Database

- A **database** is a structured collection of data that is stored and accessed electronically.
- Databases are essential for organizing, storing, managing, and retrieving information efficiently, especially when dealing with large amounts of data.
- Databases are used in almost every application. [A-Z Software Products]

Key Concepts in Databases:

Data: Raw facts and figures that can be processed to form meaningful information.

Database Management System (DBMS): A software tool that manages and interacts with databases. It provides tools to create, read, update, and delete data (often referred to as CRUD operations). Some popular DBMS include:

Relational DBMS (RDBMS): e.g., MySQL, PostgreSQL, Oracle, SQL Server.
NoSQL DBMS: e.g., MongoDB, Cassandra, Redis.

Tables: In relational databases, data is stored in tables (like spreadsheets). Each table consists of rows (records) and columns (fields or attributes). A row is a single data record, and a column is a property or characteristic of that record.

Schema: The structure or design of a database, defining how tables relate to one another, what columns they have, and the constraints (rules) on the data. In relational databases, this defines the relationships between different tables.

SQL (Structured Query Language):
SQL (Structured Query Language) is a standardized query language used to manage and manipulate relational databases.
It is designed for querying, updating, inserting, and deleting data in a database.
SQL allows users to interact with the database to perform various tasks, such as:
1. **Querying Data**: Using SELECT statements to retrieve data from one or more tables.
2. **Inserting Data:** Using INSERT INTO statements to add new records to a table.
3. **Updating Data**: Using UPDATE statements to modify existing records.
4. **Deleting Data:** Using DELETE statements to remove records from a table.
5. **Database Structure:** Defining and modifying tables, indexes, and relationships using CREATE, ALTER, and DROP statements.
6. **Data Security**: Controlling access to data using commands like GRANT, REVOKE and DENY.

* ** SQL is used in many popular database systems, including MySQL, PostgreSQL, Microsoft SQL Server, and SQLite. SQL allows us (authenticated users only) to execute complex operations (transactions) on database objects.

Relationships: In relational databases, tables can be linked using relationships. These relationships can be:

One-to-One: One record in Table A relates to one record in Table B.
One-to-Many: One record in Table A relates to many records in Table B.
Many-to-Many: Many records in Table A relate to many records in Table B (typically implemented using a junction table).

Indexes: These are used to speed up the retrieval of data. An index is a data structure that allows for faster search operations on a database table.

Normalization: The process of organizing data in a database to reduce redundancy and dependency. This is done through dividing larger tables into smaller, related tables.

ACID Properties: These are a set of properties that ensure reliable processing of database transactions:

Atomicity: All operations in a transaction are completed or none.
Consistency: A transaction will bring the database from one valid state to another.
Isolation: Transactions are executed independently and in isolation from one another.
Durability: Once a transaction is committed, it remains stored in the database, even in the case of a system failure.

Types of Databases:

Relational Databases (RDBMS): Store data in tables with predefined relationships between them. They use SQL for querying.
Examples: MySQL, PostgreSQL, Oracle, SQL Server.

NoSQL Databases: These databases are more flexible in terms of structure. They are better suited for unstructured data, large volumes of data, or applications requiring high scalability.
Examples: MongoDB, Cassandra, CouchDB, Redis.

Object-Oriented Databases: Store data as objects, similar to how data is represented in object-oriented programming.
Examples: db4o, ObjectDB.

Graph Databases: Used for storing data with complex relationships like social networks, recommendation systems, etc. They use graph structures for querying.

Examples: Neo4j, ArangoDB.

*** POC ***

In-memory Databases: Store data primarily in the computer's memory (RAM) rather than on disk, for faster access.
Examples: Redis, Memcached, H2

==========

SQL USERS:

==========

How to Create USER:

Creating new User

- We can create new user in Database using following command :
- Syntax:

  CREATE USER <username> IDENTIFIED BY <password>; username:
  The name of the user you want to create.
  password: The password for the user.

Granting Privileges

- After creating the user, you typically need to assign privileges. The following example grants the user john basic login privileges:
- Syntax:

  GRANT CONNECT, RESOURCE TO <username>;
  CONNECT: Allows the user to log in to the database.
  RESOURCE: Allows the user to create and manage schema objects like tables, views etc…

LOCKING and UNLOCKING USERS

- To lock a user account, preventing the user from logging in, use the following syntax:
- LOCKING: ALTER USER username ACCOUNT LOCK;
- UNLOKING: ALTER USER username ACCOUNT UNLOCK;

Disable/Enable Login for a User Account

- In addition to locking and unlocking the account, we can also disable or enable login access by modifying specific user account settings like password expiration or invalid attempts.
- Here's how to manage some of these settings:
  Expire a User's Password: ALTER USER username PASSWORD EXPIRE;

RDBMS :

RDBMS stands for relational Database Management System.
It is a type of database management system (DBMS) that stores data in a structured format using rows and columns, and the data is organized into tables. The relational model allows for the use of Structured Query Language (SQL) to manage and manipulate the data.

- MySQL
- PostgreSQL
- Oracle Database
- Microsoft SQL
- SQLite Server

SQL (Structured Query Language)

SQL (Structured Query Language) is a standard query language used for managing and manipulating relational databases. It allows you to interact with databases, perform operations like retrieving, updating, deleting, and inserting **DATA**, and define the structure of the database itself.

Data - Raw Fact (3 types Structured, Unstructured, semi-structured)

To interact with oracle database, oracle people released their own query language called Oracle SQL.

E.F. CODD Rules :

If any database management system follows E.F. CODD rules then such Database management systems are considered as Relation Database Management System.

Information Rule:
All data in a relational database is stored in the form of tables (also called relations). The data should be represented as values in rows and columns of these tables. This ensures that data is structured and can be queried systematically.

Guaranteed Access Rule:
Every data element (value) in the database must be accessible by a combination of the table name, primary key (unique identifier), and column name. This eliminates ambiguity in data retrieval.

Systematic Treatment of Null Values:
Null values (representing missing or undefined data) must be treated consistently. A relational database should handle nulls with special rules for comparison and operations, so it doesn't lead to unpredictable results.

Dynamic Online Catalog:
The database should store its metadata (structure and definitions of tables, columns, and constraints) in a system catalog. This catalog must be accessible using the same relational language that the database supports for user queries.

Comprehensive Data Sublanguage Rule:
A relational database must support a language that can express all operations on the data, such as querying, updating, inserting, and deleting, all using a single language (like SQL).

View Updating Rule:
Views (virtual tables created from queries on base tables) must be updatable. That is, changes made to a view should be reflected in the base tables, as long as the view doesn't violate the rules of relational integrity.

Set-Level Insertion, Update, and Deletion:
The database system should allow operations like insert, update, and delete to work on entire sets (collections) of data at once, rather than on individual rows. This enhances efficiency and supports the relational model's set theory foundation.

Physical Data Independence:
The application programs (queries, reports, etc.) should not be affected by changes in how the data is stored physically (e.g., on different storage devices or with different indexing methods). This rule ensures the data's logical view is separate from its physical implementation.

Logical Data Independence:
Changes to the logical schema (the structure of tables, columns, and relationships) should not affect application programs. For instance, renaming a column or adding a new table should not break existing applications or require changes in them.

Integrity Independence:
Integrity constraints (rules that ensure data accuracy and validity) should be defined separately from the application. These constraints should be stored in the system catalog and be enforceable automatically by the database system.

Distribution Independence:
A relational database should allow data to be distributed across different locations or machines without affecting how the data is accessed. Applications should remain unaware of where data physically resides.

Non-subversion Rule:
If a database allows low-level access (e.g., bypassing the relational model), it must not allow users to bypass the relational integrity. This ensures that all data manipulations are done according to the rules defined by the relational model, protecting the integrity of the database.

[1, 2, 5, 12]

========== O ==========

NOTE :

Already we know that data will be stored in tables (Columns, Rows format). Here the thing is Before inserting data into the Database Table, we should follow below rules :

Data should be validated with data type and constraint.

Data Type : It indicates what type of data that we are going to store in the table.
Constraint : Constraint is a validation rule whether entered data following my requirement or Not.

DATA TYPES

In Oracle SQL, there are several data types available to store different types of data in database columns. These data types are categorized into several groups based on the type of data they store. Here is an overview of the various data types available in Oracle SQL:

1. Numeric Data Types : These are used to store numbers, both integer and floating-point numbers.

NUMBER:

Used to store numbers, either integers or decimals.

Syntax: NUMBER(p, s)
where p is the precision (total number of digits)
s is the scale (number of digits to the right of the decimal point).
Example: NUMBER(10,2) can store numbers up to 10 digits in total, with 2 digits after the decimal point.

INTEGER:

A synonym for NUMBER(38) (stores integers up to 38 digits).

FLOAT:

A synonym for NUMBER with floating-point precision. It is typically used to store approximate numeric values.

DECIMAL:

A synonym for NUMBER, used for fixed-point numbers.

BINARY_FLOAT:

Stores single-precision floating-point numbers (4 bytes).

BINARY_DOUBLE:

Stores double-precision floating-point numbers (8 bytes).

2. Character Data Types

These data types store string or textual data.

CHAR:

Used to store fixed-length character strings.
Syntax: CHAR(n) where n is the length of the string (maximum of 2000 bytes).
Example: CHAR(10) will store a string of exactly 10 characters, padding with spaces if necessary.

VARCHAR2:

Used to store variable-length character strings.
Syntax: VARCHAR2(n) where n is the maximum length (up to 4000 bytes).
Example: VARCHAR2(50) can store strings up to 50 characters long.

NCHAR:

Used to store fixed-length Unicode characters (useful for storing multilingual data).
Syntax: NCHAR(n) where n is the number of characters.
Example: NCHAR(10) will store a fixed-length string of 10 characters.

NVARCHAR2:

Used to store variable-length Unicode character strings.
Syntax: NVARCHAR2(n) where n is the maximum number of characters.
Example: NVARCHAR2(50) can store strings up to 50 characters long, supporting Unicode.

3**. Date and Time Data Types**

These data types are used to store date and time values.

DATE:

Used to store date and time values (including the year, month, day, hour, minute, and second).
Syntax: DATE (stores values from January 1, 4712 BC to December 31, 9999 AD).

TIMESTAMP:

Stores date and time with fractional seconds (to nanosecond precision).
Syntax: TIMESTAMP(p) where p is the precision (fractional second digits).
Example: TIMESTAMP(3) stores fractional seconds up to 3 digits.

TIMESTAMP WITH TIME ZONE:

Stores date, time, and time zone information.
Syntax: TIMESTAMP WITH TIME ZONE.
Example: TIMESTAMP '2025-02-14 10:30:00 -05:00'.

TIMESTAMP WITH LOCAL TIME ZONE:

Stores timestamp data normalized to the session's time zone, but displays the value in the local time zone.

INTERVAL:

Used to store time intervals (difference between two date or time values).
Types: INTERVAL YEAR TO MONTH and INTERVAL DAY TO SECOND.

4. Binary Data Types

These data types store binary data such as images, files, etc.

BLOB (Binary Large Object):

Used to store large binary data such as images, multimedia, etc. Maximum size: 4 GB.

CLOB (Character Large Object):

Stores large text data (up to 4 GB).

Useful for storing large amounts of text (e.g., documents).

BFILE:

Stores binary data that is located outside the database, but referenced by Oracle.

Used for files that reside on the file system, but are accessed by the database.

5. LOB (Large Object) Data Types

These are used for storing large objects, such as text, images, and multimedia.

BLOB: Binary Large Object.

CLOB: Character Large Object.

NCLOB: National Character Large Object (stores large Unicode text).

BFILE: Binary File (used to store external binary files).

6. Rowid and Urowid

These are used to store unique identifiers for rows in a database table.

ROWID:

A unique identifier for each row in a table (internally assigned by Oracle).

It can be used for fast access to rows.

UROWID:

A universal unique identifier for rows, used with tables having ROWID as the primary key.

7. XML Data Types

For storing XML data.

XMLTYPE:

Used to store XML data.

Can store both well-formed and valid XML documents.

8. Boolean Data Type

Oracle does not have a direct Boolean data type, but you can simulate Boolean values using other data types like CHAR or NUMBER (e.g., 1 for TRUE and 0 for FALSE).

==========

Constraints :

==========

In Oracle SQL, constraints are rules or restrictions placed on columns or tables to ensure data integrity, accuracy, and consistency. These rules help maintain the quality of the data stored in a database. Oracle provides several types of constraints, each serving a specific purpose. Here's an overview of the most commonly used types of constraints in Oracle SQL:

1. NOT NULL Constraint

Purpose: Ensures that a column cannot have a NULL value.
Use Case: If you need a column to always have a value (e.g., a person's name, email address).
Syntax

Column-Name	<Data-Type>	[Constraint]
Phone	NUMBER(10)	NOT NULL

UNIQUE Constraint

- Ensures that all values in a column (or combination of columns) are unique across rows in the table.
- Unlike the primary key, a table can have multiple unique constraints.
- Can accept NULL values unless otherwise defined.
- Syntax :

Column-Name	<Data-Type>	[Constraint]
Phone	NUMBER(10)	UNIQUE NOT NULL

PRIMARY KEY Constraint

- Ensures that a column (or combination of columns) uniquely identifies each row in a table.
- A table can have only one primary key.
- The primary key column(s) cannot contain NULL values.
  Syntax :

Column-Name	<Data-Type>	[Constraint]
Phone	NUMBER(10)	PRIMARY KEY

FOREIGN KEY Constraint

- Used to create a relationship between two tables by enforcing a link between a column in one table (the foreign key) and a column in another table (the referenced primary key).

- Because above point, we can also call it as **REFERENTIAL INTEGRITY CONSTRAINT**
- Ensures referential integrity, meaning data in the foreign key column must match data in the referenced column or be NULL.

CHECK Constraint

- Defines a condition that must be satisfied for each row in a table.
- Ensures that data in a column follows a specific rule or condition.

DEFAULT Constraint

- Provides a default value for a column when no value is specified during an insert operation.
- This is useful for ensuring that certain columns always have a default value if no explicit value is provided.

INDEX (Implicit Constraint):

- Although not a direct "constraint" on data, indexes are often created implicitly to enforce unique and primary key constraints.
- Indexes help improve performance when querying a table but are not strictly required by the SQL standards for data integrity.

Composite Key (Combination of Constraints)

- Sometimes, multiple columns are combined to form a primary key, foreign key, or unique constraint.

STRANGE SITUATIONS :

Enable/Disable Constraints

- Constraints can be enabled or disabled for various maintenance purposes, for example, during bulk data loads.
- Example : ALTER TABLE employees DISABLE CONSTRAINT employees_pk;

==========

Benefits of Constraints:

==========

Data Integrity: Ensures that only valid data is inserted into the database, maintaining the accuracy and consistency of data.

Relationships Between Tables: Helps maintain logical relationships between different tables (e.g., parent-child relationships).

Data Validation: Automatically validates data during insert, update, or delete operations.
Preventing Errors: Prevents insertion of NULL values or duplicates when they are not allowed

{
NOTE : We can specify constraints information at the time of table creation and after table creation.
}

SUB-LANGUAGES OF SQL :

Data Definition Language (DDL) and AUTO-COMMIT

DDL (Data Definition Language) refers to the subset of SQL commands used for defining and managing database structures like tables, indexes, schemas, and views. It deals with the structure of the database rather than the data itself.

Primary DDL Commands:

CREATE: Used to create new database objects like tables, views, schemas, or indexes.
ALTER: Allows modifications to existing database objects (e.g., adding or removing columns from a table).
RENAME: Used to rename a database object, such as a table or column.
TRUNCATE: Removes all rows from a table but does not remove the table structure itself.
DROP: Deletes a database object, such as a table, schema, or index, from the database entirely.
CREATE : By using 'create' command we can create objects in database such as database, table, index, view, trigger, function, procedure, sequence …

CREATE TABLE ---------- Create a new table.
CREATE DATABASE ---------- Create a new database.
CREATE INDEX ---------- Create an index on one or more columns.
CREATE VIEW ---------- Create a view.
CREATE PROCEDURE ---------- Create a stored procedure.
CREATE TRIGGER ---------- Create a trigger.
CREATE SCHEMA ---------- Create a schema to organize objects.
CREATE USER ---------- Create a new user (in some DBMS).
CREATE SEQUENCE ---------- Create a sequence to generate unique values.

Table Creation Using Create Command :

Creating a table (Normal Table) :

```
CREATE TABLE <TABLE_NAME> (
COLUMN_1 <DATATYPE> [CONSTRAINT] [DEFAULT <VALUE>],
COLUMN_2 <DATATYPE> [CONSTRAINT] [DEFAULT <VALUE>],
COLUMN_3 <DATATYPE> [CONSTRAINT] [DEFAULT <VALUE>],
);
```

NOTE :
[***] INDICATES THAT – FIELD IS OPTIONAL
< *** > INDICATES THAT – FIELD IS MANDATORY

Creating Table using foreign key:

```
CREATE TABLE departments (
id INT PRIMARY KEY, -- Primary key for the parent table
dept VARCHAR(100) NOT NULL
);

CREATE TABLE employees (
id INT PRIMARY KEY, -- Primary key for the child table
name VARCHAR(100) NOT NULL,
dept INT, -- FKreferring to departments
FOREIGN KEY (dept) REFERENCES departments(id)
);
```

Creating table from existing table (with custom columns):

```
CREATE TABLE <new_table> AS
SELECT * FROM <existing_table>;
CREATING TEMP TABLE ( SESSION BASED ):
CREATE TEMPORARY TABLE <table_name> (
 column1 datatype,
 column2 datatype,
 ...
);
```

Creating table using select into statement:

```
SELECT column1, column2, ...
INTO new_table
FROM existing_table
[ WHERE condition ];
```

Creating table using IF NOT EXISTS:

```
CREATE TABLE IF NOT EXISTS table_name (
 column1 datatype,
 column2 datatype,
 ...
);
```

ALTER:

- The ALTER command in SQL is used to modify the structure of an existing database
- object, such as a table, column, index, or constraint.
- This command allows you to make changes without having to recreate the entire object.
- You can use ALTER to add, drop, or modify columns, rename tables, or change other aspects of database objects.

Add a Column to a Table:

```
ALTER TABLE table_name
ADD column_name datatype [DEFAULT default_value];
```

Note:
Impact: Adding a new column to a table with existing data typically does not affect the existing data in the table. However, the new column will contain NULL values (unless you specify a DEFAULT value).

Modifying a Column:

```
ALTER TABLE table_name
MODIFY column_name new_datatype;
```

Note: Impact: When modifying a column's data type or constraints, it's important to consider the existing data because the operation could fail if the existing data does not match the new column constraints or data type.

Renaming a Column:

```
ALTER TABLE table_name
RENAME COLUMN old_column_name TO new_column_name;
```

Drop a column

```
ALTER TABLE table_name
DROP COLUMN column_name [CASCADE CONSTRAINTS];
```

Renaming table:

```
ALTER TABLE old_table_name
RENAME TO new_table_name;
```

Adding pk:

```
ALTER TABLE table_name
ADD CONSTRAINT constraint_name PRIMARY KEY (column_name);
```

Adding FK:

```
ALTER TABLE table_name
ADD CONSTRAINT constraint_name FOREIGN KEY (column_name)
REFERENCES parent_table (parent_column);
```

Droping PK:

```
ALTER TABLE table_name
DROP CONSTRAINT constraint_name;
```

RENAME:

Renaming the existing table:

- RENAME old_table_name TO new_table_name;

Renaming the column of a table:

- ALTER TABLE table_name RENAME COLUMN old_column_name TO new_column_name;

Renaming the Index:

- ALTER INDEX old_index_name RENAME TO new_index_name;

Renaming the Constraint:

- ALTER TABLE table_name RENAME CONSTRAINT old_constraint_name TO

TRUNCATE :

The TRUNCATE command removes all rows from a table, effectively resetting the table to an empty state. Unlike DELETE, TRUNCATE is not transactional and cannot be rolled back once executed, unless the session is under a transaction.

Truncate existing table:

```
TRUNCATE TABLE table_name;
```

DROP :

DROP command is a Data Definition Language (DDL) operation used to completely remove a database object, such as a table, index, view, or other schema objects, from the database. Once an object is dropped, it cannot be recovered unless you have a backup.

Characteristics of DROP & PURGE:

Permanent Deletion: The object is permanently deleted from the database, including all its data and structure.

No Rollback: DROP is a DDL operation, and like other DDL commands, it cannot be rolled back once executed (unless inside a transaction, but even then, it is only temporarily undone within that session).

Affects Dependencies: Dropping an object may affect other objects that depend on it, such as foreign key constraints, triggers, views, or other references.

Syntax:

DROP <Object> <Name> PURGE;

Object may be a [view, database, table, sequence, procedure, function, trigger] etc…

==========

DATA MANIPULATION LANGUAGE

==========

DML stands for Data Manipulation Language.

- It is a subset of SQL (Structured Query Language) used for managing data within relational databases.
- DML is primarily concerned with the manipulation of data, rather than the structure of the database itself.
- The common DML commands are:
    1) INSERT
    2) INSERT ALL
    3) UPDATE
    4) DELETE

INSERT:

WAY 01 :

```
INSERT INTO
<TABLE_NAME>   (column1, column2, column3, ...)
VALUES         (value1, value2, value3, ...);
```

[columns count == values count]

WAY 02 :

```
INSERT ALL
    INTO table1 (column1, column2, ...)  VALUES (value1, value2, ...)
    INTO table2 (column1, column2, ...)  VALUES (value1, value2, ...)
  ...
SELECT * FROM dual;
```

NOTE: dual: A special dummy table in Oracle used when no actual data is needed in the SELECT statement (usually for operations like this).

WAY 03 : *** imp ****

Table is having 5 columns but I want to insert data into first 2 columns only

```
INSERT INTO
<TABLE_NAME>   (column1, column2)
VALUES              (value1, value2);
[ columns count == values count ]
```

Note: Mentioning the columns details after table name is optional]

```
INSERT INTO
<TABLE_NAME>   VALUES   (value1, value2);
```

Note: Dynamic data binding

```
INSERT INTO employees (first_name, last_name, salary)
VALUES ('&first_name', '&last_name', &salary);
```

UPDATE :

UPDATE statement is used to modify existing data in a table. You can update one or more columns for specific rows based on a condition.

```
UPDATE   table_name
SET   column1 = value1,   column2 = value2,   ...
WHERE condition;
```

[If we do not use where condition on that time all records will be updated with provided value]

DELETE :

In Oracle, DML (Data Manipulation Language) is a subset of SQL used to manage data within tables. The DELETE operation is a part of DML and is used to remove one or more rows from a table.

```
DELETE FROM table_name
[WHERE condition];
```

DELETE vs. TRUNCATE:

- The DELETE operation can delete specific rows and allows the use of conditions, and it is slower because it logs each row deletion.
- The TRUNCATE operation removes all rows in a table, but it is faster and does not
- log individual row deletions. However, TRUNCATE cannot be rolled back in the
- same way that DELETE can.

TCL (Transaction Control Language)

Transaction Control Language :

- TCL is a subset of SQL used to manage the changes made by DML (Data Manipulation Language) statements like INSERT, UPDATE, and DELETE.
- TCL allows you to control the transaction boundaries, ensuring that a series of operations can be treated as a single unit of work.
- The primary purpose of TCL is to ensure the ACID properties (Atomicity, Consistency, Isolation, Durability) of transactions.

COMMIT:

- The COMMIT command is used to permanently save all changes made during the current transaction to the database. After a COMMIT, the changes become visible to other users and the transaction is complete.
- Syntax: COMMIT;

ROLLBACK:

- The ROLLBACK command is used to undo any changes made during the current transaction. It reverts all DML operations (like INSERT, UPDATE, DELETE) to their previous state. □ Syntax: ROLLBACK

SAVEPOINT:

- A SAVEPOINT is used to set a point within a transaction to which you can later roll back. It allows partial rollback within a larger transaction.
- Syntax: SAVEPOINT savepoint_name;

SET TRANSACTION:

- The SET TRANSACTION command is used to configure the properties of the current transaction, such as setting the isolation level or setting the transaction mode.
- Syntax: SET TRANSACTION ISOLATION LEVEL ;

LEVELS:

READ COMMITTED: This is the default isolation level in Oracle. A transaction can only see committed data. If another transaction modifies data, the current transaction sees the updated value only after it is committed.
SERIALIZABLE: This isolation level provides the highest level of isolation. It ensures that the current transaction sees a consistent snapshot of the data and no other transactions can modify data that it reads until it is completed.
READ ONLY: Ensures that the transaction can only read data but cannot modify it. Useful for long-running reports or querying.
READ WRITE: Allows both reading and writing of data. This is typically the default unless explicitly set.

Savepoint:

A savepoint is a named point within a transaction that allows you to roll back only a portion of the work done in the transaction, instead of undoing the entire transaction. This can be useful if you want to revert to a known state within a transaction without losing all the progress made up to that point.

Sytax: ROLLBACK TO SAVEPOINT savepoint_name;

DCL (Data Control Language) :

- DCL in Oracle SQL refers to a subset of SQL commands used to manage access to data and control permissions within a database.
- DCL primarily consists of the following two commands:
  1) GRANT
  2) REVOKE

GRANT:

The GRANT statement is used to give users or roles permission to perform certain operations (such as SELECT, INSERT, UPDATE, DELETE) on database objects (tables, views, procedures, etc.). It can be applied to individual users or roles, allowing for specific privileges to be granted.
Syntax:

GRANT privilege_name ON object_name TO user_or_role;

```
GRANT SELECT, INSERT ON employees TO john;
```

GRANT WITH OPTION:

John can grant those select and insert to other users too;

```
GRANT SELECT, INSERT ON employees TO john WITH GRANT OPTION;
```

REVOKE

The REVOKE statement is used to remove previously granted permissions from a user or role. It allows administrators to revoke specific privileges from users.

```
REVOKE privilege_name ON object_name FROM user_or_role;
```

Example:

```
REVOKE SELECT, INSERT ON employees FROM john;
```

In case of assigning permissions to ROLE;

```
GRANT SELECT, INSERT, UPDATE ON employees TO <ROLE_NAME>;
```

OBJECT PRIVILEGES

Object privileges allow users to perform operations on specific database objects (e.g., tables, views, sequences, procedures). Here are the key object privileges:

SELECT: Allows the user to query data from a table, view, or synonym.

- INSERT: Allows the user to insert data into a table or view.
- UPDATE: Allows the user to modify data in a table or view.
- DELETE: Allows the user to delete data from a table or view.
- ALTER: Allows the user to modify the structure of a table or view (e.g., adding or removing columns).
- DROP: Allows the user to delete a table, view, or other object.
- INDEX: Allows the user to create or drop indexes on a table.
- REFERENCES: Allows the user to create foreign key constraints that reference a specific table or column.
- EXECUTE: Allows the user to execute a stored procedure or function.

Role-Based Privileges

Roles are a collection of privileges that can be granted to users or other roles. Common predefined roles in Oracle include:

- DBA: Provides a user with all system privileges in the database (full administrative rights).
- RESOURCE: Provides privileges to create and manage database objects (e.g., tables, views).
- CONNECT: Provides basic privileges to connect to the database and create objects (like tables and views).
- SELECT_CATALOG_ROLE: Provides read access to Oracle data dictionary views.
- EXECUTE_CATALOG_ROLE: Provides execute privileges on procedures and functions in the data dictionary.

System Privileges

System privileges allow users to perform operations at the system level (such as creating and managing database objects, modifying users, etc.). Some common system privileges include:

- CREATE SESSION: Allows the user to connect to the database.
- CREATE TABLE: Allows the user to create a new table in the database.
- CREATE VIEW: Allows the user to create a view.
- CREATE PROCEDURE: Allows the user to create a stored procedure or function.
- ALTER SESSION: Allows the user to change session-level settings.
- CREATE USER: Allows the user to create new database users.
- DROP USER: Allows the user to drop a database user.
- GRANT ANY PRIVILEGE: Allows the user to grant any privilege to any other user.
- CREATE ROLE: Allows the user to create roles.

System Privileges

- DROP ROLE: Allows the user to drop roles.
- SELECT ANY TABLE: Allows the user to query any table in the database, regardless of ownership.
- DELETE ANY TABLE: Allows the user to delete from any table.
- INSERT ANY TABLE: Allows the user to insert data into any table.
- UPDATE ANY TABLE: Allows the user to update data in any table.

DQL (Data Query Language)

- DQL (Data Query Language) in Oracle SQL is used for querying data from a database.
- It includes the SELECT statement, which is the primary command used to retrieve data from one or more tables.
    - Here's an overview of key concepts and operations in DQL :

Selection :

- Selection : Selection refers to the operation of filtering rows based on a certain condition.
- This operation is analogous to the WHERE clause in SQL, which selects rows that meet a specified condition.
- Example

```
SELECT *
FROM employees
WHERE department = 'Sales';
```

Projection:

- Projection refers to the operation of selecting certain columns from a table, ignoring the others.
- This operation is analogous to the SELECT clause in SQL, which specifies which columns to retrieve.
- Example

```
SELECT first_name, last_name
FROM employees;
```

JOINS:

JOIN is a keyword used to combine the results of two or more tables.
We have various JOINS to perform specific task.

CLAUSE :

- Clause refers to a part or component of a SQL query that specifies a particular condition or operation.
- It's a building block that helps define the query's purpose and behavior.
- We have several clauses in Oracle SQL they are :

SELECT: Specifies the columns to retrieve.
FROM: Specifies the table(s) from which to retrieve data.
WHERE: Filters rows based on a condition.
GROUP BY: Groups rows based on specified columns.
HAVING: Filters groups after grouping with GROUP BY.
ORDER BY: Sorts the result set in ascending or descending order.
JOIN: Combines rows from two or more tables based on related columns.
UNION: Combines results of two SELECT queries, removing duplicates.
DISTINCT: Removes duplicate records from the result set.
LIMIT / FETCH: Limits the number of rows returned.
INTO: Assigns the result of a query to variables (used in PL/SQL).
CONNECT BY: Performs hierarchical queries.
WITH: Defines a temporary result set (Common Table Expression).
PIVOT: Converts rows into columns.
UNPIVOT: Converts columns into rows.
INSERT: Adds new rows to a table.
UPDATE: Modifies existing rows in a table.
DELETE: Removes rows from a table.
MERGE: Performs insert, update, or delete based on a condition.
CASE: Provides conditional logic within queries.
EXISTS: Tests if a subquery returns any rows.
ALL / ANY: Compares a value to all or any values returned by a subquery

---

ORDER OF EXECUTION :

In Oracle SQL, the order of execution for a SQL query follows a specific sequence. Understanding this order helps in writing more efficient queries and avoids errors when applying filters, joins, or aggregations. Here's the typical order of execution for a SELECT statement:

FROM: The tables and joins are processed first. All the data is selected from the specified tables.

JOIN: If there are any joins, they are performed at this point. The system combines rows from two or more tables based on the join conditions.

WHERE:
After the tables are joined, the WHERE clause is applied to filter the rows based on the specified conditions.

GROUP BY:
This clause groups the filtered rows into subsets based on one or more columns.

HAVING:
After the rows are grouped, the HAVING clause filters the groups (unlike WHERE, which filters rows before grouping).

SELECT:
The SELECT clause is used to specify the columns you want in the final result. It is executed after filtering and grouping.

DISTINCT:
If you are using DISTINCT, duplicate rows are removed after the SELECT statement.

ORDER BY:
The ORDER BY clause sorts the results based on one or more columns. This is applied at the final stage.

LIMIT / FETCH (optional):
Finally, any row limiting operation like LIMIT, FETCH FIRST, or ROWNUM is applied to restrict the number of rows returned

```
SELECT column1, COUNT(*)
FROM employees
JOIN departments ON employees.department_id = departments.department_id
WHERE department_name = 'Sales'
GROUP BY column1
HAVING COUNT(*) > 10
ORDER BY column1;
```

Execution :

FROM: employees and departments tables are joined.
JOIN: The condition for joining the tables is applied.
WHERE: Filters rows where department_name = 'Sales'.
GROUP BY: Groups the data based on column1.
HAVING: Filters groups with a count greater than 10.
SELECT: Specifies the columns (column1 and COUNT(*)).
ORDER BY: Sorts the result by column1.

GROUP BY :

The GROUP BY clause groups rows that have the same values in specified columns into summary rows. It is often used in conjunction with aggregate functions like COUNT(), SUM(), AVG(), MAX(), and MIN() to summarize the data.

```
SELECT department_id, AVG(salary)
FROM employees
[ +WHERE condition ]
GROUP BY department_id;
```

This groups the employees by their department and calculates the average salary per department.

Key Points:

Group Data: The GROUP BY clause groups rows that have the same values in specified columns into summary rows.
Aggregate Functions: When you group the data, you can perform aggregate calculations such as COUNT(), SUM(), AVG(), MAX(), MIN() on the columns.
Columns in SELECT: Any column that isn't an aggregate function must be included in the GROUP BY clause.
Order of Grouping: The rows are grouped according to the columns specified in the GROUP BY clause, and the result set will show one row per group.

GROUP BY WITH HAVING :

==========

The HAVING clause can be used to filter groups after applying the GROUP BY. It's similar to WHERE but works on aggregated data.

```
SELECT salesperson_id, SUM(amount) AS total_sales
FROM sales
[ WHERE condition ]
GROUP BY salesperson_id
HAVING SUM(amount) > 1000;
```


github.com/vamsi97049

FUNCTIONS

Functions are named block of statements which are used to perform a specific task.
SQL Function contains 3 important parts (1) Method Name
(2) Method Parameters
(3) Return type
[Note : SQL supports only built-in functions] (User Defined Functions present in PL/SQL)

SQL supports two types of built in Functions
(1) Single Row Functions (2) Multi Row Functions

Single Row Functions (SRF) :

Single-row functions take **one input row** (from a dataset) and produce **one output value** for each row processed. These functions operate on each row individually and return a single result per row, not multiple outputs for each input.

1. Character Functions

- **CONCAT(str1, str2)**
  - **Parameters**: Two string values (str1, str2).
  - **Functionality**: Combines (concatenates) two strings into one.
- **LENGTH(str)**
  - **Parameters**: A string value (str).
  - **Functionality**: Returns the number of characters in the string.
- **LOWER(str)**
  - **Parameters**: A string value (str).
  - **Functionality**: Converts all characters in the string to lowercase.
- **UPPER(str)**
  - **Parameters**: A string value (str).
  - **Functionality**: Converts all characters in the string to uppercase.
- **INITCAP(str)**
  - **Parameters**: A string value (str).
  - **Functionality**: Converts the first letter of each word to uppercase and the rest to lowercase.
- **TRIM(trim_character FROM str)**
  - **Parameters**: A string value (str) and an optional trim character (trim_character).
  - **Functionality**: Removes the specified trim_character from both the beginning and end of the string.
- **SUBSTR(str, start_position, length)**
  - **Parameters**: A string value (str), start position (start_position), and length (length).
  - **Functionality**: Extracts a substring from the string, starting at a specific position and with a specified length.

2. Numeric Functions

- **ABS(number)**
  - **Parameters**: A numeric value (number).
  - **Functionality**: Returns the absolute (non-negative) value of the given number.
- **CEIL(number)**
  - **Parameters**: A numeric value (number).
  - **Functionality**: Returns the smallest integer greater than or equal to the number.
- **FLOOR(number)**
  - **Parameters**: A numeric value (number).
  - **Functionality**: Returns the largest integer less than or equal to the number.
- **ROUND(number, decimal_places)**
  - **Parameters**: A numeric value (number) and the number of decimal places (decimal_places).
  - **Functionality**: Rounds the number to the specified number of decimal places.
- **TRUNC(number, decimal_places)**
  - **Parameters**: A numeric value (number) and the number of decimal places (decimal_places).
  - **Functionality**: Truncates the number to the specified number of decimal places.
- **MOD(number1, number2)**
  - **Parameters**: Two numeric values (number1, number2).
  - **Functionality**: Returns the remainder when number1 is divided by number2.

3. Date Functions

- **SYSDATE**
  - **Parameters**: None.
  - **Functionality**: Returns the current date and time of the system.
- **CURRENT_DATE**
  - **Parameters**: None.
  - **Functionality**: Returns the current date and time of the session's time zone.
- **ADD_MONTHS(date, months)**
  - **Parameters**: A date value (date) and the number of months to add (months).
  - **Functionality**: Adds a specified number of months to the given date.
- **MONTHS_BETWEEN(date1, date2)**
  - **Parameters**: Two date values (date1, date2).
  - **Functionality**: Returns the number of months between two dates.
- **TRUNC(date, format)**
  - **Parameters**: A date value (date) and a date format (format).
  - **Functionality**: Truncates a date to the specified format (e.g., day, month, year).

4. Conversion Functions

- **TO_NUMBER(expr)**
  - **Parameters**: An expression (expr).
  - **Functionality**: Converts an expression to a numeric value.
- **TO_CHAR(expr)**
  - **Parameters**: An expression (expr).

- **Functionality**: Converts an expression to a string value.

- **TO_DATE(expr, format)**
  - **Parameters**: An expression (expr) and a date format (format).
  - **Functionality**: Converts a string expression to a date value based on the specified format.
- **CAST(expr AS datatype)**
  - **Parameters**: An expression (expr) and a target data type (datatype).
  - **Functionality**: Converts the expression to the specified data type.

5. Null-Related Functions

- **NVL(expr1, expr2)**
  - **Parameters**: Two expressions (expr1, expr2).
  - **Functionality**: Returns expr2 if expr1 is NULL, otherwise returns expr1.
- **COALESCE(expr1, expr2, ...)**
  - **Parameters**: Multiple expressions (expr1, expr2, ...).
  - **Functionality**: Returns the first non-null expression from the list of parameters.

6. Miscellaneous Functions

- **DECODE(expr, search1, result1, search2, result2, ..., default)**
  - **Parameters**: An expression (expr), followed by pairs of search values and results (search1, result1, search2, result2,...), and an optional default value (default).
  - **Functionality**: Compares the expression to each search value, returning the corresponding result for the first match. If no match is found, the default value is returned.

Multi Row Functions (SRF) :

Multi-row functions (also called as **aggregate functions**) operate on multiple rows of data and return a single result for the entire group of rows, rather than operating on each individual row. These functions are typically used in GROUP BY queries, where you aggregate or summarize data from several rows.

- COUNT(…)
- MAX(…)
- MIN(…)
- AVG(…)
- SUM(…)
- GROUP_CANCAT(…)
- STDDEV(…)

GROUP_CONCAT(…) : USED TO GROUD ALL COLUMN VALUES IN A SINGEL STATEMENT (STRING TYPE)

STDDEV(…) : USED TO GET STANDARD DEVIATION OF NUMERIC COLUMN

NORMALIZATION :

- Normalization in Oracle SQL refers to the process of organizing data in a database to reduce redundancy and dependency.
- It helps ensure that the database is efficient, minimizes data duplication, and makes it easier to maintain.
- The process involves breaking down a database into smaller, more manageable tables, each representing a distinct entity or concept.
- The normalization process generally follows several normal forms (NF), each one addressing a specific type of redundancy and dependency.

1 Normal Form :

- Goal: Eliminate repeating groups and ensure that all values in a column are atomic (indivisible).
- Steps:
  - Each column should contain only one value (atomic).
  - There should be no repeating groups or arrays.
  - All entries in a column must be of the same type.
- Example :

Before :

StudentID	Name	Courses
101	Alice	Math, Science
102	Bob	History, English, Math

After :

StudentID	Name	Course
101	Alice	Math
101	Alice	Science
102	Bob	History
102	Bob	English
102	Bob	Math

Second Normal Form (2NF)

- ❖ Goal: Eliminate partial dependency (when a non-key column depends on part of a composite primary key).
- ❖ Steps:

StudentID	Course	Instructor
101	Math	Dr. Smith
101	Science	Dr. Johnson
102	Math	Dr. Smith

  - Achieve 1NF first.
  - Remove partial dependencies by ensuring that all non-key attributes are fully dependent on the entire primary key.

StudentID	Course
101	Math
101	Science
102	Math

Before 2NF :

Course	Instructor
Math	Dr. Smith
Science	Dr. Johnson

After 2NF

Third Normal Form (3NF)

- ❖ Goal: Eliminate transitive dependency (when a non-key attribute depends on another non-key attribute).
- ❖ Steps:
  - Achieve 2NF first.
  - Remove transitive dependencies by ensuring that non-key attributes depend only on the primary key.

StudentID	Name	Department	DeptHead
101	Alice	Science	Dr. Brown
102	Bob	Math	Dr. Green

Before 3NF :

StudentID	Name	Department
101	Alice	Science
102	Bob	Math

Department	DeptHead
Science	Dr. Brown
Math	Dr. Green

After 3NF

Boyce-Codd Normal Form (BCNF)

- ❖ Goal: A stricter version of 3NF, ensuring that every determinant is a candidate key.
- ❖ Steps:
  - Achieve 3NF first.
  - Ensure that every functional dependency is a dependency on a candidate key (there should be no exceptions).

Fourth Normal Form (4NF)

- ❖ Goal: Eliminate multi-valued dependencies (when one attribute depends on two or more independent attributes).
- ❖ Steps:
  - Achieve 3NF or BCNF first.
  - Remove any multi-valued dependencies.

Fifth Normal Form (5NF):

- ❖ A table is in 5NF if:
- ❖ It is in 4NF.

- ❖ It has no join dependency and it cannot be decomposed into smaller tables without losing information.
- ❖ 5NF deals with situations where a table has complex relationships and it is difficult to decompose it while maintaining data integrity.

*** NOTE ***

[Normalization concept comes into the picture when we are designing a Database for a software application. DBA team will create multiple tables to store the data instead of storing everything in single database table.]

==========

JOINS IN SQL :

==========

In SQL, a **JOIN** is used to combine records from two or more tables based on a related column between them.

Types of JOINS in SQL ;

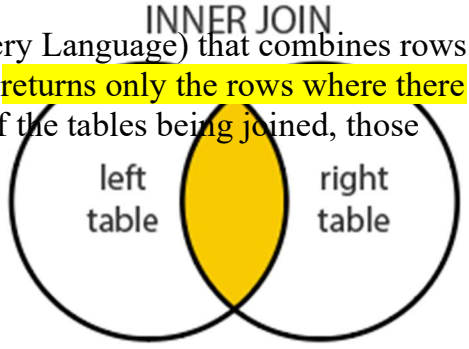
1) INNER JOIN
2) LEFT JOIN (or LEFT OUTER JOIN)
3) RIGHT JOIN (or RIGHT OUTER JOIN)
4) FULL JOIN (or FULL OUTER JOIN)
5) CROSS JOIN
6) SELF JOIN

It is highly recommended to use ANSI Join syntax over Oracle's old-style join syntax for the following reasons:

- Readability: ANSI syntax is more explicit, clear, and easier to understand.
- Standardization: It is a SQL standard used by almost all databases.

INNER JOIN :

An **INNER JOIN** is a type of join operation in SQL (Structured Query Language) that combines rows from two or more tables based on a related column between them. It returns only the rows where there is a match in both tables. If there is no match between the columns of the tables being joined, those rows will not appear in the result.



Syntax : ANSI SYNTAX

```
SELECT column1, column2, ...
FROM table1
INNER JOIN table2
ON table1.column = table2.column;
```

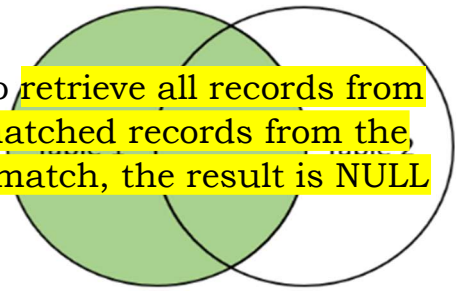
E-Commerce Systems: When analyzing sales data, you might use inner joins to link orders and customers. For example, you could find details about the customers who have made purchases by joining the orders table with the customers table.

Financial Systems: In banking or financial systems, inner joins are used to connect transactions with accounts or customers. For example, showing all transactions that occurred in a specific account, only for those accounts with active balances.

ORACLE SQL SYNTAX :

```
SELECT column1, column2, ...
FROM table1, table2
WHERE table1.column = table2.column;
```

LEFT JOIN :



LEFT JOIN (also known as LEFT OUTER JOIN) is used to retrieve all records from the left table (the table before the LEFT JOIN), and the matched records from the right table (the table after the LEFT JOIN). If there is no match, the result is NULL on the side of the right table.

(ANSI)

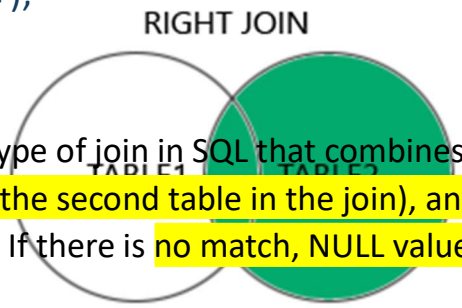
```
SELECT columns
FROM table1
LEFT JOIN table2
ON table1.column = table2.column;
```

Example : Training institute management team decided to give 10 % discount on every course for students (registered students) even if they have not enrolled any courses.

In case of an e-commerce store, Analyst want to filter all records like all orders placed by the customers and order details, even they did not complete the payment yet.

(Oracle syntax)

```
SELECT column1, column2, ...
FROM table1 t1, table2 t2
WHERE t1.column_name = t2.column_name(+);
```



RIGHT JOIN :

A **RIGHT OUTER JOIN** (also known simply as a **RIGHT JOIN**) is a type of join in SQL that combines records from two tables. It returns all rows from the right table (the second table in the join), and the matching rows from the left table (the first table in the join). If there is no match, NULL values are returned for columns from the left table.

Syntax :

```
SELECT columns
FROM table1
RIGHT OUTER JOIN table2
ON table1.column = table2.column;
```

Example : In case of an E-commerce store, staff wants to track all orders delivery status along with unfulfilled orders too.

FULL JOIN :

A **FULL JOIN** is used when we want to:

- Fetch **matching records** from both the left and right tables.
- Also include **unmatched records** from both the left table **and** the right table.

- In cases where there's **no corresponding row** in one of the tables, **NULL** values are used to represent the missing data.

Example : In large MNC companies, in some situations there is chance where employee do not belong to any department and In some scenarios, employees are not present in the department. In these type of scenarios if Management team wants to know about employees who does not belong to any department, department,

which does not have working employees.

In case of financial systems, if I want to track the both successful transactions along with unsuccessful transaction between two persons or two financial institutes.

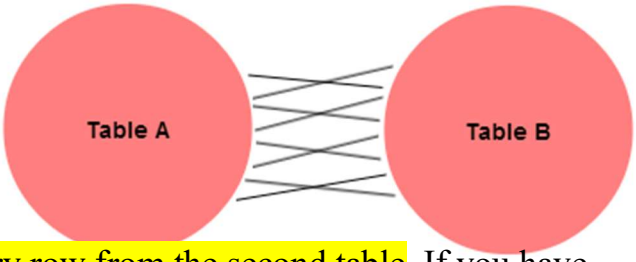
Syntax :

```
SELECT column_list
FROM table1
FULL OUTER JOIN table2
ON table1.column = table2.column;
```

Old oracle Syntax

```
SELECT column_list
FROM table1, table2
WHERE table1.column = table2.column(+)
OR table1.column(+) = table2.column;
```

CROSS JOIN

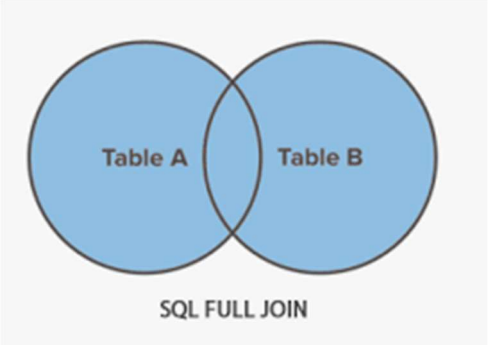


it simply combines each row from the first table with every row from the second table. If you have two tables, Table A with 3 rows and Table B with 4 rows, the result of a **CROSS JOIN** will have 3 * 4 = 12 rows.

Syntax :

```
SELECT column1, column2, ...
FROM table1
CROSS JOIN table2;
```

SELF JOIN :



A **self-join** is a type of join in SQL where a table is joined with itself. This is useful when we need to compare rows within the same table. We typically use a self-join when we have hierarchical data or need to find relationships between records that share the same structure.

```
SELECT a.column1, a.column2, b.column1, b.column2
FROM table_name a
JOIN table_name b ON a.some_column = b.some_column;
```

Example : Presenting the relationships (Parent-child or tree based)

- When you have hierarchical data, like an organization structure or a tree structure, and need to represent relationships like managers and employees, or categories and subcategories.
- When we want to know all duplicate orders ordered buy the same customer.

Syntax : (ANSI)

```
SELECT
a.column1, a.column2, b.column1, b.column2
FROM table_name a
JOIN table_name b
ON a.some_column = b.some_column;
```

(Old oracle Syntax)

```
SELECT a.column1, a.column2, b.column1,
b.column2
FROM table_name a, table_name b
WHERE a.some_column = b.some_column;
```

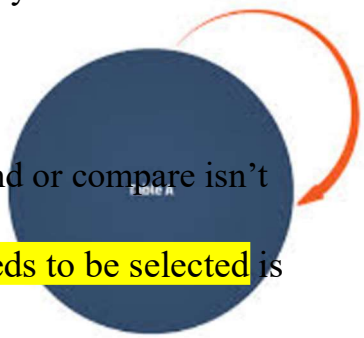
SUB QUERIES :

A **subquery** in Oracle SQL is a query embedded inside another query. It allows you to perform more complex operations by using the result of one query within another query.

Subqueries are incredibly useful :

Scenario 01 : When we have unknowns [Data that we want to compute, find or compare isn't directly available in the current table]

Scenario 02 : When we have conditions needs to be executed or data needs to be selected is



present in the different tables irrespective of their relationship (if present or not present)

Syntax :

```
SELECT column1
FROM table1
WHERE column2 = (SELECT column2 FROM table2 WHERE condition);
```

Types of SUB-QUERIES :

Sub-Queries are classified into two types (1) Single Row Subquery
(2) Multi Row Subquery

** Correlated Subquery (It might be SRSQ* or MRSQ*)

Single Row Subquery :

- If a subquery return only one record after its execution, then such types of subqueries are considered as Single Row Subquery.
- When we are dealing with sub-query concept either we should use relational operators [>, <, <=, >=, <>, =] or Sub-query operators [IN, ANY, ALL] along with subquery.

Multi Row Subquery :

- If a subquery return only one record after its execution, then such types of subqueries are considered as Single Row Subquery.
- When we are dealing with sub-query concept either we should use relational operators [>, <, <=, >=, <>, =] or Sub-query operators [IN, ANY, ALL] along with subquery.

Correlated Sub-Query :

- A **correlated subquery** is a type of subquery where the inner query (subquery) is dependent on the outer query.

- This means that for each row processed by the outer query, the inner query will be executed once using values from that row of the outer query.
- The inner query refers to a column from the outer query, which is why it's called "correlated."

Execution Process :

1) **Outer query** executes first and generates rows.
2) **Inner query** uses a value from the current row of the outer query.
3) The inner query is executed for each row returned by the outer query.
4) The results from the inner query are used to filter, calculate, or make decisions in the outer query.

Syntax :

```
SELECT column1, column2, ...
FROM outer_table outer_alias
WHERE column_name operator (
   SELECT column_name
   FROM inner_table inner_alias
   WHERE outer_alias.column_name = inner_alias.column_name
);
```

[We can nest subqueries up to 256 **]**

Example

```
SELECT p.ProductName
FROM Products p
WHERE p.ProductID IN (
   SELECT s.ProductID
   FROM Sales s
   WHERE s.ProductID IN (
      SELECT s1.ProductID
      FROM Sales s1
      WHERE s1.ProductID IN (
         SELECT s2.ProductID
         FROM Sales s2
         WHERE s2.ProductID IN (
            -- Repeat this for 256 levels
            SELECT s255.ProductID
            FROM Sales s255
            WHERE s255.ProductID = s.ProductID
         )
      )
   )
GROUP BY s.ProductID
HAVING SUM(s.SaleAmount) > (SELECT AVG(saleAmount) FROM Sales WHERE ProductID = s.ProductID) );
```

- **Correlated Subquery**: The subquery references the outer query's ProductID field, making it "correlated."
- **Nested Subqueries**: The query has several layers of nested subqueries (up to 256), which can lead to inefficiency or a limit being hit in certain databases.
- **Performance**: While this is a conceptual example, having 256 nested subqueries like this would result in performance problems in practice. It's better to use Common Table Expressions (CTEs) or temporary tables in most cases to simplify and improve performance.

CTE (Common Table Expression)

- A **Common Table Expression (CTE)** is a temporary result set that can be referenced within a SELECT, INSERT, UPDATE, or DELETE statement.
- CTEs are defined using the WITH keyword at the beginning of the query.
- A CTE improves the readability and modularity of SQL queries by allowing subqueries to be written once and referenced multiple times in the same query.

Syntax :

```
WITH cte_name AS (
   -- Your query that generates a result set
   SELECT column1, column2
   FROM table_name
   WHERE condition
)
SELECT *
FROM cte_name;
```

Improved Readability:
CTEs allow you to write complex queries in a modular way. Instead of nesting subqueries, which can make a query hard to read and understand, a CTE breaks the query into logical blocks.

Reusability:
A CTE can be referred to multiple times within the main query. This avoids the need to repeat the same subquery in various places, making the query more efficient and easier to maintain.

Self-Referencing CTEs:

Recursive CTEs are supported, allowing you to query hierarchical data (e.g., organizational charts, bill of materials). This recursive capability is something that's difficult to achieve with traditional subqueries.

Simplifies Complex Queries:
Complex queries that require multiple levels of subqueries can become difficult to manage. CTEs allow you to break them down into separate, easily understandable steps.

Temporary Result Set:
Since the result of a CTE is temporary and only valid for the duration of the query, it reduces the need for creating temporary tables or intermediate tables in your database, making it more efficient.

Eliminates Repeated Subqueries:
A common issue with subqueries is the repetition of the same query in various parts of the SQL. Using a CTE, the subquery can be defined once and referenced multiple times, thus enhancing performance.

CTE instead of SUBQUERY : (CTE and RECURSIVE CTE)

When dealing with deep levels of nesting, these subqueries can become difficult to maintain and can lead to performance issues. CTEs help by:

1. **Improving Clarity and Modularity:**
   - Instead of writing deeply nested subqueries, we can break each subquery into a CTE, which improves the readability of the query and makes it easier to manage.
2. **Reducing Repeated Execution:**
   - In a correlated subquery, the subquery may be executed multiple times for each row in the outer query. With a CTE, you can calculate the result once and reference it multiple times, reducing redundant execution and improving performance.
3. **Minimizing Complex Nested Queries:**
   - Deeply nested queries often lead to problems in debugging and performance issues. A CTE simplifies these queries, making them more maintainable and less prone to errors.
4. **Better Performance:**
   - In cases of high nesting, CTEs can sometimes be optimized better by the query planner, leading to better performance compared to highly nested subqueries.

RECURSIVE CTE

A **recursive Common Table Expression (CTE)** in Oracle SQL is a type of CTE that allows you to perform recursive queries, meaning it can repeatedly call itself to return a result set. Recursive CTEs are useful when dealing with hierarchical or tree-structured data, such as organizational charts, file systems, or bill-of-materials, where a parent-child relationship exists between rows.

How a Recursive CTE Works

A recursive CTE consists of two parts:

1. **Base Case (Anchor Member):** This is the initial query that selects the starting point for the recursion. It typically selects the root or top-level data in the hierarchy (e.g., the top-most manager in an organization).
2. **Recursive Case (Recursive Member):** This part references the CTE itself and uses the results of the base case to generate the next level of results. The recursive query will keep calling itself until a specified termination condition is met.

The recursive process in a CTE works like this:

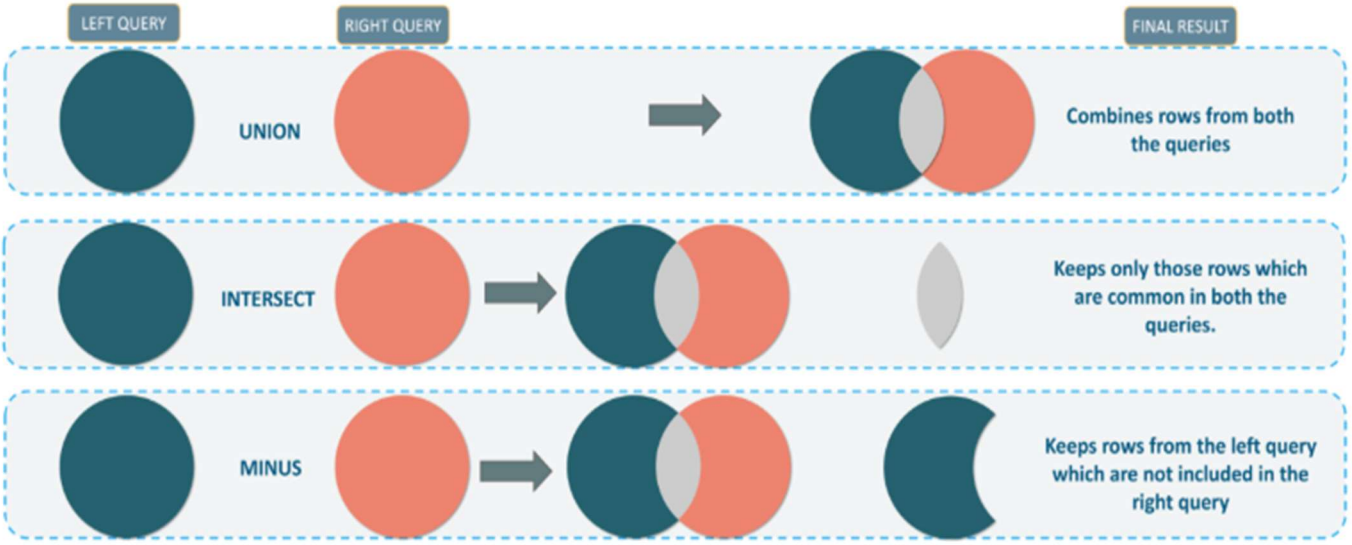
- The base case is executed first.
- The recursive part is executed repeatedly, referencing the previous iteration's result to fetch the next level of data.
- The recursion stops when no more data is returned by the recursive query.

Syntax for Recursive CTE :

```
WITH RECURSIVE cte_name (column1, column2, ...) AS (
   -- Base case: select the initial set of rows
   SELECT column1, column2, ...
   FROM table_name
   WHERE condition_for_base_case

   UNION ALL

   -- Recursive case: select rows that reference the previous result set
   SELECT t.column1, t.column2, ...
   FROM table_name t
   JOIN cte_name c ON t.column = c.column
   WHERE condition_for_recursion
)
SELECT * FROM cte_name;
```



SET OPERATORS :

this part will be removed

Set operators are used to combine the results of two or more SELECT queries. These operators allow you to perform operations like merging, intersecting, or finding the difference between result sets. Here are the most common set operators in Oracle SQL:

UNION

- The **UNION** operator combines the results of two or more SELECT queries into a single result set. It eliminates duplicate rows.

Syntax : SELECT column_name FROM table1

```
UNION
SELECT column_name FROM table2;
```

- The number and order of columns must be the same in each SELECT query.
- By default, **UNION** removes duplicates. If you want to include duplicates, use **UNION ALL** (explained below).

UNION ALL

- The **UNION ALL** operator also combines the result sets of two or more SELECT queries, but unlike **UNION**, it does not remove duplicate rows.
- **Syntax**:

```
SELECT column_name FROM table1
UNION ALL
SELECT column_name FROM table2;
```

- **Important Points**: It is generally faster than **UNION** because it does not need to perform a check for duplicate rows.

INTERSECT

- The **INTERSECT** operator returns only the rows that are common to both SELECT queries.
- **Syntax**:

```
SELECT column_name FROM table1
INTERSECT
SELECT column_name FROM table2;
```

- **Important Points**:
  It returns only the rows that appear in both result sets.
  Similar to **AND** in logic, it shows the intersection of two datasets.

MINUS

- The **MINUS** operator returns the rows from the first SELECT query that are **not** in the second SELECT query.
- **Syntax**:

```
SELECT column_name FROM table1
MINUS
SELECT column_name FROM table2;
```

- **Important Points**:
  - It returns the difference between two datasets.
  - Rows from the first dataset that are not present in the second are included in the result set.

Partitioning :

- Partitioning is a database design technique used to split large tables or indexes into smaller, more manageable pieces, called partitions.
- Each partition is stored separately but behaves like a single table.
- We have 4 types of partitions :
  - Range Partitioning
  - List Partitioning
  - Hash Partitioning
  - Composite Partitioning

❖ Benefits :

1. Improves Query performance (By limiting the scanning)
2. Parallel Processing : Index partitions can be processed parallel.
3. Efficient Index Management : Partitions of an index can be rebuilt easily.
4. Complex Query handling
5. Fast row retrievals for frequent SELECT queries.
6. Index Pruning.

[Without Indexing concept Partitioning will provide limited benefits]
[Query maintenance, Parallel processing]

➢ **Range partitioning** :

- Data is divided based on a specific range of values.

- For example, records could be split into different partitions based on numeric ranges like age ranges (e.g., 0-20, 21-40, etc.) or date ranges.
- Examples : Sales - Monthly Sales data
  Age - Custom age group
  Bank - Custom transactions
  Logs - Deleting Old Log Files
- Drawbacks :
  - This partitioning is not recommended when the data is unevenly distributed.
  - This technique is suitable for predictable partitioning only.
  - It is not good for smaller data sets.

- Syntax :

```
CREATE TABLE table_name (
  column1 datatype,
  column2 datatype,
  ...
  )       PARTITION BY RANGE (partition_column) (
          PARTITION partition_name_1 VALUES LESS THAN (value1),
          PARTITION partition_name_2 VALUES LESS THAN (value2),
          PARTITION partition_name_3 VALUES LESS THAN (value3),
          ...
  );
```

Adding New Partitioning :

```
ALTER TABLE table_name
ADD PARTITION (PARTITION partition_name VALUES LESS THAN (value));
```

Splitting an existing partition :

```
ALTER TABLE table_name

SPLIT PARTITION partition_name AT (value) INTO (

    PARTITION partition_name_1 VALUES LESS THAN (value1),
```

```
    PARTITION partition_name_2 VALUES LESS THAN (value2)
);
```

Dropping Existing Partition :

```
ALTER TABLE table_name
DROP PARTITION partition_name [, partition_name_2.....] ;
[ or ]
ALTER TABLE sales
DROP PARTITION  <partition_name>
UPDATE GLOBAL INDEXES;
```

Note : when we drop a partition in a partitioned table, the data stored in that partition is typically permanently deleted unless specific actions are taken to handle the data. This behavior is dependent on the database management system (DBMS) you are using, and the overflow strategy refers to how you want the data to be handled before it is removed from the partition.

**** Common Overflow Strategies When Dropping a Partition ****

Some DBMSs, like MySQL and Oracle, allow you to specify what should happen to the data in a partition when it is dropped. The two main strategies are:

1. Automatic Overflow (Move Data to Another Partition or Table)
   - In some DBMSs, you can specify an alternative partition or a storage area where the data should be moved to before the partition is dropped.
   - For MySQL: In MySQL, before dropping a partition, you might want to merge partitions or move data into other partitions to retain it.
2. Data Deletion (Remove Data Permanently)
   - By default, most DBMSs will simply delete the data within the partition when it is dropped.

- This approach is typically the most common. Once the partition is removed, its data is gone.

Merging to partitions :

ALTER TABLE sales

MERGE PARTITIONS p_2020, p_2021;

P_2021 data merged with p_2020 then relative update operations will be done internally.

List Partitioning :

- List partitioning divides data based on a discrete set of values for a specific column.
- This is typically used when you want to partition data by categories, such as regions or product types.
- Used when we have limited and set of categorical values
- For example, If you have a table of employees, you might use list partitioning based on their department IDs (e.g., HR, IT, Sales, Marketing).
- Discreet and non-sequential data
- Discrete Categories: The countries (US, UK, DE, IN) are distinct, non-sequential values, so list partitioning is a perfect choice.
- Query Optimization: If your queries frequently filter by the country (e.g., **SELECT * FROM Customer-Orders WHERE Country = 'US'**), list partitioning can speed up the query by limiting the search to the partition of interest.
- Storage Management: This partitioning method makes it easy to manage and maintain data by grouping it according to a logical division.

When to not use List partitioning: When we have dynamic categorical data, Uneven Distribution data then list partitioning is not recommended.

- Syntax :

```
CREATE TABLE table_name (
 column1 datatype,
 column2 datatype,
 ...
 )
 PARTITION BY LIST (column_name) (
 PARTITION partition_name_1 VALUES ('value1', 'value2'),
 PARTITION partition_name_2 VALUES ('value3', 'value4'),
 ...
 );
```

- Hash Partitioning
  - Hash partitioning is a method used to distribute data across multiple partitions (or nodes) by applying a hash function to a key attribute (often a primary key or another identifying value).
  - This method ensures that data is spread evenly across partitions, and it's commonly used in distributed databases to optimize performance and scalability.

How Hash partitioning is working :

Hash Function:
A hash function is applied to a selected key (e.g., user ID, order ID). The hash function produces a value that is used to determine which partition the record will be placed in.

Partitioning Strategy:

The hash value is usually mapped to one of the available partitions using modulo arithmetic or a similar approach. For instance, if there are 4 partitions, the system may use the formula:

partition_id = hash(key) % number_of_partitions

Benefits :

Even Distribution: Because of the hash function, data tends to be evenly distributed, which helps to avoid overloading a single partition.

Scalability: When new partitions are added, the system can easily rebalance the data by reapplying the hash function.
Performance: Query performance improves when data is distributed evenly because it prevents hotspots where a single partition is overloaded with data.

Drawbacks :

Adding new partitions which are having same hash value may lead to uneven data distribution.

When new partitions are added then scaling is required, this scaling activity might require rehashing and redistribution of the data.

Syntax :

```
CREATE TABLE table_name (
    column1 datatype,
    column2 datatype,
    ...
)
PARTITION BY HASH (column_name)
PARTITIONS N;  -- N is the number of partitions
```

Composite Partitioning :

Composite partitioning is a combination of two or more partitioning strategies. It allows you to first partition the table using one method (e.g., range), and then sub-partition each partition using a different method (e.g., hash or list).

```
CREATE TABLE table_name (
  column1 datatype,
  column2 datatype,
  ...
)
PARTITION BY RANGE (column_name1)
SUBPARTITION BY HASH (column_name2)
(
  PARTITION partition_name_1 VALUES LESS THAN (value1),
  PARTITION partition_name_2 VALUES LESS THAN (value2),
  ...
);
```

Reference Partitioning :

This technique is used when we want to create a partitioned table using partitioning scheme of another table. This is useful when there is relation between partitioned tables.

```
CREATE TABLE table_name (
column1 datatype,
column2 datatype,
...
 CONSTRAINT fk_name FOREIGN KEY (column) REFERENCES
parent_table(column)
)
 PARTITION BY REFERENCE (fk_name);
```

Interval Partitioning :

Interval partitioning is a variant of range partitioning where new partitions are automatically created when data exceeds a predefined range. It is useful for time-series data where new partitions need to be added over time.

```
CREATE TABLE table_name (
  column1 datatype,
  column2 datatype,
  ...
)
PARTITION BY RANGE (column_name) INTERVAL (interval_value) (
  PARTITION initial_partition VALUES LESS THAN (value)
);
```

Note : **INTERVAL : (INTERVAL '1' MONTH)**

INDEX

- An **index** in Oracle SQL is a database object that improves the speed of data retrieval operations on a table.
- It works similarly to an index in a book—allowing the database engine to quickly locate rows based on the values of one or more columns, without scanning the entire table.
- When you create an index on one or more columns of a table, Oracle creates a separate data structure (typically a B-tree or bitmap index) that allows the database to quickly find values in those columns.
- The index stores the values in the indexed columns along with pointers to the corresponding rows in the table.
- When you execute a query with conditions (e.g., WHERE clauses), Oracle can use the index to directly access the relevant rows, rather than having to search through every row in the table.

When to Use Indexes:

- Frequent Searches: If your application frequently queries a table with specific filters (e.g., looking for customers by their customer_id), indexing the columns involved in those filters will improve performance.
- Sorting and Grouping: If your queries often sort or group by certain columns, those columns might benefit from being indexed.
- Join Operations: If your queries often join tables on certain columns, creating indexes on those columns can speed up the join operations.

When not to use Indexes :

- For small tables
- If the table is transactional table (DML operations are common on the table) Why because, Indexes are needs to be updated when underlying data changes.

Syntax to create a index on table column:

```
CREATE [UNIQUE] INDEX index_name
ON table_name (column1, column2, ..., columnN);
```

index_name: The name you want to assign to the index.
table_name: The name of the table where the index is created.
column1, column2, ..., columnN:
The list of columns to be
indexed. These can be one or more columns.

B-tree Index (Balanced Tree Index)

- **Default Index Type**: This is the default index type in Oracle and is used when we create an index without specifying a type.
- **Structure**: It uses a balanced tree structure (B-tree), where the data is stored in sorted order, allowing for efficient searching, inserting, updating, and deleting.
- **Use Case**: Best for most general-purpose queries, especially those that involve equality or range-based searches.
- **Example**:

  CREATE INDEX idx_employee_id ON employees(employee_id);

Bitmap Index

- **Structure**: Bitmap indexes use a bitmap for each distinct value in the indexed column, where each bit corresponds to a row in the table. This index is efficient for columns with a low cardinality (i.e., a small number of distinct values).
- **Use Case**: Ideal for columns with a small number of distinct values, such as gender, status flags, or Boolean fields. Bitmap indexes are also useful for complex queries involving multiple conditions.
- **Advantages**: Bitmap indexes are space-efficient and provide fast query performance for certain types of queries, especially those involving OR conditions.
- **Disadvantages**: They are not suitable for tables with frequent updates, inserts, or deletes, as they require maintenance for every change.

- **Example**:

  CREATE BITMAP INDEX idx_gender ON employees(gender);

Composite Index (Concatenated Index)

- **Structure**: A composite index, also known as a concatenated index, is an index that includes multiple columns. It speeds up queries that filter on more than one column.
- **Use Case**: Useful for queries with multiple conditions, such as WHERE column1 = value1 AND column2 = value2.
- **Note**: The order of columns in the index matters. The index will be most effective if queries filter on the leftmost columns.
- **Example**:

  CREATE INDEX idx_name_salary ON employees(last_name, salary);

Unique Index

- **Structure**: A unique index ensures that no two rows in the table have the same value for the indexed columns. It is automatically created when you define a PRIMARY KEY or UNIQUE constraint on a column.
- **Use Case**: Used for enforcing data integrity and ensuring uniqueness in the table.
- **Example**:

  CREATE UNIQUE INDEX idx_unique_email ON employees(email);

6. Function-Based Index

- **Structure**: This index is based on expressions or functions applied to columns rather than the raw column values.
- **Use Case**: Useful for queries that filter on computed values or use functions like LOWER(), UPPER(), TO_DATE(), etc.
- **Example**:

  CREATE INDEX idx_lower_email ON employees(LOWER(email));

7. Reverse Key Index

- **Structure**: In a reverse key index, the bytes of the indexed column values are reversed before being stored in the index.
- **Use Case**: Effective when you have sequential data (e.g., a column that stores increasing integers) to avoid hot spots in the index and improve index performance for insert-heavy operations.
- **Example**:

  CREATE INDEX idx_reverse_id ON employees(employee_id REVERSE);

8. Domain Index

- **Structure**: A domain index is a custom index that Oracle allows you to create for specific data types. These indexes are often used in situations involving large objects (LOBs), spatial data, or text search (e.g., for full-text indexing).
- **Use Case**: Used in specialized applications, such as geographic data or full-text search, where a standard index is insufficient.
- **Example**:

  CREATE INDEX idx_spatial ON spatial_table(location) INDEXTYPE IS MDSYS.SPATIAL_INDEX;

9. Cluster Index (Table Clusters)

- **Structure**: A table cluster stores multiple tables in the same physical space based on a common column. This index type is used when you frequently query related tables that share a common key.
- **Use Case**: Suitable for scenarios where you often need to join tables based on common columns, improving performance by storing related rows together.
- **Example**:

  CREATE CLUSTER emp_dept_cluster (department_id NUMBER);
  CREATE INDEX emp_dept_idx ON CLUSTER emp_dept_cluster;

10. Bitmap Join Index

- **Structure**: A bitmap join index is used for join operations involving large tables. It stores a bitmap for the results of the join.
- **Use Case**: Useful in data warehousing environments where you frequently perform complex queries that join multiple large tables with low cardinality columns.
- **Example**:

  CREATE BITMAP INDEX idx_join_sales ON sales(customer_id, product_id);

11. Partial Index

- **Structure**: A partial index is an index created on a subset of the data based on a condition or filter (like a WHERE clause).
- **Use Case**: Useful when you only need to index part of a table's data, improving performance and reducing storage costs.
- **Example**:

```
CREATE INDEX idx_active_employees ON employees(employee_id)
WHERE status = 'ACTIVE';
```

12. XML Index

- **Structure**: XML indexes are used to speed up the retrieval of XML data stored in Oracle databases. They index XML documents in XMLType columns.
- **Use Case**: Useful for applications that deal with XML data stored in Oracle.
- **Example**:

```
CREATE INDEX idx_xml_doc ON xml_table(xml_column) INDEXTYPE IS
CTXSYS.CONTEXT;
```

Ranking :

- **ranking** refers to the process of assigning a unique rank or position to each row in a result set, typically based on the values of one or more columns.
- Oracle provides several ranking functions to facilitate this process, each with its own specific behavior.
- These functions are often used with OVER() clauses to calculate the rank of rows within a specific partition or the entire result set.

RANK() Purpose:

- Assigns a unique rank to each row in a result set. Rows with the same value receive the same rank, but the next row(s) will have a rank that is incremented by the number of tied rows.
- This causes gaps in the rank sequence
- Example :

```
SELECT employee_id, salary,
       RANK() OVER (ORDER BY salary DESC) AS salary_rank
 FROM employees;
```

DENSE_RANK() Purpose :

Similar to RANK(), but without gaps between ranks when there are ties. Rows with the same value receive the same rank, and the next rank is the immediate next number (no skipping).

Example:

```
SELECT employee_id, salary,
       DENSE_RANK() OVER (ORDER BY salary DESC) AS dense_salary_rank
FROM employees;
```

ROW_NUMBER()

Purpose: Assigns a unique sequential integer to each row, regardless of whether there are ties. This function does not account for duplicates or ties in values.
Example:

```
SELECT employee_id, salary,
       ROW_NUMBER() OVER (ORDER BY salary DESC) AS row_num
FROM employees;
```

NTILE() Purpose:
Divides the result set into a specified number of roughly equal parts (called "tiles"). It assigns a tile number to each row.
Example:

```
SELECT employee_id, salary,
       NTILE(4) OVER (ORDER BY salary DESC) AS salary_quartile
FROM employees;
```

This example divides employees into four groups (quartiles) based on their salary. The rows are assigned a tile number (from 1 to 4) based on their rank.

OVER() Clause :

The OVER() clause is essential for these ranking functions. It defines how the rows are partitioned (i.e., grouped) and ordered before the ranking is applied.

You can optionally use the PARTITION BY clause to group rows into partitions (like categories), and ORDER BY to specify the order within those partitions.
For example:

```
SELECT employee_id, department_id, salary,
    RANK() OVER (PARTITION BY department_id ORDER BY salary DESC)
    AS dept_salary_rank
FROM employees;
```

Here, RANK() is applied within each department_id, and employees are ranked by their salary within their respective departments.

Key Differences:

RANK():	Can have gaps in the ranking sequence when there are ties.
DENSE_RANK():	Does not have gaps in ranking, even when there are ties.
ROW_NUMBER():	Provides a unique number for each row, with no regard for ties.
NTILE():	Divides the result set into a specified number of tiles, assigning a tile number to each row.

VIEWS

- A **view** is like a "virtual table". A view in Oracle SQL is a virtual table that represents the result of a query.
- It does not store data physically but instead stores the SQL query that generates data dynamically when the view is accessed.

- Views are used to simplify complex queries, encapsulate logic, and improve security by restricting access to specific data within a table.
- Imagine we have a big table with lots of data, but we only want to show or use part of it in a simple way.
- So, instead of storing that data again, we create a **view** to show that data based on a query.
- It's just a shortcut to access the data that we need, without keeping a copy of it.

Types of Views

There are two main types of views in Oracle SQL:

1) Simple Views
2) Complex Views

Simple View :

A **Simple View** is a view that is based on a single table or a set of columns in a table. It does not contain any complex joins, subqueries, or group functions. It directly maps to a table and provides a straightforward view of the data.

When to use simple view :

- Use it when you just need to show specific columns from a table.
- For example, if a table has a lot of columns and you only need to see some of them, you make a simple view.
- Syntax :

```
CREATE VIEW <NameOfTheView>AS  <SQLProjectionQuery>
```

Complex View :

- A Complex View is a bit more advanced.
- It pulls data from multiple tables.
- We might use joins (to combine data), subqueries, and other advanced SQL tricks.
- This is useful when we need to combine data from different sources in a smart way.

When to use Complex Views :

- Use it when you need to create reports or combine data from multiple tables, or when you need to calculate sums, averages, etc.
- Basically, if the query you want is complicated, create a complex view.
- Syntax :

```
CREATE VIEW <ViewName> on <ComplexSqlJoinQuery>;
```

Other types of view :
Inline view : These are temporary views. We can create these types of views directly using query (like sub-query in from clause)

Example :

```
SELECT a.department_id, b.avg_salary
    FROM (SELECT department_id, AVG(salary) AS
avg_salary FROM employees
GROUP BY department_id) b;
```

Materialized Views:

A materialized view is a special type of view that stores the result physically, like a snapshot. This means you don't have to keep running the same complex query again and again. You just refresh the materialized view to get updated data when needed.

```
CREATE MATERIALIZED VIEW <view_name> AS
<Sql_Query>

REFRESH ON COMMIT;      // AUTO UPDATE
        (or)
REFRESH ON DEMAND;    // WE NEED TO UPDATE MANUALLY
```

Refresh M-View : EXEC DBMS_MVIEW.REFRESH('<ViewName>');

CREATE MATERIALIZED VIEW <VIEW_NAME> TABLESPACE users AS <SQL_QUERY>;

Updatable Views:

Some views are updatable, meaning you can modify the data in the base tables through the view (insert, update, delete). However, for a view to be updatable, it must meet certain criteria (e.g., it should not have complex joins, aggregates, or GROUP BY clauses).

If a view is not updatable, you can use INSTEAD OF triggers to provide functionality for updates.

Read-Only Views:

Views that involve complex operations (e.g., joins, aggregation, or subqueries) may be read-only. You cannot insert, update, or delete data through these views directly.

Limitation of view :

- Views may not support some operations, such as INSERT, UPDATE, or DELETE, when the view involves complex operations (joins, subqueries, aggregation).
- If the underlying table structure changes (e.g., column names or types), the view may become invalid.
- Views do not store data themselves, so querying a view may not perform as well as querying a base table directly, especially with complex queries.

DROP A VIEW:

```
DROP VIEW <VIEW_NAME>;
```

ROLES

- **Roles** are a collection of privileges (permissions) that can be granted to users or other roles.
- Roles help simplify the management of user privileges and are an essential part of Oracle's security model.
- Instead of assigning individual privileges to each user, we can create roles with a set of privileges and then assign those roles to users.
- This approach helps streamline privilege management, especially in large systems.

- **Privileges**: These are the rights or permissions granted to users to perform specific actions on database objects (e.g., SELECT, INSERT, UPDATE, DELETE).

- **Roles**: Roles are named groups of privileges. They can contain system privileges, object privileges, or both. Roles can be assigned to users or other roles.
- **System Privileges**: These are privileges that allow users to perform actions such as creating tables, creating views, or managing users.
- **Object Privileges**: These are privileges related to specific database objects, such as SELECT, INSERT, UPDATE, DELETE on tables or views.

CREATING A ROLE :

```
CREATE OR REPLACE ROLE <ROLE_NAME>;
```

Common Object Privileges:

SELECT: Allows the user to retrieve (query) data from a table, view, or synonym.
INSERT: Allows the user to add new rows of data into a table or view.
UPDATE: Allows the user to modify data in a table or view.
DELETE: Allows the user to remove rows of data from a table or view.
ALTER: Allows the user to modify the structure of a table or view (such as adding or removing columns).
DROP: Allows the user to delete a table, view, or other object.
INDEX: Allows the user to create or drop indexes on a table.
REFERENCES: Allows the user to create foreign key constraints that reference specific tables or columns.
EXECUTE: Allows the user to execute a stored procedure or function.

Role-Based Privileges : Role-based privileges refer to a role, which is a collection of privileges (both system and object privileges). Roles simplify privilege management

by grouping multiple privileges together and assigning them to users. Once a role is granted to a user, they automatically inherit all the privileges associated with that role.

Types of Roles:

- Predefined Roles:
  - DBA: Full administrative rights to perform all actions in the database (such as creating users, managing tables, etc.).
  - RESOURCE: Grants the ability to create and manage schema objects (e.g., tables, views).
  - CONNECT: Grants basic privileges to connect to the database and perform basic operations like creating tables and views.
  - SELECT_CATALOG_ROLE: Grants read access to Oracle's data dictionary views (system views).
  - EXECUTE_CATALOG_ROLE: Grants execute privileges on system procedures and functions in the data dictionary.

System Privileges: These are the rights that allow users to perform administrative actions at the system level. These privileges are not tied to specific objects but instead allow users to perform broad operations, such as creating users, managing tables, or altering database settings.

Common System Privileges:

- CREATE SESSION: Allows the user to connect to the database.
- CREATE TABLE: Allows the user to create new tables in the database.
- CREATE VIEW: Allows the user to create views in the database.
- CREATE PROCEDURE: Allows the user to create stored procedures or functions.
- ALTER SESSION: Allows the user to alter session-level settings.
- CREATE USER: Allows the user to create new database users.
- DROP USER: Allows the user to drop (delete) database users.
- GRANT ANY PRIVILEGE: Allows the user to grant any privilege to another user.
- CREATE ROLE: Allows the user to create roles.
- DROP ROLE: Allows the user to delete roles from the database.

.

Roles and Security:

Roles help enforce security policies by controlling access to database objects. By grouping privileges into roles and assigning those roles to users, administrators can maintain better control over who has access to which resources.

Assigning privileges to role :

GRANT <PrivilegesSet> TO <RoleName>

We can assign privileges to roles to access custom database objects.

GRANT <ExistingRoleName> TO <OtherRoleName>

GRANT <ROLE_NAME> TO <USER_NAME>

Revoke Powers from Role, User

REVOKE PRIVILEGE ON DB_OBJECT FROM <RoleName>

Dropping Roles :

DROP ROLE <ROLE_NAME>;

Profile

- An **Oracle Profile** is a collection of **resource limits** and **password management policies** that are applied to user accounts.
- Profiles allow DBAs to define constraints and security measures on individual user accounts or groups of users.
- They help in controlling how users interact with the database in terms of both system resource usage and password security.
- profiles are important because they help maintain the integrity and performance of the database by limiting how much resources a user can consume (e.g., CPU, memory, connections), and by enforcing password policies (e.g., password expiration, complexity).

SESSIONS_PER_USER :
This limit controls how many concurrent sessions a user can have. If a user reaches this limit, they are not allowed to create any more sessions.

CPU_PER_SESSION: pecifies the maximum CPU time (in hundredths of a second) that a user's session can consume. Once the limit is exceeded, the session is terminated.

CPU_PER_CALL:
Specifies the maximum CPU time (in hundredths of a second) that a single SQL call (i.e., query execution) can consume.

CONNECT_TIME:
Limits how long (in minutes) a user can be connected to the database in a session. After this period, the
session is disconnected automatically.

IDLE_TIME:
This sets the maximum amount of time (in minutes) a session can remain idle before being terminated. Idle means there's no SQL activity (queries, updates, etc.).

LOGICAL_READS_PER_SESSION:
Limits the number of logical reads (the number of blocks read into memory) that a user can perform in a session.

PRIVATE_SGA:
Limits the amount of memory that can be allocated for a user's private session in the Shared Global Area (SGA).

COMPOSITE_LIMIT:
This is a combination of limits like CPU, I/O, and other resource usage. It is used for controlling the overall resource usage of a session in a more granular way.

PASSWORD_LIFE_TIME:
Defines the maximum number of days a password remains valid. After this period, the user is required to change their password.

PASSWORD_GRACE_TIME:
Specifies the number of days after a password has expired that the user can still log in before being forced to change the password.

PASSWORD_REUSE_TIME:
Specifies the number of days that must pass before a user can reuse a previously used password.

PASSWORD_REUSE_MAX:
Defines how many previous passwords Oracle will remember. Users will be prevented from reusing these remembered passwords.

FAILED_LOGIN_ATTEMPTS:
Specifies the number of consecutive failed login attempts before the account is locked

LOCK_ACCOUNT:

Determines whether the account should be locked after exceeding the FAILED_LOGIN_ATTEMPTS threshold.

PASSWORD_VERIFY_FUNCTION:

Defines a custom function that checks the complexity of passwords. The function can enforce rules like minimum length, requiring upper/lowercase letters, numbers, etc.

Creating and Managing Profiles

Creating a Profile ::: CREATE PROFILE <PROFILE_NAME>;

We can create a profile using the CREATE PROFILE statement and can define both resource limits and password management policies.

Assigning profile to user(s) :
ALTER USER user_name PROFILE user_profile;

Modify the profile :
ALTER PROFILE user_profile LIMIT SESSIONS_PER_USER 10;

DROP profile :
DROP <PROFILE_NAME>

View Profile Info :
SELECT * FROM DBA_PROFILES WHERE PROFILE = 'USER_PROFILE';

Example :

CREATE PROFILE <PROFILE_NAME>
LIMIT
SESSIONS_PER_USER <NumberOfConnections2DB>
CPU_PER_SESSION <MaxCPUusagePerSessionInSeconds>
CONNECT_TIME <MaxContinuosConnectTimeInMinutes>
IDLE_TIME <MaxInactivityTImeToAutoLogOutInMinutes>
PASSWORD_LIFE_TIME <MaxPasswordLifeSpan>
FAILED_LOGIN_ATTEMPTS <AccountLockingAfter3FailedLoginAttemps>
PASSWORD_LOCK_TIME <AfterFailedAttemptsMaxTimeOfAccountLockInMinutes>
PASSWORD_REUSETIME <UserCanNotUseSamePasswordFor-IN-Days>

SEQUENCE

- **sequence** is an object that is used to generate a series of unique numbers.
- These numbers are commonly used for generating primary key values, unique identifiers, or any other purpose where you need a sequential number.
- In oracle SQL, we have only one type of sequence with several configurations.
- Syntax to create sequence :

```
CREATE SEQUENCE sequence_name
START WITH start_value
INCREMENT BY increment_value
[ MAXVALUE max_value | NOMAXVALUE ]
[ MINVALUE min_value | NOMINVALUE ]
[ CYCLE | NOCYCLE ]
[ CACHE cache_size | NOCACHE ]
[ ORDER | NOORDER ];
```

- sequence_name: The name of the sequence you are creating.
- START WITH start_value: The number at which the sequence starts (default is 1).
- INCREMENT BY increment_value: The value by which the sequence is incremented (default is 1).
- MAXVALUE max_value: The maximum value that the sequence can generate (default is NO MAXVALUE, which means the sequence can generate values up to the maximum value allowed by the datatype).
- MINVALUE min_value: The minimum value that the sequence can generate (default is 1).
- CYCLE | NOCYCLE: Specifies whether the sequence should start from the MINVALUE again once the MAXVALUE is reached. CYCLE allows it, while NOCYCLE does not.
- CACHE cache_size: Specifies the number of sequence numbers to cache for faster access. Default is 20.
- ORDER | NOORDER: Determines whether the sequence will guarantee the order of numbers or not (in a RAC environment).

Advantages of Using Sequences:
1. Automatic Generation of Unique Numbers: Sequences provide a mechanism to automatically generate unique values, thus reducing the need for manual assignment of values.
2. Independent of Table Data: Sequences are independent of table data. They generate numbers even if there is no corresponding row in the table.
3. Performance: Sequences improve performance when you need to generate unique identifiers quickly because they avoid locking rows or tables.
4. Customizable: Sequences can be customized to suit specific needs (start value, increment, cycle behavior, cache size, etc.).
5. Thread-safe (in non-RAC setups): Sequences can be safely used in multi-user environments without the need for additional locks, making it thread-safe in a single-instance Oracle database.

Disadvantages of Using Sequences:
1. No Rollback: Once a sequence value is generated (via NEXTVAL), it is consumed, and you cannot "rollback" that value even if the transaction is rolled back. This might lead to gaps in the sequence numbers.
2. RAC Issues: In a Real Application Cluster (RAC), sequences can generate non-ordered values unless you explicitly define the ORDER option. This can cause issues if you require a strictly ordered sequence of numbers.
3. Limited to Integer Values: The values generated by a sequence are of numeric type and are limited to the range of numbers supported by Oracle's numeric data types. Very large sequences may not be practical due to system resource limits.
4. Potential for Gaps: Because sequences are often cached for performance reasons, gaps can occur if the system crashes or if numbers are generated but not used. Additionally, the sequence may be incremented for failed insertions or rolled-back transactions.
Sequence and Transactions:
• The sequence values are generated before a transaction is committed, meaning they are issued as soon as NEXTVAL is called, not when the transaction is completed.
• ROLLBACK does not affect the sequence values, so if you get the next value and then roll back the transaction, the sequence will not "undo" the number generated.
When to Avoid Using Sequences:
1. When Strict Ordering Is Required in a Distributed Environment: If strict ordering is needed for sequential numbers, and you are working in a Real Application Cluster (RAC) or distributed environment, managing sequences might become tricky. You may need to use the ORDER option or consider other methods.

2. When You Require No Gaps in Sequence Numbers: If it is essential that no gaps exist in your numbering system, using a sequence may not be suitable due to the

possibility of gaps from failed transactions, cache management, or rolling back a transaction.

Alternatives to Sequences:
• Auto-increment in other databases (like MySQL or PostgreSQL): Some other databases provide automatic increment functionality directly tied to the table's column.
• Triggers: You could use triggers to generate numbers when inserting rows, although sequences are generally more efficient.
• UUID (Universally Unique Identifier): Sometimes, when unique values are required, a globally unique identifier (GUID) can be used instead of numeric sequences.

When to Use UUID/GUID Instead of Sequences:

Distributed Systems:
If you need unique identifiers across multiple machines or services (e.g., microservices or a distributed database setup), UUIDs are the better choice.
Security Concerns: If exposing numeric patterns or auto-incrementing IDs could be a security risk, UUIDs can provide better obfuscation and prevent such attacks.

Decentralized Applications:
If the system has no central authority and needs to generate unique IDs locally (like in offline applications or distributed systems), UUIDs are an excellent fit.

Data Merging:
If you're merging datasets from different systems or databases, UUIDs can help ensure that each record gets a globally unique identifier without conflicts.

Procedural Language / SQL

- procedural extension to Structured Query Language (SQL).
- It enables the integration of SQL with procedural constructs such as loops, conditions, variables, and exception handling.
- Designed specifically for Oracle databases, PL/SQL allows developers to write complex database-centric applications, including stored procedures, functions, triggers, and packages.
- This language enhances the functionality of SQL by supporting modularity, reusability, and performance optimization through execution of logic directly within the database.
- PL/SQL is tightly coupled with Oracle's SQL, providing seamless interaction with the database while maintaining security, efficiency, and error handling capabilities.

Advantages of PL/SQL

Tight Integration with Oracle Database: PL/SQL is tightly integrated with Oracle SQL. It allows you to write complex logic directly inside the database, improving performance by reducing the number of round-trips between an application and the database.

Improved Performance: With PL/SQL, the code is executed directly within the Oracle database. This reduces the need for repeated data retrieval between client and database, resulting in faster execution of complex logic.

Portability: PL/SQL code is portable across any platform that supports Oracle databases, making it easier to maintain and deploy.

Support for Modularity and Reusability: You can write modular code by creating procedures, functions, and packages in PL/SQL. These modules can be reused across multiple applications, improving maintainability and reducing redundancy.

Exception Handling: PL/SQL supports robust exception handling, allowing you to catch and handle errors gracefully. This is useful for ensuring smooth execution and better error diagnostics.

Security: With PL/SQL, sensitive logic can be embedded in stored procedures and functions within the database. This can help avoid exposing logic or critical data to the client side, ensuring security.

Triggers: You can define triggers that automatically fire when certain events occur in the database, such as when a row is inserted, updated, or deleted. This allows automation of tasks like data validation or auditing.

Block Structure

A PL/SQL block is the fundamental unit of execution in PL/SQL.
It consists of a collection of SQL statements, procedural constructs, and control structures that can be grouped together.

Here's an explanation of the complete structure and the elements of a PL/SQL block:

PL/SQL Block Structure
A PL/SQL block is divided into four sections:

Declaration Section (optional)
Execution Section (mandatory)
Exception Handling Section (optional)
End of the Block

1. **Declaration Section (optional)**
This is where you declare variables, constants, cursors, and types that will be used in the execution section. It is optional; not every PL/SQL block requires a declaration section.

Variables: Can be of any data type, such as numbers, strings, or more complex data types like records and collections.
Constants: Constants are variables that cannot change their value once assigned.

Constants: Constants are variables that cannot change their value once assigned.

Cursors: Named SQL queries that can be used to retrieve and process multiple rows of data.

Exceptions: Custom exception handlers can also be declared in this section.

```
DECLARE
  v_employee_name VARCHAR2(100);
  v_employee_id   NUMBER;
  v_salary        NUMBER(8, 2);
BEGIN
  -- Execution logic will go here
END;
```

Execution Section (mandatory)
The execution section is where the actual business logic or SQL statements are executed. This section contains the procedural code that interacts with the database, such as SELECT, INSERT, UPDATE, DELETE statements, loops, conditional statements, and assignments.

The execution section must always begin with the keyword BEGIN and end with END.

```
BEGIN
  -- SQL or PL/SQL code goes here
  SELECT employee_name INTO v_employee_name FROM employees
WHERE employee_id = 100;
```

```
    v_salary := 5000;
    -- Any other SQL or PL/SQL code
END;
```

Exception Handling Section (optional) :

This section is used to handle errors that occur during the execution of the SQL and PL/SQL statements in the execution section. You can handle specific exceptions or general ones. If no exception section is defined and an error occurs, it will cause the block to terminate abruptly.

You can define named exceptions (e.g., NO_DATA_FOUND, TOO_MANY_ROWS, etc.) or handle Oracle predefined exceptions.

This section starts with the keyword EXCEPTION.

End of the Block

Every PL/SQL block ends with the END; keyword. This marks the end of the block.

```
END;
```

Complete Block Structure :

```
DECLARE
  // Variables Section
BEGIN
 // Exception Section
EXCEPTION
  // Exception Handling Section
END;
```

Variables In PLSQL :

- variables are used to store temporary data that can be manipulated within the PL/SQL block.
- These variables are declared in the DECLARE section of a PL/SQL block and can store values of different data types like numbers, strings, and dates.

- Variable Declaration :
  Variables are declared in the DECLARE block section.

```
DECLARE
   variable_name data_type [:= initial_value];
BEGIN
   -- PL/SQL code
END;
```

- Assigning the values to variables

```
variable_name := value;
```

- In the execution block when ever we need to access the declared variable value on that we will use its name.
- Using variables in SQL statements
  Example :

```
DECLARE
  emp_id NUMBER(6);
  emp_name VARCHAR2(50);
BEGIN
  SELECT employee_id, employee_name
```

```
    INTO emp_id, emp_name
    FROM employees
    WHERE employee_id = 1001;
    DBMS_OUTPUT.PUT_LINE('Employee ID: ' || emp_id);
    DBMS_OUTPUT.PUT_LINE('Employee Name: ' || emp_name);
END;
```

❖ Variables are two types:

1) Local variables 2) Global Variables

Datatypes :

- **Numeric Types**: NUMBER, INTEGER, FLOAT, etc.
- **String Types**: VARCHAR2, CHAR, etc.
- **Date and Time Types**: DATE, TIMESTAMP, etc.
- **Boolean Type**: BOOLEAN (True/False)
- **Collections**: ARRAY, VARRAY, TABLE, NESTED TABLE, etc.
- **Composite Data Types:** RECORD, TABLE, REF TYPE (OBJECT, CURSOR)
- **Other types :** ROWID, UROWID, XMLTYPE
- **Pseudo types :** %ROWTYPE, %TYPE

composite data types are used to group multiple related data elements into a single structure.
This allows you to handle complex data in a more organized way.
The key composite data types include **RECORD**, **TABLE**, and **REF TYPE** (which can be further classified into **OBJECT** and **CURSOR** types).

A RECORD is used to group different types of data into a single unit, much like a row in a database table. Each field in a RECORD can hold a different datatype.

```
DECLARE
  -- Define a RECORD type with fields of different data types
  TYPE EmployeeRecord IS RECORD (
    emp_id    NUMBER,
    emp_name  VARCHAR2(100),
    hire_date DATE
  );

  -- Declare a variable of that RECORD type
  emp_info EmployeeRecord;
```

```
BEGIN
  -- Assign values to the fields of the RECORD
  emp_info.emp_id := 101;
  emp_info.emp_name := 'John Doe';
  emp_info.hire_date := SYSDATE;

  -- Output the values
  DBMS_OUTPUT.PUT_LINE('Employee ID: ' || emp_info.emp_id);
  DBMS_OUTPUT.PUT_LINE('Employee Name: ' || emp_info.emp_name);
  DBMS_OUTPUT.PUT_LINE('Hire Date: ' || emp_info.hire_date);
END;
```

A TABLE is used to store a collection of elements (like an array) of the same type. In PL/SQL, this is typically used for collections such as **Nested Tables** or **VARRAYs**.

```
DECLARE
  -- Define a Nested Table type to hold a list of employee IDs
  TYPE EmployeeTable IS TABLE OF NUMBER;

  -- Declare a variable of that Nested Table type
  emp_ids EmployeeTable := EmployeeTable(101, 102, 103);
BEGIN
  -- Add a new element to the Nested Table
  emp_ids.EXTEND;
  emp_ids(emp_ids.LAST) := 104;

  -- Loop through the Nested Table and print each element
  FOR i IN 1..emp_ids.COUNT LOOP
    DBMS_OUTPUT.PUT_LINE('Employee ID: ' || emp_ids(i));
  END LOOP;
END;
```

```
DECLARE
  -- Define a VARRAY type to hold a fixed number of employee IDs
  TYPE EmployeeVArray IS VARRAY(5) OF NUMBER;

  -- Declare a variable of that VARRAY type
```

```
  emp_ids EmployeeVArray := EmployeeVArray(101, 102, 103, 104, 105);
BEGIN
  -- Accessing elements from the VARRAY
  DBMS_OUTPUT.PUT_LINE('Employee ID: ' || emp_ids(1));  -- Outputs 101
END;
```

CONTROL STRUCTURES (IF, CASE, LOOP)

- ❖ control structures such as IF, CASE, and LOOP provide essential mechanisms for implementing conditional logic and iterative processes.
- ❖ These constructs enable dynamic decision-making and repetition of code execution, which are vital for complex data manipulation, reporting, and procedural operations.
- ❖ They are two types (1) Conditional Statements (2) Looping Statements

Conditional Statements:

IF-THEN-ELSE (Conditional Branching) :

The IF statement is used to perform conditional checks and execute code blocks based on specific criteria. It is commonly used in PL/SQL blocks, procedures, and functions to control the flow of execution depending on the result of a logical test.

```
IF <condition> THEN
  -- Executable statements when condition is TRUE
ELSIF <another_condition> THEN
  -- Executable statements for a second condition
ELSE
  -- Executable statements if none of the conditions are TRUE
END IF;
```

Example :

```
DECLARE
  v_salary NUMBER := 3500;
BEGIN
  IF v_salary > 5000 THEN
```

```
    DBMS_OUTPUT.PUT_LINE('High salary');
  ELSIF v_salary BETWEEN 3000 AND 5000 THEN
    DBMS_OUTPUT.PUT_LINE('Average salary');
  ELSE
    DBMS_OUTPUT.PUT_LINE('Low salary');
  END IF;
END;
```

CASE Expression :

The CASE expression is used to evaluate a series of conditions and return a result based on the first condition that is true. It is particularly useful for handling conditional logic directly within SQL queries. It can be seen as an alternative to IF-THEN-ELSE statements but is often preferred when working with SQL queries.

Types of CASE Expression

There are two types of CASE expressions in Oracle SQL:

1. **Simple CASE Expression**: This type compares a single expression to a series of possible values.
2. **Searched CASE Expression**: This type evaluates multiple Boolean expressions (conditions) and returns the result of the first condition that evaluates to true.

The **Simple CASE** expression evaluates a specific expression and compares it against a series of potential values. Based on the match, it returns a corresponding result.

```
SELECT CASE <expression>
        WHEN <value1> THEN <result1>
        WHEN <value2> THEN <result2>
        ELSE <default_result>
    END AS <alias>
FROM <table_name>;
```

Searched CASE Expression

The **Searched CASE** expression is more flexible because it evaluates multiple conditions (expressions) directly rather than comparing a single expression to

multiple values. The result is returned based on the first condition that evaluates to TRUE.

```
SELECT CASE
        WHEN <condition1> THEN <result1>
        WHEN <condition2> THEN <result2>
        ELSE <default_result>
    END AS <alias>
FROM <table_name>;
```

Looping Statements :

- **loops** are fundamental constructs used to execute a block of statements repeatedly.
- Loops are crucial in scenarios where an operation needs to be performed multiple times, such as processing records, performing calculations, or executing business logic iteratively.
- PL/SQL supports three primary types of loops:
    - Simple Loop,
    - FOR Loop,
    - WHILE Loop.

  Each loop type serves a distinct purpose depending on the requirement of the iteration.

Simple Loop :
Simple Loop is an unconditional loop that continues indefinitely until an explicit **EXIT** condition is met.

```
LOOP
  -- Executable statements
  EXIT WHEN <condition>;
END LOOP;
```

Useful when the number of iterations is unknown, and the loop must continue until a specific condition is true.
Example :-

```
DECLARE
  v_counter NUMBER := 1;
BEGIN
```

```
  LOOP
    DBMS_OUTPUT.PUT_LINE('Counter: ' || v_counter);
    v_counter := v_counter + 1;
    EXIT WHEN v_counter > 5;
  END LOOP;
END;
```

FOR Loop The **FOR Loop** is a counter-controlled loop where the number of iterations is known beforehand. The loop runs over a specified range of values, and the loop variable is automatically incremented.

```
FOR <counter> IN <start_value>..<end_value> LOOP
  -- Executable statements
END LOOP;
```

Ideal for scenarios where you know the exact number of iterations required.
Example :

```
FOR i IN 1..5 LOOP
  DBMS_OUTPUT.PUT_LINE('Counter: ' || i);
END LOOP;
```

WHILE Loop The **WHILE Loop** is a condition-controlled loop. It executes as long as the specified condition remains true. If the condition is false at the beginning, the loop will not execute.

```
WHILE <condition> LOOP
  -- Executable statements
END LOOP;
```

Appropriate when the number of iterations is not fixed, but the loop must continue based on a dynamic condition.

```
DECLARE
```

```
  v_counter NUMBER := 1;
BEGIN
  WHILE v_counter <= 5 LOOP
    DBMS_OUTPUT.PUT_LINE('Counter: ' || v_counter);
    v_counter := v_counter + 1;
  END LOOP;
END;
```

Cursors

A cursor is essentially a pointer that allows you to fetch rows one at a time from the result set of a query. There are two types of cursors in Oracle:

1. **Implicit Cursors**: Automatically created by Oracle when a SQL query is executed, such as SELECT, INSERT, UPDATE, or DELETE. These are used for single SQL operations, and Oracle manages them internally.
2. **Explicit Cursors**: These are user-defined cursors that allow you to explicitly fetch, process, and manipulate the result set row by row. This type of cursor is more flexible and allows for more control over the result set.

Implicit Cursor : An implicit cursor is created automatically by Oracle when executing DML (Data Manipulation Language) statements like SELECT INTO, INSERT, UPDATE, and DELETE. You can access implicit cursors using the predefined SQL cursor attributes.

```
SELECT employee_id, first_name, last_name
FROM employees
WHERE department_id = 10;
```

Explicit Cursor

An explicit cursor is defined by the user and gives more control over the result set. The following steps are involved in using an explicit cursor in Oracle SQL:

1. **Declare the Cursor**: This defines the query associated with the cursor.
2. **Open the Cursor**: This executes the query and sets up the result set.
3. **Fetch from the Cursor**: This retrieves individual rows from the result set.

4. **Close the Cursor**: This releases the cursor when it's no longer needed.

Cursor Attributes :

Some common cursor attributes include:

- **%FOUND**: Returns TRUE if the last fetch returned a row, and FALSE otherwise.
- **%NOTFOUND**: Returns TRUE if the last fetch did not return a row, and FALSE otherwise.
- **%ISOPEN**: Returns TRUE if the cursor is open, and FALSE otherwise.
- **%ROWCOUNT**: Returns the number of rows fetched.

Advantages of Using Cursors

- Allows you to process individual rows of a result set one at a time.
- More control over the result set, especially when complex logic is needed for processing each row.

Example :

```
DECLARE
   -- Declare the cursor
   CURSOR emp_cursor IS
      SELECT employee_id, first_name, last_name
      FROM employees
      WHERE department_id = 10;

   -- Declare variables to hold the fetched data
   emp_id employees.employee_id%TYPE;
   emp_first_name employees.first_name%TYPE;
   emp_last_name employees.last_name%TYPE;

BEGIN
   -- Open the cursor
   OPEN emp_cursor;

   -- Loop to fetch and process rows
   LOOP
      FETCH emp_cursor INTO emp_id, emp_first_name, emp_last_name;
```

```
        EXIT WHEN emp_cursor%NOTFOUND;

        -- Process the row (for example, display the data)
        DBMS_OUTPUT.PUT_LINE('Employee ID: ' || emp_id ||
                             ', First Name: ' || emp_first_name ||
                             ', Last Name: ' || emp_last_name);
    END LOOP;

    -- Close the cursor
    CLOSE emp_cursor;
END;
```

Cursor for Update :

```
DECLARE
   CURSOR emp_cursor IS
      SELECT employee_id, salary
      FROM employees
      WHERE department_id = 10
      FOR UPDATE;  -- Lock the selected rows for update

   emp_id employees.employee_id%TYPE;
   emp_salary employees.salary%TYPE;

BEGIN
   OPEN emp_cursor;

   LOOP
      FETCH emp_cursor INTO emp_id, emp_salary;
      EXIT WHEN emp_cursor%NOTFOUND;

      -- Update salary by 10%
      UPDATE employees
      SET salary = emp_salary * 1.10
```

```
    WHERE CURRENT OF emp_cursor;  -- Update the row that the cursor is pointing
to
  END LOOP;

  CLOSE emp_cursor;
END;
```

Note :

FOR UPDATE: Locks rows for update within a cursor.
FOR UPDATE OF: Locks specific columns for update.
WHERE CURRENT OF: Used for performing updates on rows that the cursor is pointing to.
WAIT / NOWAIT: Specifies whether the cursor should wait or fail if rows are locked by other transactions.
RETURNING INTO: Retrieves values from affected rows after DML operations

REF CURSOR :

Cursor Variables (REF CURSOR)

- A cursor variable is a pointer that can hold the result set of a query. It can be used to return a cursor from a function or procedure, allowing for dynamic query results.
- Cursor variables are useful when you want to return a result set from a stored procedure or function to a calling program.
- Syntax:

```
DECLARE
  -- Declare a ref cursor type
  TYPE ref_cursor_type IS REF CURSOR;
  -- Declare a variable of the cursor type
  my_cursor ref_cursor_type;
  -- Other variables (if needed)
  my_var VARCHAR2(100);
BEGIN
  -- Logic for opening the cursor and fetching data
END;
```

Cursor in procedure:

```
CREATE OR REPLACE PROCEDURE
    get_employee_details (p_cursor OUT SYS_REFCURSOR) AS
BEGIN
  -- Open the cursor to retrieve data
  OPEN p_cursor FOR
    SELECT employee_id, employee_name, department_id
    FROM employees
    WHERE department_id = 10;
END;
```

(If in cursor declaration return type is present then it is considered as strongly typed Other-wise that cursor can be taken as weakly typed cursor)

Exception Handling

- **Exception handling** refers to the mechanism that allows developers to handle runtime errors (exceptions) that occur during the execution of PL/SQL (Procedural Language/SQL) blocks.
- This feature helps you catch and respond to errors without causing the entire program or transaction to fail.
- Exception handling in Oracle SQL is primarily managed using the EXCEPTION block within a PL/SQL code block.

Syntax :

```
DECLARE
  -- Declare variables or exceptions
BEGIN
  -- Executable statements (your code)
EXCEPTION
  -- Exception handling
  WHEN <exception_name> THEN
    -- Handle the exception (e.g., log it, print a message, etc.)
  WHEN OTHERS THEN
    -- Handle any other exceptions
END;
```

Predefined Exceptions: These are common errors automatically provided by Oracle. Examples include:

- NO_DATA_FOUND: Raised when a SELECT INTO statement does not return any data.
- TOO_MANY_ROWS: Raised when a SELECT INTO statement returns more than one row.
- ZERO_DIVIDE: Raised when a division by zero occurs.
- OTHERS: A catch-all for any exceptions not explicitly handled.

User-Defined Exceptions: Developers can define their own exceptions for custom error conditions. These are usually raised using the RAISE statement.

The Structure of Exception Handling: An exception handling block typically includes:

- BEGIN block for the executable statements.
- EXCEPTION block for catching and handling exceptions.

QLERRM can be used to get the error message associated with an exception.

Example :

```
DECLARE
  v_salary NUMBER := 1000;
BEGIN
  -- Trying to divide by zero, which will raise an exception
  v_salary := v_salary / 0;
EXCEPTION
  WHEN ZERO_DIVIDE THEN
    DBMS_OUTPUT.PUT_LINE('Cannot divide by zero!');
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('An unexpected error occurred: ' || SQLERRM);
END;
```

Predefined Exceptions :

1. NO_DATA_FOUND
2. TOO_MANY_ROWS
3. ZERO_DIVIDE
4. VALUE_ERROR
5. INVALID_CURSOR
6. CURSOR_ALREADY_OPEN
7. DUP_VAL_ON_INDEX
8. LOGIN_DENIED
9. STORAGE_ERROR
10. PROGRAM_ERROR
11. COLLECTION_IS_NULL
12. SUBSCRIPT_OUTSIDE_LIMIT

13. TRANSACTION_BACKED_OUT
14. INVALID_NUMBER
15. DEADLOCK_DETECTED
16. ROWTYPE_MISMATCH
17. CASE_NOT_FOUND
18. INVALID_CURSOR
19. ACCESS_INTO_NULL
20. INVALID_NUMBER

Raising User Defined Exceptions:

```
DECLARE
  my_exception EXCEPTION;
BEGIN
  -- Some logic here
  IF some_condition THEN
    RAISE my_exception;
  END IF;
EXCEPTION
  WHEN my_exception THEN
    DBMS_OUTPUT.PUT_LINE('A custom error occurred');
  -- Handle other exceptions if necessary
END;
```

PROCEDURES (or) STORED PROCEDURE

Procedure :

- A **Stored Procedure** in Oracle is a collection of SQL statements and PL/SQL code that can be executed on demand, stored in the database, and reused multiple times.
- They will offer modular programming, help in managing logic on the database side, and can encapsulate complex business logic.
- Basic Syntax :

```
CREATE [OR REPLACE] PROCEDURE procedure_name
[ (parameter1 [mode] datatype, parameter2 [mode] datatype, ...) ]
IS
  -- Declare local variables here (optional)
BEGIN
  -- SQL and PL/SQL statements here
  -- Procedural logic goes here
EXCEPTION
  -- Handle exceptions (optional)
END procedure_name;
```

Modes :

- IN: Used to pass input parameters.
- OUT: Used to return output values.
- IN OUT: Used to pass values and also receive updated values.

When to use this :

1) Transaction Management
2) Batch Processing (Cleanup activities (logs), Report Generations)

FUNCTION :

- A Function in Oracle SQL is similar to a Stored Procedure but differs in that it returns a value.
- Functions are typically used to perform computations, return results, or transform data.
- They can be invoked directly in SQL queries, unlike stored procedures, which must be executed using the EXECUTE command or through other applications.
- A function is a named PL/SQL block that performs a task and returns a single value. It can take input parameters, perform an operation or calculation, and return a result to the calling environment.
- Basic Syntax :

```
CREATE [OR REPLACE] FUNCTION function_name
  (parameter1 [mode] datatype, parameter2 [mode] datatype, ...)
  RETURN return_datatype
IS
  -- Declare local variables here (optional)
BEGIN
  -- SQL and PL/SQL statements here
  -- Procedural logic goes here
```

```
  RETURN result;  -- return the value
EXCEPTION
  -- Handle exceptions (optional)
END function_name;
```

Function to return area of the circle :

```
CREATE OR REPLACE FUNCTION Calculate_Area (radius IN NUMBER)
  RETURN NUMBER
IS
  area NUMBER;
BEGIN
  area := 3.1416 * radius * radius;
  RETURN area;
END Calculate_Area;
```

Triggers :

- A **trigger** in Oracle SQL is a stored procedure that is automatically executed (or "fired") in response to certain events on a specified table or view.
- Triggers can be set to activate on insertions, updates, or deletions of data in the table, and they can help enforce business rules, data integrity, and automations within the database.

Key Components of a Trigger:

Event:

This is the action that causes the trigger to fire. It can be one of the following:

INSERT: Trigger fires when a new row is inserted into a table.
UPDATE: Trigger fires when an existing row is updated.
DELETE: Trigger fires when a row is deleted.

Timing:

Triggers can be set to execute either before or after the event.

BEFORE: The trigger fires before the data modification occurs.
AFTER: The trigger fires after the data modification has occurred.

Condition (Optional):
You can set conditions that determine when the trigger should execute based on the data involved in the event. This allows more granular control over the execution of the trigger.

Action: This is the code that gets executed when the trigger is fired. The action could be a simple statement or a complex set of operations.

Types of Triggers in Oracle:

Row-level Trigger:
The trigger fires once for each row affected by the event (INSERT, UPDATE, or DELETE). It has access to the individual column values that were inserted, updated, or deleted.

Example:
A BEFORE INSERT trigger might check if the inserted value meets certain conditions.

Statement-level Trigger: The trigger fires once per SQL statement, regardless of how many rows are affected.

Example:
An AFTER INSERT trigger might be used to log an action in an audit table every time an insertion is made.

Basic Syntax :

```
CREATE [OR REPLACE] TRIGGER trigger_name
  {BEFORE | AFTER} {INSERT | UPDATE | DELETE}
  ON table_name
  [FOR EACH ROW]  -- Optional for row-level triggers
  DECLARE
    -- Declare variables (optional)
  BEGIN
    -- Trigger action code (PL/SQL block)
  END;
```

{BEFORE | AFTER} {INSERT | UPDATE | DELETE}

This part specifies the event that will activate the trigger, and the timing of its execution:

- **BEFORE**: The trigger fires before the specified event (INSERT, UPDATE, DELETE) is executed. It allows you to modify or validate data before the database action takes place.
- **AFTER**: The trigger fires after the specified event has been executed. It allows you to perform actions based on the changes that have been made to the data (e.g., logging, auditing, or modifying related data).

The **event types** are:

- **INSERT**: The trigger fires when a new row is inserted into the table.
- **UPDATE**: The trigger fires when an existing row is updated in the table.
- **DELETE**: The trigger fires when a row is deleted from the table.

[FOR EACH ROW]

- This is an optional clause used for **row-level triggers**. When you specify FOR EACH ROW, the trigger will fire once for each row affected by the event.
- You will also have access to the old and new values for each row being affected.
- Without this clause, the trigger is a **statement-level trigger**, which fires once for the entire SQL statement, not for each individual row.

:NEW:

- Refers to the **new values** of the columns in the row being inserted or updated.
- It is only available in **INSERT** and **UPDATE** triggers.
- For **INSERT**, :NEW contains the values that will be inserted.
- For **UPDATE**, :NEW contains the values that will be updated.

:OLD:

- Refers to the **old values** of the columns in the row being updated or deleted.
- It is only available in **UPDATE** and **DELETE** triggers.
- For **UPDATE**, :OLD contains the original values of the row before the update.
- For **DELETE**, :OLD contains the values of the row being deleted.

:ROWID:

- Refers to the unique identifier of the row being affected. You can use this pseudo-variable to directly reference the physical location of the row in the database.
- Useful in both **UPDATE** and **DELETE** triggers.

PACKAGES

packages are a way to group related procedures, functions, variables, and other elements into a single unit. They help in organizing and managing code more efficiently. A package consists of two main components:

1. **Package Specification**: This defines the interface to the package. It declares the public procedures, functions, variables, cursors, and types that can be accessed from outside the package.
2. **Package Body**: This contains the actual implementation of the procedures, functions, and other elements declared in the package specification. It's where the logic of the package resides.

Benefits of using Packages:

- **Encapsulation**: Packages allow you to hide implementation details and expose only necessary parts to the outside world.
- **Improved Performance**: Once a package is loaded into memory, all procedures and functions are available without the need to load them again.

- **Modularity**: Packages provide a way to logically organize and group related PL/SQL code.
- **Security**: Packages help in controlling access to database objects, allowing finer control over who can access or modify certain parts of the package.

Package Specifications:

```
CREATE OR REPLACE PACKAGE package_name AS
   -- Public procedure declaration
   PROCEDURE procedure_name(parameter_list);
   -- Public function declaration
   FUNCTION function_name (parameter_list) RETURN
return_datatype;
   -- Public variable declaration
   variable_name datatype;
   -- Public cursor declaration
   CURSOR cursor_name IS SELECT_statement;
END package_name;
/
```

Package Body Syntax :

```
CREATE OR REPLACE PACKAGE BODY package_name AS
   -- Procedure implementation
   PROCEDURE procedure_name(parameter_list) IS
   BEGIN
      -- Procedure logic
   END procedure_name;
   -- Function implementation
   FUNCTION function_name(parameter_list) RETURN return_datatype IS
   BEGIN
      -- Function logic
      RETURN some_value;
```

```
    END function_name;
END package_name;
/
```

Collections in Oracle SQL (PL/SQL)

- ❖ Collections are **data structures** that allow storing and manipulating multiple values in a **single variable**.
- ❖ Oracle provides three types of collections:
  - (1) Associative Arrays (Index-by tables)
  - (2) Nested tables
  - (3) VARRAYs (Variable Size Arrays)

Associative Arrays :

An **associative array** (also known as an **index-by table**) is a **key-value pair collection** in PL/SQL. It functions like a **hash table or dictionary** in other programming languages.

Unbounded – No fixed size.
Sparse – Indexes can be non-sequential.
In-memory only – Cannot be stored in the database.
Fast lookup – Uses **integer or string keys** for fast access.

Syntax :

```
TYPE array_type IS TABLE OF element_type INDEX BY index_type;
```

- element_type → The type of values stored in the array (e.g., NUMBER, VARCHAR2).
- index_type → The type of the key (either PLS_INTEGER or VARCHAR2).

```
DECLARE
    -- Declare an associative array type
    TYPE emp_array IS TABLE OF VARCHAR2(50) INDEX BY PLS_INTEGER;
    -- Declare a variable of that type
    employees emp_array;
BEGIN
    -- Assign values using integer keys
    employees(101) := 'John';
    employees(102) := 'Jane';
    employees(103) := 'Robert';
```

```
    -- Retrieve and display a value
    DBMS_OUTPUT.PUT_LINE('Employee 101: ' || employees(101));
END;
/
```

Operations on Associative Arrays

Operation	Description
`array(index) := value;`	Assign a value to a specific key
`array.DELETE;`	Delete all elements
`array.DELETE(index);`	Delete a specific element
`array.COUNT;`	Get the number of elements
`array.EXISTS(index);`	Check if a key exists
`array.FIRST;`	Get the first index
`array.LAST;`	Get the last index
`array.NEXT(index);`	Get the next index
`array.PRIOR(index);`	Get the previous index

```
DECLARE
   TYPE emp_table IS TABLE OF VARCHAR2(50) INDEX BY PLS_INTEGER;
   employees emp_table;
   idx PLS_INTEGER;
BEGIN
   -- Assigning values
   employees(101) := 'John';
   employees(102) := 'Jane';
   employees(104) := 'Robert';  -- Skipping 103 (sparse)
```

```
   -- Looping through the array using FIRST and NEXT
   idx := employees.FIRST;
   WHILE idx IS NOT NULL LOOP
      DBMS_OUTPUT.PUT_LINE('Employee ID: ' || idx || ', Name: ' || employees(idx));
      idx := employees.NEXT(idx);  -- Move to next index
   END LOOP;
END;
/
```

Nested Tables

- A nested table is a collection type in PL/SQL that behaves like a one-dimensional array.
- Unlike associative arrays, nested tables can be stored in the database.
- They are useful when you need a flexible, dynamic collection that supports storage, retrieval, and modification of elements.
    - Can be stored in a database table
    - Unbounded size (can grow dynamically)
    - Sparse (elements can be deleted, leaving gaps)
    - Supports set operations (like SQL queries on collections)

Syntax :

```
TYPE nested_table_type IS TABLE OF element_type;
```

```
DECLARE
   -- Define a nested table type
   TYPE num_table IS TABLE OF NUMBER;

   -- Declare a variable of that type
   numbers num_table := num_table(10, 20, 30, 40);
BEGIN
   -- Adding an element dynamically
   numbers.EXTEND;
   numbers(5) := 50;
```

```
    -- Accessing elements
    DBMS_OUTPUT.PUT_LINE('First Element: ' || numbers(1));
    DBMS_OUTPUT.PUT_LINE('Last Element: ' || numbers(5));
END;
/
```

Operations on Nested Tables

Operation	Description
table_var.EXTEND(n);	Add n empty elements
table_var.DELETE;	Remove all elements
table_var.DELETE(i);	Delete element at index i
table_var.COUNT;	Get the number of elements
table_var.EXISTS(i);	Check if index i exists
table_var.FIRST;	Get the first index
table_var.LAST;	Get the last index
table_var.NEXT(i);	Get the next index
table_var.PRIOR(i);	Get the previous index

VARRAYS :

A **VARRAY** (Variable-Size Array) is a type of collection in Oracle PL/SQL that stores a **fixed number of elements**. Unlike **nested tables**, VARRAYs have a **predefined size limit**, meaning they **cannot grow beyond the specified limit**.

- ☑ **Can be stored in a database table**
- ☑ **Fixed size** (must be declared at creation)
- ☑ **Dense** (no gaps; all elements exist within the range)
- ☑ **Maintains order** (elements are stored sequentially)

Syntax :

TYPE varray_type IS VARRAY(n) OF element_type;

- `n` → Maximum number of elements allowed.
- `element_type` → The type of values stored in the collection (`NUMBER`, `VARCHAR2`, etc.).

Operation	Description
varray_var.EXTEND(n);	Add n empty elements (only if space is available)
varray_var.DELETE;	Remove all elements
varray_var.DELETE(i);	Delete element at index i
varray_var.COUNT;	Get the number of elements
varray_var.EXISTS(i);	Check if index i exists
varray_var.FIRST;	Get the first index
varray_var.LAST;	Get the last index

DYNAMIC SQL :

- Dynamic SQL in Oracle SQL refers to SQL statements that are constructed and executed at runtime rather than being hardcoded in the application.
- Oracle provides **Native Dynamic SQL (NDS)** and **DBMS_SQL package** to handle dynamic SQL execution.

Native Dynamic SQL (NDS) using EXECUTE IMMEDIATE

Oracle provides the EXECUTE IMMEDIATE statement to execute dynamic SQL. It is simpler and more efficient than using DBMS_SQL.

Execute Basic SQL QUERY :

```
BEGIN

   EXECUTE IMMEDIATE 'CREATE TABLE test_table (id NUMBER, name
VARCHAR2(50))';

END;

/
```

Inserting data using BIND VARIABLES

```
DECLARE

   v_id NUMBER := 1;

   v_name VARCHAR2(50) := 'John Doe';

BEGIN

   EXECUTE IMMEDIATE 'INSERT INTO test_table (id, name) VALUES (:1, :2)'
USING v_id, v_name;

END;

/
```

SELECTING DATA INTO VARIABLES

```
DECLARE

   v_name VARCHAR2(50);

BEGIN
```

```
   EXECUTE IMMEDIATE 'SELECT name FROM test_table WHERE id = :1' INTO v_name
USING 1;

   DBMS_OUTPUT.PUT_LINE('Name: ' || v_name);

END;

/
```

DROPPING A TABLE (ddl)

```
BEGIN

   EXECUTE IMMEDIATE 'DROP TABLE test_table';

END;

/
```

Dynamic SQL Using DBMS_SQL Package

The DBMS_SQL package is useful when:

- The number of bind variables is unknown.
- The SQL statement type is dynamic (e.g., it could be a SELECT, INSERT, or UPDATE).
- The statement needs to be parsed dynamically.

Dynamic Query Execution :

```
DECLARE
   v_cursor   NUMBER;
   v_query    VARCHAR2(200);
   v_result   VARCHAR2(50);
BEGIN
   v_query := 'SELECT name FROM test_table WHERE id = :id';
   v_cursor := DBMS_SQL.OPEN_CURSOR;
   DBMS_SQL.PARSE(v_cursor, v_query, DBMS_SQL.NATIVE);
   DBMS_SQL.BIND_VARIABLE(v_cursor, ':id', 1);
   DBMS_SQL.DEFINE_COLUMN(v_cursor, 1, v_result, 50);
   IF DBMS_SQL.EXECUTE_AND_FETCH(v_cursor) > 0 THEN
      DBMS_SQL.COLUMN_VALUE(v_cursor, 1, v_result);
      DBMS_OUTPUT.PUT_LINE('Result: ' || v_result);
   END IF;
   DBMS_SQL.CLOSE_CURSOR(v_cursor);
EXCEPTION
   WHEN OTHERS THEN


     IF DBMS_SQL.IS_OPEN(v_cursor) THEN
        DBMS_SQL.CLOSE_CURSOR(v_cursor);
      END IF;
      RAISE;
```

```
END;

/
```

REF CURSORS

Cursor variables in Oracle SQL are a special type of cursor that allows for dynamic query execution. Unlike static cursors, which are tied to a specific query, cursor variables can be associated with different queries at runtime. They are particularly useful in PL/SQL for handling multiple result sets dynamically.

Key Features of Cursor Variables:

1. **Flexibility**: Cursor variables can be assigned different queries dynamically, making them more flexible than static cursors.
2. **Scope**: They can be passed as parameters between procedures and functions, allowing for modular code design.
3. **Memory Management**: Cursor variables are automatically managed by the PL/SQL engine, reducing the risk of memory leaks.
4. **Ref Cursor**: Cursor variables are implemented using the REF CURSOR type, which is a reference to a cursor.

Declaring and Using Cursor Variables:

Define a REF CURSOR type:

```
TYPE ref_cursor_type IS REF CURSOR;
```

Declare a cursor variable:

```
my_cursor_variable ref_cursor_type;
```

Open a cursor variable dynamically:

OPEN my_cursor_variable FOR SELECT * FROM employees;

Fetch data from the cursor:

FETCH my_cursor_variable INTO variable1, variable2;

Close the cursor variable:

CLOSE my_cursor_variable;

Example Usage:

```
DECLARE
   TYPE emp_cursor IS REF CURSOR;
   emp_cur emp_cursor;
   emp_record employees%ROWTYPE;
BEGIN
   OPEN emp_cur FOR SELECT * FROM employees WHERE department_id = 10;

   LOOP
      FETCH emp_cur INTO emp_record;
      EXIT WHEN emp_cur%NOTFOUND;
      DBMS_OUTPUT.PUT_LINE(emp_record.first_name || ' ' ||
emp_record.last_name);
   END LOOP;

   CLOSE emp_cur;
END;
/
```

Benefits of Cursor Variables:

- **Reusability**: Can be used across different queries dynamically.
- **Performance**: Reduces overhead compared to opening and closing static cursors multiple times.
- **Modularity**: Allows passing result sets between procedures and functions efficiently.

Cursor variables provide a powerful way to handle query execution dynamically in PL/SQL, making them a vital tool for database developers.

BULK PROCESSING :

- Bulk processing refers to the ability to process multiple rows of data efficiently in a single operation rather than processing rows one at a time.

- This is particularly useful when dealing with large volumes of data because it significantly reduces the overhead associated with multiple context switches between SQL and PL/SQL engines.
- By reducing the number of context switches, bulk processing improves performance and resource utilization.

There are two main techniques in Oracle SQL for bulk processing:

Bulk Collect

The BULK COLLECT clause allows you to fetch multiple rows from a query into PL/SQL collections (like **associative arrays**, **nested tables**, or **varrays**) all at once. This is much more efficient than fetching rows one at a time.

```
DECLARE
  TYPE t_id_array IS TABLE OF employees.employee_id%TYPE;
  TYPE t_name_array IS TABLE OF employees.first_name%TYPE;

  v_ids   t_id_array;
  v_names t_name_array;
BEGIN
  SELECT employee_id, first_name
  BULK COLLECT INTO v_ids, v_names
  FROM employees
  WHERE department_id = 10;

  -- Process the data in memory
  FOR i IN 1..v_ids.COUNT LOOP
    DBMS_OUTPUT.PUT_LINE(v_ids(i) || ': ' || v_names(i));
  END LOOP;
END;
```

FORALL

The FORALL statement is used for performing DML (Data Manipulation Language) operations (like INSERT, UPDATE, or DELETE) in bulk on collections. It allows you to execute the DML operation for all elements in a collection without the need to loop through each individual element.

```
DECLARE
  TYPE t_id_array IS TABLE OF employees.employee_id%TYPE;
  v_ids t_id_array := t_id_array(100, 101, 102);

BEGIN
  FORALL i IN 1..v_ids.COUNT
    UPDATE employees
    SET salary = salary * 1.1
    WHERE employee_id = v_ids(i);

  COMMIT;
END;
```

Key Benefits of Bulk Processing:

- **Performance:** By reducing the context switches between SQL and PL/SQL, bulk operations are much faster than processing each row individually.
- **Resource Efficiency:** Minimizes the number of context switches and network round trips, which is important when processing large datasets.
- **Simplicity:** Bulk operations can simplify complex PL/SQL code by allowing you to handle multiple records in a single statement.

Performance Tuning :

PL/SQL performance tuning in Oracle involves optimizing PL/SQL code to make it execute faster and use fewer resources, such as CPU, memory, and disk. Here's an in-depth explanation of how to achieve better performance in PL/SQL:

1. Efficient SQL Queries

Since PL/SQL works extensively with SQL queries, optimizing SQL queries is the first step in enhancing performance. Poor SQL performance can affect the entire PL/SQL block.

- **Use Proper Indexes**: Ensure appropriate indexes exist on the tables that are being queried. An index on the columns that are frequently used in WHERE, JOIN, and ORDER BY clauses can reduce query time.
- **Minimize Full Table Scans**: Avoid full table scans when unnecessary. Use indexed columns or ensure that indexes are available for columns used in filtering and joining.
- **Avoid SELECT * Statements**: Retrieve only the necessary columns rather than using SELECT *. This reduces the amount of data transferred and improves performance.
- **Optimize Joins**: Avoid complex joins when simpler joins will suffice. Be careful with outer joins (LEFT JOIN, RIGHT JOIN) as they are more expensive.
- **Use Bind Variables**: Using bind variables helps to reduce the overhead associated with parsing SQL statements, as the same SQL query can be reused with different values.

Use BULK Collect: When retrieving large sets of rows, avoid processing each row individually with a FOR loop. Instead, use BULK COLLECT to fetch multiple rows at once into a collection (e.g., an array). This minimizes context switching between SQL and PL/SQL engines and improves speed

```
FORALL i IN 1..some_collection.count
   UPDATE my_table
   SET col = some_collection(i)
   WHERE id = some_collection(i);
```

Use FORALL for DML Operations: If you need to insert, update, or delete many rows, use FORALL instead of looping through each row and executing separate DML statements. FORALL allows the database to perform these operations in bulk, which significantly reduces context switching between SQL and PL/SQL engines.

```
DECLARE
  TYPE num_list IS TABLE OF NUMBER;
  numbers num_list := num_list(1, 2, 3, 4, 5);
BEGIN
  FORALL i IN INDICES OF numbers
     UPDATE my_table SET column_name = numbers(i) WHERE id = i;
END;
```

Minimize Context Switching Between SQL and PL/SQL

Every time control switches between SQL and PL/SQL, it incurs a performance cost. This is called **context switching**, and it can significantly degrade performance in complex applications.

- **Use PL/SQL Arrays/Collections**: Instead of executing multiple SQL queries, collect data into PL/SQL collections (like VARRAY, nested tables, or associative arrays) and process it in memory.
- **Limit the Number of SQL Statements**: Reducing the number of SQL statements inside your PL/SQL block can minimize the context switching. Combining multiple DML operations into a single statement where possible (such as using MERGE) helps.

Transaction Control :

Transaction Control in Oracle SQL refers to the management of the work or operations within a database that happen as a single unit. It ensures the consistency and integrity of data, allowing for multiple operations to be grouped together as one logical unit.

A **transaction** is a sequence of one or more SQL operations that are executed as a single unit. A transaction can include commands like INSERT, UPDATE, DELETE, or even SELECT. The key properties of a transaction are defined by the **ACID** properties:

- **Atomicity**: All operations in a transaction are treated as a single unit. If one operation fails, the entire transaction fails.
- **Consistency**: The transaction ensures that the database moves from one consistent state to another.
- **Isolation**: Transactions are executed independently, without interference from other transactions.
- **Durability**: Once a transaction is committed, its changes are permanent.

SET PROPERTIES TO THE TRANSACTION :

```
SET TRANSACTION [READ WRITE | READ ONLY];
```

- **READ COMMITTED** (default): A transaction can only read committed data; it won't see uncommitted changes from other transactions.
- **SERIALIZABLE**: Guarantees the highest level of isolation, ensuring no other transactions can access the data being modified until the transaction is complete.
- **READ ONLY**: Ensures that a transaction can only perform SELECT operations and cannot modify data.
- **READ UNCOMMITTED**: Allows a transaction to read uncommitted data from other transactions (dirty reads).

LOCKING :

A **transaction** is a sequence of SQL operations executed as a single unit of work. The **isolation level** defines the degree to which one transaction's changes are visible to other transactions.

```
-- Begin transaction
BEGIN;

-- Update salary for employee
UPDATE employees SET salary = salary + 1000 WHERE employee_id = 1001;

-- Perform another operation
UPDATE employees SET salary = salary + 500 WHERE employee_id = 1002;

-- Commit the changes (or rollback if needed)
COMMIT;
```

Read Committed (Default One) :

```
-- This reads only committed data from the table or view.
SELECT salary FROM employees WHERE employee_id = 1001;
```

Serializable: Guarantees the highest level of isolation. It makes the transaction behave as if it is executed serially (one after another).

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

Optimistic Locking

In **optimistic locking**, transactions are allowed to read and modify data without acquiring locks upfront. When the transaction tries to commit, it checks whether any changes were made by other transactions in the meantime

```
SELECT employee_id, salary, version FROM employees WHERE employee_id = 1001;
-- Output: 1001 | 3000 | 1
```

Row-Level Locking with `SELECT FOR UPDATE`

In cases where you want to lock specific rows that are returned by a query, you use `SELECT FOR UPDATE`. This allows you to safely modify the rows without interference from other transactions until the transaction is committed.

```
-- Lock rows where department_id = 10
SELECT * FROM employees WHERE department_id = 10 FOR UPDATE;

-- Perform operations on locked rows
UPDATE employees SET salary = salary + 1000 WHERE department_id = 10;

-- Commit the transaction
COMMIT;
```

Table Locking

Table locking is used when you want to prevent all other transactions from accessing a table, which can be useful for certain administrative tasks (e.g., maintenance, bulk data operations).

Pessimistic vs Optimistic Concurrency Control

- **Pessimistic Concurrency Control** assumes that conflicts will happen, so it locks the data upfront (using locks).
- **Optimistic Concurrency Control** assumes that conflicts are rare and checks for conflicts at commit time (without locking data initially).

Pessimistic Example (Locking):

```
-- Transaction A locks the row
```

SELECT * FROM employees WHERE employee_id = 1001 FOR UPDATE;

-- Transaction B will be blocked if it tries to access the same row until Transaction A commits.

Optimistic Example (Version Check):

-- Transaction A reads the data
SELECT salary, version FROM employees WHERE employee_id = 1001;

-- Transaction B reads the same data and updates it
UPDATE employees SET salary = salary + 500 WHERE employee_id = 1001;

If **Transaction A** tries to commit its changes, it will check if the version has changed. If so, the transaction is rolled back to prevent overwriting data.