correlation Names.

Triggers created on a schema or the entire database responds to events involving DDL statements or database operations. These triggers are categorized as system triggers due to their broader scope and ability to handle database-level events.

A **“conditional trigger”** includes a “**WHEN”** clause that defines a SQL condition evaluated by the database for each row affected by triggering statement. For further details regarding “WHEN” clause, please refer to the relevant documentation.

The “**INSTEAD OF” trigger** is either:

- A DML trigger can be created on either a non-editable view or a nested table column within a non-editable view.
- A system trigger is defined specifically on a “CREATE” statement.

In such cases, database activates “INSTEAD OF” trigger in place of executing triggering statement.

"System Triggers": **A system trigger can be created on either a schema or entire database. It activates based on specific triggering events, which can be either Data Definition Language statements or database operation statements.**

A system trigger fires at exactly one of these timing points:

- **Before triggering statement runs:** This type of trigger, known as a statement-level BEFORE trigger, executes before the triggering statement initiates.
- **After triggering statement runs:** Referred to as a statement-level AFTER trigger, this trigger executes after the triggering statement has completed its execution.
- **Instead of triggering CREATE statement:** Known as an "INSTEAD OF CREATE" trigger, this trigger executes in place of the triggering "CREATE" statement.

Triggers fire is in this order:

1. "All **BEFORE STATEMENT** triggers"
2. "All **BEFORE EACH ROW** triggers"
3. "All **AFTER EACH ROW** triggers"
4. "All **AFTER STATEMENT** triggers"

Uses of Triggers:

- Automatically generate values for virtual columns.
- Log events for auditing and tracking purposes.
- Collect statistics on table access to optimize performance.
- Enable DML statements issued against views to modify underlying table data.
- Enforce referential integrity across distributed databases.
- Restrict DML operations on a table outside of regular business hours.
- Ensure transactions adhere to validity constraints.
- Implement complex business or referential integrity rules that cannot be defined using standard constraints.

A trigger operates exclusively on new data. For instance, it can be employed to prevent a DML statement from inserting a "NULL" value into a database column. However, any NULL values existing in the column prior to defining the trigger or during periods when the trigger was inactive would remain unaffected.

Triggers serve several purposes in databases:

1. Audit Trail: Triggers can track changes to data, recording who made the change and when it occurred. This is crucial for maintaining an audit trail of database operations.

2. Enforcement of Business Rules: Triggers enforce business rules by automatically validating data modifications against predefined criteria. For example, ensuring that certain conditions are met before allowing an update or insert operation.

3. Complex Integrity Constraints: Triggers can enforce complex integrity constraints that cannot be easily defined using standard declarative constraints.

4. Logging and Error Handling: Triggers can log errors or exceptions that occur during data manipulation operations, providing insight into issues that may arise.

5. Synchronous Replication: Triggers can be used in synchronous replication scenarios to propagate changes immediately to other databases or systems.

6. Derived Data Maintenance: Triggers can maintain derived or calculated data that depends on other data within the database.

7. Security Enforcement: Triggers can enforce security policies, such as access control rules, by preventing unauthorized changes to data.

8. Automated Tasks: Triggers can automate tasks that need to be performed in response to specific events, reducing the need for manual intervention.

Conditional Predicates for Detecting Triggering DML Statement:

A DML trigger's triggering event may include more than one triggering statement. Conditional predicates let the trigger to decide the precise action when one of these assertions sets it off:

- **“INSERTING”:** Indicates that an “INSERT” statement fired the trigger.
- **“UPDATING”:** Indicates that an “UPDATE” statement fired the trigger.

- **“UPDATING ('column')”:** Specifies that an “UPDATE” statement affecting specified column fired trigger.
- **“DELETING”:** Indicates that a “DELETE” statement fired trigger.

Example :

“

```
CREATE OR REPLACE TRIGGER t1
 BEFORE
INSERT OR
UPDATE OF salary, deptid OR
DELETE
  ON emp
BEGIN
      WHEN INSERTING THEN
      DBMS_OUTPUT.PUT_LINE('Inserting');
    WHEN UPDATING('salary') THEN
      DBMS_OUTPUT.PUT_LINE('Updating salary');
    WHEN UPDATING('deptid') THEN
      DBMS_OUTPUT.PUT_LINE('Updating department ID');
    WHEN DELETING THEN
      DBMS_OUTPUT.PUT_LINE('Deleting');
  END;
/
```

”

Correlations:

Correlation names allow a row-level trigger to retrieve the data in the row it is currently processing. Trigger needs to refer value before and after DML statement. The default correlation names are : “OLD, :**NEW**. **OLD** and **NEW”** refers **Old and new values**.

Statement	Old value	New value
“INSERT”	“NULL”	“Post-insert value”
“UPDATE”	“Pre-update value”	“Post-update value”
“DELETE”	“Pre-delete value”	“NULL”

In SQL, an **AFTER** trigger does not modify **NEW** field values because it executes after the triggering statement completes. Attempting to alter **NEW** field values in an **AFTER** trigger will lead to an error.

Conversely, a **BEFORE** trigger can modify **NEW** field values before an **INSERT** or **UPDATE** statement inserts them into the table.

If a statement triggers both a **BEFORE** and an **AFTER** trigger, and the **BEFORE** trigger changes a **NEW** field value, the **AFTER** trigger will reflect this modified value.

Example:

Creates Trigger to insert row in log table after any **UPDATE** of the **SALARY** column of the **EMP** table.

Create trigger that inserts row in log table after **employees salary** is updated:

```
“CREATE OR REPLACE TRIGGER salary_up
  AFTER UPDATE OF salary ON emp
  FOR EACH ROW
BEGIN
  INSERT INTO Emplog (Emp_id, ldate, New_salary, Action)
  VALUES (:NEW.empid, SYSDATE, :NEW.salary, 'New Salary');
END;
/”
```

Update EMPLOYEES SALARY:

```
“UPDATE emp
SET salary = salary + 1050
WHERE Department_id = 20;”
```

Example: creates a trigger that prints salary on a DELETE, INSERT, or UPDATE on EMPLOYEES table.

```
“CREATE OR REPLACE TRIGGER salary_changes
  BEFORE DELETE OR INSERT OR UPDATE ON employees
  FOR EACH ROW
```
WHEN (NEW.job_id <> 'PRES')
```
DECLARE
  sal_diff  NUMBER;
BEGIN
  sal_diff  := :NEW.salary  - :OLD.salary;
  DBMS_OUTPUT.PUT(:NEW.last_name || ': ');
  DBMS_OUTPUT.PUT('Old salary = ' || :OLD.salary || ', ');
  DBMS_OUTPUT.PUT('New salary = ' || :NEW.salary || ', ');
  DBMS_OUTPUT.PUT_LINE('Difference: ' || sal_diff);
END;
/”
```

Query:

```
“SELECT last_name, department_id, salary, job_id
FROM employees
WHERE department_id IN (10, 20, 90)
ORDER BY department_id, last_name;
”
```

Rows selected.

Triggering statement:

```
“UPDATE employees
SET salary = salary * 1.05
WHERE department_id IN (10, 20, 90);
”
```

Example INSTEAD OF Trigger

Consider tables,

emp(emp_no,ename salary mgr,deptno)

dept(Deptno, Dept_name, add)

Create a view

```
“CREATE VIEW emp_view(
Emp_name,dept_name,add) AS
SELECT emp.ename,dept.dept_name,dept.add
FROM emp,depdep.
WHERE emp.deptno=dept.deptno;
/”
```

If we update view, error is due. To avoid the error in updating view in the we use "instead of trigger."

```
“CREATE TRIGGER view_modify_trg
INSTEAD OF UPDATE
ON emp_view
FOR EACH ROW
BEGIN
UPDATE dept
SET add=:new.add
WHERE deptname=:old.deptname;
END;”
```

New error is handled by instead of trigger and updating of view is done.

Example: Trigger Derives New Column Values:

```
“CREATE OR REPLACE TRIGGER Derived
BEFORE INSERT OR UPDATE OF Ename ON Emp
```

```
/* Before updating the ENAME field, derive the values for
   the UPPERNAME and SOUNDEXNAME fields. Restrict users
   from updating these fields directly: */
FOR EACH ROW
BEGIN
  :NEW.Uppername := UPPER(:NEW.Ename);
  :NEW.Soundexname := SOUNDEX(:NEW.Ename);
END;
/”
```

“Trigger Enabling and Disabling”:

By default, when you use “CREATE TRIGGER” statement, trigger is created in an enabled state. However, if you specify “DISABLE”, trigger will be created in a disabled state instead. This allows you to ensure that trigger compiles correctly without errors before enabling it.

There are several reasons why you might want to temporarily disable a trigger:
- The trigger references an object that is currently unavailable or undergoing maintenance.
- Large data loads must be completed rapidly and without the need for triggers to fire.

- During a data reload into the database, you may need to temporarily suspend trigger actions to expedite the process.

To enable or disable a single trigger:
“ALTER TRIGGER [*schema*.]*trigger_name* { ENABLE | DISABLE };”

To enable or disable all triggers created on a specific table:
“ALTER TABLE *table_name* { ENABLE | DISABLE } ALL TRIGGERS;”

Knowledge Check 2

State True or False

1. To enable or disable a single trigger, ALTER TRIGGERcommand is used.
2. CREATE TRIGGER statement creates a trigger in the Disabled state.

3. This is trigger timings tell when trigger will be executed.
4. Triggers prevent invalid transactions

Outcome Based Activity

Create a trigger after insert on employee table.

10.4 Summary:

When creating a function, it is essential to define its specific tasks or operations, which are executed upon invocation. Invoking a function transitions program control to the function itself. Upon completing its tasks, the function returns program control to the main program. Parameters required by the function must be passed along with its name during invocation, and if the function returns a value, this can be captured and stored.

Database triggers are objects that automatically execute in response to specific events like "INSERT, UPDATE, or DELETE" operations. Triggers cannot be explicitly invoked but are triggered upon occurrence of defined events when enabled. They can be enabled or disabled using "ALTER TRIGGER" command. Triggers are created using "CREATE TRIGGER" statement and can be applied to tables, views, or schemas.

10.5 Self-Assessment Questions

1. Define Functions. Explain with Example.
2. Define Triggers. Explain with example.
3. Explain parameter modes in functions.
4. Explain Row level and statement level Trigger.
5. Explain references or correlations in triggers.
6. Explain types of triggers.

10.6 References

1. https://www. https://www.tutorialspoint.com/
2. https://docs.oracle.com
3. Geeks for Geeks, W3School
4. Data analysis using SQL by Gordon Linoff

5. The Applied SQL Data Analytics by Benjamin Johnston, Matt Goldwasser, and Upom Malik
6. Learning SQL by Alan Beaulieu
7. The Applied SQL Data Analytics - Quick, Interactive Approach to Learning Analytics with SQL, 2nd Edition by Upom Malik, Matt Goldwasser, Benjamin Johnston
8. Data Analytics: Data Science for Beginners, SQL Computer Programming for Beginners, Statistics for Beginners by Matt Foster

CHAPTER 11:
PERFORMANT SQL

Learning Outcomes

- Students will learn Concept of Database Scanning.
- Students will learn concept of Query Planning.
- Students will learn the Index scanning

Structure

11.1 DATABASE SCANNING METHOD

SQL database offers many methods for scanning, searching and selecting data. Appropriate scanning method is to be selected, amount of data in database, Number of required fields, records to be returned, no of times query is executed.

The method used to determine when to handle a record is database scanning.

Types of scanning:

- **“Periodic”**: A record can be processed in regular time interval.
- **“Event”**: Event scanning involves detecting the occurrence of an event triggered by another software component, which calls the routine `postevent`.
- **“I/O Event”**: This is a request for processing record to respond a hardware interrupt.

- **“Passive”**: Passive database scanning refers to the process of monitoring and analyzing database activity without actively interfering with or modifying the operations of the database. This technique allows for the collection of data and detection of potential security threats or anomalies while maintaining the normal functioning of the database system.
- **“Scan once”**: Arranges for a record to be processed one time.

Database scanning is explained in this chapter. It begins by outlining the database fields that are used for scanning. The scanning system's interface is then covered. A brief description of scanners' implementation is provided in final section.

11.2 Query Planning

The query plan is a process to get information about operations required to execute a query.The query plan gives information about,:

- Number of steps in operation.
- Which Tables and columns are involved in operation.
- Amount of data is processed
- Operation Cost.
- Which operations in a query are using the most resources.

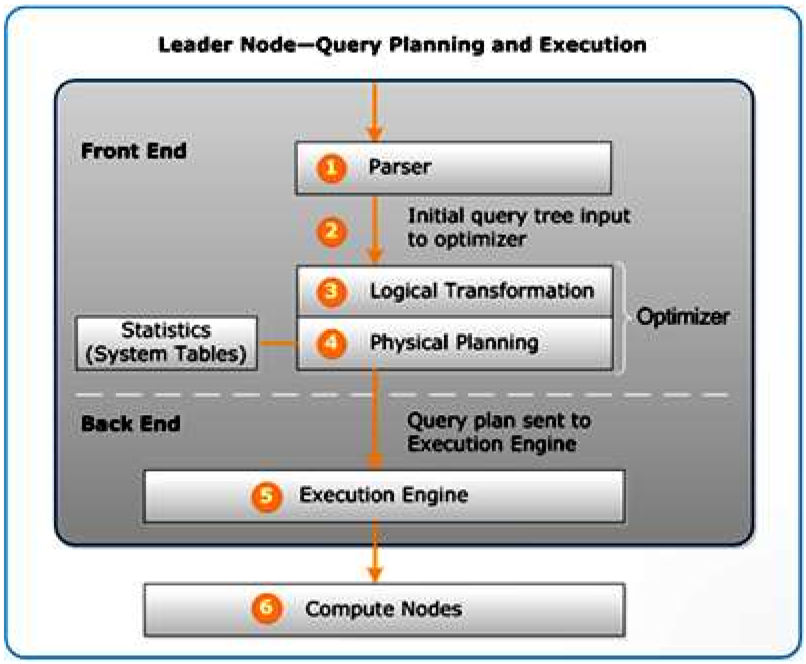

Leader Node—Query Planning and Execution
Front End
1 Parser
2 Initial query tree input to optimizer
3 Logical Transformation
4 Physical Planning
Optimizer
Statistics (System Tables)
Back End
Query plan sent to Execution Engine
5 Execution Engine
6 Compute Nodes

Figure: 11.1 Query Planning

A query execution plan, often referred to simply as a query plan, outlines the sequence of operations required to retrieve data from a SQL RDMS. It represents practical application of relational model's concept of access plans.

Structured Query Language is a declarative language with multiple execution paths for a given query, each varying in performance. When a query is submitted, the query optimizer creates several potential execution plans and chooses the most efficient one. However, since query optimizers are not always perfect, database users and administrators frequently need to manually review and tweak these plans to improve performance.

The optimizer generates multiple plans for the SQL statement, taking into account the available access paths. It then estimates the cost of each plan, which represents the anticipated resource usage required to execute the query. This cost estimation includes factors like I/O, CPU, and memory usage for the access paths and join orders. Generally, plans with higher costs are expected to take more time to execute than those with lower costs. However, for parallel plans, the relationship between resource usage and elapsed time is more complex. Ultimately, the optimizer chooses the plan with the lowest estimated cost.

11.3 Index Scanning:

Index Scans are utilized by query optimizer to generate most efficient execution plan for the Structured Query Language engine. During an index scan, rows are retrieved by scanning the index based on the values specified in the query. This process fetches data from an index that exists on one or more columns. To perform an index scan, the index is searched for the column values specified in the query. If the query only involves columns that are within the index, the data is retrieved directly from the index, bypassing the table itself.

The index contains:

- “Indexed value”
- “Rowid of row”

If other columns other than indexed columns are accessed, then rows in table are located by using rowed.

An index scan is of following types:

- “Assessing I/O for Blocks”
- “Index Unique Scans”

- “Index Range Scans”
- “Index Range Scans Descending”
- “Index Skip Scans”
- “Full Scans”
- “Fast Full Index Scans”
- “Index Joins”
- “Bitmap Indexes”
- Assessing I/O for Blocks

Input/Output operations are performed in blocks. The optimizer evaluates full table scans based on the percentage of blocks accessed rather than the number of rows. This evaluation is known as the index clustering factor. When each block contains a single row, the number of rows accessed corresponds directly to number of blocks accessed. However, most tables store multiple rows per block. As a result, target rows might be concentrated in a small number of blocks or dispersed across many blocks. The index clustering factor is a characteristic of index that reflects distribution of similar indexed column values across the data blocks in a table. A lower clustering factor indicates that the rows are tightly grouped within fewer blocks, whereas a higher clustering factor suggests that the rows are more widely distributed across multiple blocks. Consequently, a high clustering factor implies that more blocks must be accessed to retrieve the desired rows using a range scan, increasing the cost of fetching data by rowid.

- “Index Unique Scans”:

This type of scan retrieves at most a single rowid. Oracle employs a unique scan when a query involves a "UNIQUE" or "PRIMARY KEY" constraint, ensuring that only one row is accessed.

-“Optimizer Uses Index Unique Scans”:

This access path is used when all columns of a unique B-tree index, or an index created by a primary key constraint, are specified with equality conditions.

- Index Unique Scan Hints:

Generally, using a hint for a unique scan is unnecessary. However, there are exceptions, such as when accessing the table through a database link from a local table or when the table is small enough that the optimizer chooses a full table scan. The hint “INDEX(alias index_name)” guides the optimizer to use a specific index, but it does not specify the access path (e.g., range scan or unique scan).

- Index Range Scans:

An “index range scan” is frequently used to retrieve selective data. This operation can be either bounded (“with limits on both sides”) or unbounded (“with one or both sides open”). Rows with identical values are sorted in ascending order by their rowid, and the data is returned in ascending order based on the index columns. To sort data explicitly, use the ORDER BY clause rather than depending on the index. However, if an index can fulfill the requirements of an ORDER BY clause, the optimizer will use the index to avoid additional sorting.

- Optimizer Uses Index Range Scans:

When optimizer discovers one or more leading columns in an index that meets certain requirements, like the ones listed below, it performs a range scan.

- “col1 = :b1”
- “col1 < :b1”
- “col1 > :b1”

When using an AND combination of preceding conditions for the leading columns in an index, wildcard searches should not be at the beginning of the search pattern. For instance, using “col1 LIKE 'ASD%”` is appropriate, whereas “col1 LIKE '%ASD” is not, as the latter does not facilitate a range scan. Range scans can utilize both unique and non-unique indexes and can assist in avoiding sorting when the indexed columns are involved in the "ORDER BY" or "GROUP BY" clause.

- Index Range Scan Hints:

Sometimes, the optimizer might opt for a different index or perform a full table scan, in which case a hint can be necessary. The hint “INDEX(table_alias index_name”) directs the optimizer to use a specified index. For more details on “INDEX” hint, refer to “Hints for Access Paths.”

- Index Range Scans Descending:

An index range scan descending is essentially the same as an index range scan, with the distinction that data is retrieved in descending order. Typically, database indexes are stored in ascending order by default. This type of scan is commonly employed when data needs to be ordered in descending order, such as retrieving most recent entries first, or when searching for values less than a specified value.

- Optimizer Uses Index Range Scans Descending:

When an index can effectively fulfill the descending order supplied in a "order by" clause, the optimizer uses an index range scan descending.

- Index Range Scan Descending Hints:

For this access path, hint “INDEX_DESC(table_alias index_name)” is employed.

- Index Skip Scans:

Index skip scans use non-prefix columns to improve index scans. Index blocks can usually be accessed more quickly than table data blocks. It is possible to conceptually split a composite index into more manageable sub-indices by skip scanning. When a query for skip scanning omits the first column of the composite index, it is considered "skipped". The number of unique values in the first column is equal to the number of these logical sub-indices. When there are many distinct values in the non-leading columns of the composite index but few distinct values in the leading column, skip scanning becomes useful.

- Full Scans:

A full index scan eliminates the necessity for a sorting operation because the data is inherently ordered by the index key, and it reads blocks sequentially. Such a scan is utilized in several scenarios:

1. **“ORDER BY” Clause**: When using an “ORDER BY” clause and all columns specified in the “ORDER BY” are included in the index, their order must correspond to the order of the leading index columns. “The ORDER BY” can reference all or a subset of the index columns.

2. **“Sort Merge Join”:** Instead of performing a full table scan followed by a sorting operation, a full index scan can be used if all columns referenced in the query are part of the index. The order of these columns in the query must match the order of the leading index columns. The query can reference all or a subset of the index columns.

3. “**GROUP BY Clause**”: If a “GROUP BY” clause is present and the columns in the “GROUP BY” are part of the index, their order in the index does not necessarily need to match

their order in the “GROUP BY” clause. The “GROUP BY” clause can reference all or a subset of the index columns.

These conditions allow for efficient query execution by leveraging the ordering provided by the index, thereby optimizing performance without the need for additional sorting operations.

- Fast Full Index Scans:

Fast full index scans serve as an efficient alternative to full table scans in SQL relational database management systems. This method is utilized when an index contains all necessary query columns and includes at least one NOT NULL constrained column in its index key. Unlike a full table scan, which accesses entire table, a fast full index scan retrieves data directly from the index itself.

However, it's important to note that a fast full index scan does not facilitate sorting of data by the index key, as the retrieved data is not ordered in this manner. The process involves reading the entire index through multiblock reads, which allows for parallelization similar to table scans.

It's crucial to understand that fast full index scans are incompatible with bitmap indexes. To implement this method, you can specify it using initialization parameter “OPTIMIZER_FEATURES_ENABLE” or by employing the “INDEX_FFS” hint. This approach leverages multiblock I/O and supports parallel processing, thereby enhancing query performance compared to traditional full index scans.

- Index Joins:

An index join utilizes a hash join operation with indexes that include all columns referenced in the query. It enables retrieving all required column values directly from these indexes, thus eliminating the necessity for table access. However, it's essential to understand that an index join does not eliminate the need for a sort operation.

- Bitmap Indexes:

In a bitmap join, key values are represented using bitmaps, where a mapping function converts each “bit position into a rowed”. This approach enables efficient merging of indexes associated with multiple conditions in a “WHERE” clause, leveraging Boolean operations to handle both “AND and OR” conditions effectively.

11.4 Summary

SQL databases provide a range of methods for scanning, searching, and selecting data, with the optimal method chosen based on factors such as database size, required fields, records to return, and query frequency. Database scanning dictates how records are processed and is detailed in a query plan, also known as a query execution plan, within SQL relational database management systems. This concept is a specific application of access plans within the relational model. Index scans are leveraged by the query optimizer to create efficient execution plans for the SQL engine, fetching data by traversing the index according to the indexed column values specified in the query statement.

11.5 Self-Assessment Questions

1. Discuss Database scanning concept.
2. Discuss types of scanning.
3. Explain query planning
4. Explain Index scanning

11.6 References

9. https://www. https://www.tutorialspoint.com/
10. https://docs.oracle.com
11. Geeks for Geeks, W3School
12. Data analysis using SQL by Gordon Linoff
13. The Applied SQL Data Analytics by Benjamin Johnston, Matt Goldwasser, and Upom Malik
14. Learning SQL by Alan Beaulieu
15. The Applied SQL Data Analytics - Quick, Interactive Approach to Learning Analytics with SQL, 2nd Edition by Upom Malik, Matt Goldwasser, Benjamin Johnston
16. Data Analytics: Data Science for Beginners, SQL Computer Programming for Beginners, Statistics for Beginners by Matt Foster