Unit – I Introduction to Database System and SQL commands

Concepts and Definitions: Database and database systems and database environment

Database:

Definition: A database is a structured collection of data that is organized in a way to facilitate efficient storage, retrieval, and manipulation of information. It can be thought of as an electronic filing system.

Database System:

Definition: A database system is a combination of software and hardware that allows for the creation, management, and utilization of databases. It includes database management system (DBMS) software, the database itself, and the associated applications and users.

Database Management System (DBMS):

Definition: A DBMS is software that provides an interface for interacting with databases. It enables users to define, create, retrieve, update, and manage data in a database. Popular examples include MySQL, Oracle, Microsoft SQL Server, and PostgreSQL.

Database Environment:

Definition: The database environment refers to the surroundings and conditions in which a database system operates. It encompasses various elements, including hardware, software, data, procedures, users, and the overall infrastructure.

Components of Database Environment:

a. Hardware: The physical equipment, such as servers and storage devices, that supports the database system.
b. Software: Includes the DBMS software, operating system, and any other applications interacting with the database.
c. Data: The actual information stored in the database, organized in a structured manner.
d. Procedures: Guidelines and rules governing the use and maintenance of the database.
e. Users: Individuals or applications that interact with the database system, such as administrators, developers, and end-users.
f. Infrastructure: The overall network and communication setup that supports the database system.

Database Architecture:

Definition: The design or structure that defines how the components of a database system are organized and interact. Common architectures include client-server, three-tier, and peer-to-peer.

Data Model:

Definition: A data model is a conceptual representation of the data and its relationships within a database. Common data models include the relational model, hierarchical model, and object-oriented model.

Database Schema:

Definition: The blueprint or structure that defines the organization of data in a database. It includes tables, fields, relationships, and constraints.

Query Language:

Definition: A language that allows users to interact with a database by querying and manipulating data. SQL (Structured Query Language) is a widely used query language for relational databases.

terms: Data, Information, Records, Fields, Metadata, Data warehouse, Data dictionary

Data:

Raw and Unprocessed: Data is unorganized, raw information in its basic form, lacking structure or context.
Variety: Data comes in various types, such as text, numbers, images, and more, contributing to the concept of big data.
Prevalence: Data is omnipresent in our digital world, constantly generated by various activities and devices.
Basis for Information: Data serves as the foundation for generating meaningful information when processed and analyzed.

Information:

Processed Data: Information results from the processing and analysis of raw data, adding context and significance.
Contextual Relevance: Information is meaningful and relevant, providing insights and understanding of a particular subject.
Decision-Making: Information empowers decision-making processes by offering valuable insights and knowledge.
Transformative: Information transforms data into a usable and comprehensible resource for individuals and organizations.

Records:

Entity Representation: Records represent individual entities or objects in a database, grouping related data fields together.
Structured Format: Records are organized in a structured format within a database table, facilitating efficient data management.
Database Fundamental: Records are fundamental units in relational database systems, forming the basis for data storage and retrieval.
Information Storage: Information about a specific entity is stored in a record, making it a key element in database organization.

Fields:

Attribute Representation: Fields represent individual attributes or properties of an entity within a record.
Data Types: Each field has a defined data type (e.g., text, number, date) specifying the kind of data it can store.
Column in a Table: In a database table, fields are equivalent to columns, and each record is a row.
Database Schema: Fields, along with their data types, contribute to the overall database schema, defining the structure of the data.

Metadata:

Data About Data: Metadata is information that describes and provides context for other data.
Structured Information: Metadata includes details like data definitions, data types, relationships, and source information.
Data Management: Metadata is essential for effective data management, aiding in data governance and quality control.
Documentation: Acts as a comprehensive documentation tool, ensuring clarity and consistency in understanding data elements.

Data Warehouse:

Centralized Repository: A data warehouse serves as a central storage location for large volumes of structured data.
Business Intelligence: Designed to support business intelligence, data warehouses enable efficient querying and reporting.
Integration of Data: Involves the integration of data from diverse sources to provide a comprehensive view.
Historical Perspective: Data warehouses often store historical data, allowing for trend analysis and long-term insights.

Data Dictionary:

Metadata Repository: A data dictionary serves as a repository for metadata, containing information about database elements.
Definitions and Descriptions: Includes definitions, descriptions, and attributes of data elements, tables, and relationships.
Reference Tool: Acts as a reference tool for database administrators, developers, and users to understand the database structure.

Consistency and Standards: Ensures consistency and adherence to standards in data definitions and usage within an organization.

Schemas, Sub-schemas, and Instances:

Schema (Database Schema):

Definition: A database schema is a logical design or blueprint that represents the overall structure of the database. It defines how data is organized and how relationships among data elements are maintained.
Key Points:
Tables and Relationships: Schemas include tables, their fields, and the relationships between tables.
Data Integrity: Specifies constraints, such as primary keys and foreign keys, to maintain data integrity.
Security: Defines access permissions for users and roles.
Organizational Structure: Provides an organizational structure for the database, facilitating efficient data storage and retrieval.

Sub-schema:

Definition: A sub-schema is a subset of a larger schema. It represents a portion of the database schema that is relevant to a particular group of users or applications.
Key Points:
Customization: Allows different views of the data for specific users or applications.
Security Focus: Sub-schemas can be used to limit access to only the relevant parts of the database.
Simplification: Provides a simplified view for users, hiding unnecessary details.
Modularity: Enhances modularity by breaking down the overall schema into manageable components.

Instance:

Definition: An instance refers to a specific occurrence or snapshot of the data in a database at a given moment. It represents the actual content stored in the database at a particular point in time.
Key Points:
Concrete Data: Instances are concrete representations of data, reflecting real-world information.
Dynamic Nature: Changes over time as data is inserted, updated, or deleted.
Record Rows: In a relational database, an instance is often a specific row or record in a table.
Temporal Aspects: Instances capture the temporal aspect of the data, reflecting its state at a specific moment.

In summary:

Schema: The overall logical design of a database, defining its structure and organization.

Sub-schema: A subset of a larger schema, providing a customized view for specific users or applications.
Instance: A specific occurrence or snapshot of the data in a database at a particular moment, representing the actual content stored.

DBMS Data types

In a Database Management System (DBMS), data types are used to define the nature of data that can be stored in a particular column of a table. Different database systems may have variations in the supported data types, but here are some common data types found in many relational database management systems (RDBMS):

Numeric or Integer Types:

INT (Integer): Represents whole numbers.
TINYINT, SMALLINT, BIGINT: Different sizes of integers with varying ranges.

Floating-Point Types:

FLOAT: Represents single-precision floating-point numbers.
DOUBLE or REAL: Represents double-precision floating-point numbers.

Decimal Types:

DECIMAL or NUMERIC: Fixed-point numbers with exact precision and scale.

Character String Types:

CHAR(n): Fixed-length character string with a specified length 'n'.
VARCHAR(n): Variable-length character string with a maximum length of 'n'.
TEXT: Variable-length character string with a larger storage capacity than VARCHAR.

Date and Time Types:

DATE: Represents a date (year, month, day).
TIME: Represents a time of day.
DATETIME or TIMESTAMP: Represents a combination of date and time.

Boolean Type:

BOOLEAN or BOOL: Represents true or false values.

Binary Large Object Types (BLOB):

BLOB: Stores binary data, such as images, audio, or video files.

Character Large Object Types (CLOB):

CLOB: Stores large amounts of character data.

Row Identifier Types:

UUID: Represents a universally unique identifier.

Enumerated Types (ENUM):

ENUM: Represents a set of predefined values.

JSON Types:

JSON: Stores JSON-formatted data.

XML Type:

XML: Stores XML-formatted data.

These data types provide a way to define the kind of data that can be stored in each column of a table, ensuring consistency and integrity within the database. It's important to consult the specific documentation of the DBMS being used for the precise details and variations in data type implementation.

Database Language commands: Data Definition Language (DDL): CREATE, ALTER, TRUNCATE, DROP

In a Database Management System (DBMS), the Data Definition Language (DDL) is a set of commands used to define and manage the structure of the database. DDL commands enable users to create, modify, and manage the structure of database objects like tables, indexes, and views. Here are explanations for some key DDL commands:

CREATE:

Purpose: The CREATE command is used to create new database objects, such as tables, indexes, or views.
Example (Table):

```
CREATE TABLE employees (
  emp_id INT PRIMARY KEY,
  emp_name VARCHAR(50),
  emp_salary DECIMAL(10,2)
);
```

This example creates a table named "employees" with columns for employee ID, name, and salary.

ALTER:

Purpose: The ALTER command is used to modify the structure of an existing database object, such as adding, modifying, or dropping columns in a table.
Example (Add Column):

```
ALTER TABLE employees
ADD COLUMN emp_department VARCHAR(30);
```

This example adds a new column named "emp_department" to the existing "employees" table.

TRUNCATE:

Purpose: The TRUNCATE command is used to remove all rows from a table, effectively deleting all data while keeping the table structure intact.
Example:

```
TRUNCATE TABLE employees;
```

This example removes all rows from the "employees" table, leaving an empty table with the same structure.

DROP:

Purpose: The DROP command is used to delete an entire database object, such as a table, index, or view.
Example:

```
DROP TABLE employees;
```

This example deletes the entire "employees" table from the database.
These DDL commands play a crucial role in database management by allowing users to define the schema, modify the structure, clear data, or remove entire objects from the database. It's important to exercise caution when using DDL commands, especially DROP, as they can have a significant impact on the database structure and data. Always ensure that you have a backup and understand the consequences of your actions before executing DDL commands.

Database Language: Data Manipulation Language (DML): INSERT, SELECT, UPDATE, DELETE

Data Manipulation Language (DML) is a part of the Database Language that allows users to manipulate data stored in a database. DML commands primarily include operations for inserting, retrieving, updating, and deleting data within database tables. Here are explanations for key DML commands:

INSERT:

Purpose: The INSERT command is used to add new records (rows) into a table.
Example:

```
INSERT INTO employees (emp_id, emp_name, emp_salary)
VALUES (1, 'John Doe', 50000);
```

This example inserts a new record into the "employees" table with specific values for employee ID, name, and salary.

SELECT:

Purpose: The SELECT command is used to retrieve data from one or more tables. It is the most commonly used DML command.
Example:

```
SELECT emp_id, emp_name, emp_salary
FROM employees
WHERE emp_department = 'IT';
```

This example retrieves data from the "employees" table, selecting only those records where the department is 'IT'.

UPDATE:

Purpose: The UPDATE command is used to modify existing records in a table.
Example:

```
UPDATE employees
SET emp_salary = 55000
WHERE emp_id = 1;
```

This example updates the salary of the employee with ID 1 in the "employees" table.

DELETE:

Purpose: The DELETE command is used to remove records from a table.
Example:

```
DELETE FROM employees
WHERE emp_id = 1;
```

This example deletes the record of the employee with ID 1 from the "employees" table.

These DML commands provide the essential tools for interacting with the data stored in a database. They allow users to insert new data, retrieve specific information, update existing records, and delete unnecessary data. Proper use of DML commands is crucial for maintaining data accuracy, consistency, and integrity within a database. It's important to construct queries and commands carefully to achieve the desired outcomes while avoiding unintended consequences.

Transactional Control: Commit, Save point, Rollback

Transactional control commands are part of Database Management Systems (DBMS) and are used to manage transactions within a database. Transactions are sequences of one or more SQL statements that are executed as a single unit of work. Here are explanations for key transactional control commands:

COMMIT:

Purpose: The COMMIT command is used to permanently save the changes made during the current transaction to the database. Once a COMMIT is issued, the changes become permanent, and other transactions can see the modifications.
Example:

```
BEGIN TRANSACTION;
-- SQL statements
COMMIT;
```

This example starts a transaction, executes some SQL statements, and then commits the changes to make them permanent.

SAVEPOINT:

Purpose: SAVEPOINT is used to set a point within a transaction to which you can later roll back. It allows you to create a marker within the transaction to which you can later return if needed.
Example:

```
SAVEPOINT my_savepoint;
-- SQL statements
ROLLBACK TO my_savepoint;
```

This example sets a savepoint named "my_savepoint" within a transaction and later rolls back to that savepoint, undoing changes made after it was set.

ROLLBACK:

Purpose: The ROLLBACK command is used to undo changes made during the current transaction or to a specified savepoint. It allows reverting the database to its state before the transaction started.
Example:

```
BEGIN TRANSACTION;
-- SQL statements
ROLLBACK;
```

This example starts a transaction, executes some SQL statements, and then rolls back the changes made during the transaction, undoing all modifications.
These transactional control commands are crucial for maintaining the consistency and integrity of a database. They provide a mechanism to ensure that a series of related operations either succeed as a whole (COMMIT) or are fully rolled back to a

consistent state (ROLLBACK or SAVEPOINT). SAVEPOINTs allow for more granular control over parts of a transaction, making it possible to roll back to specific points within a transaction. Understanding and properly using these commands are essential for managing the reliability and robustness of database transactions.

DCL Commands: Grant and Revoke

Data Control Language (DCL) commands in a Database Management System (DBMS) are used to control access to data within the database. Two primary DCL commands are GRANT and REVOKE. Let's explore each:

GRANT:

Purpose: The GRANT command is used to provide specific privileges or permissions to database users or roles. These privileges may include the ability to SELECT, INSERT, UPDATE, DELETE, or execute specific stored procedures.
Syntax:

```
GRANT privilege_name
ON object_name
TO user_name;
```

Example:

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

This example grants the HR user the privileges to select and insert data into the "employees" table.

REVOKE:

Purpose: The REVOKE command is used to revoke previously granted privileges from a user or role. It restricts or removes specific permissions that were previously provided using the GRANT command.
Syntax:

```
REVOKE privilege_name
ON object_name
FROM user_name;
```

Example:

```
REVOKE INSERT
ON employees
FROM hr_user;
```

This example revokes the privilege to insert data into the "employees" table from the HR user.

These DCL commands play a crucial role in ensuring data security and access control within a database. By using GRANT and REVOKE, database administrators can define and manage who can perform specific actions on certain database objects. It's essential to carefully manage and audit these permissions to maintain the confidentiality, integrity, and availability of the data. Access control is a fundamental aspect of database security, and DCL commands are key tools for enforcing it.

Unit – II SQL In built functions and Joins

i. Single row function.
ii. Date functions (add-months, months-between, round, truncate).
iii. Numeric Functions (abs, power, mod, round, trunc, sqrt)
iv. Character Fucntions (initcap, lower, upper, ltrim, rtrim, replace, substring, instr)
v. Conversion Function

Single-Row Functions:
i. Single-Row Function:
Definition: Single-row functions operate on individual rows to return a single result per row. These functions can be used in SELECT, WHERE, and ORDER BY clauses.
Example:

```
SELECT UPPER(employee_name) AS uppercase_name
FROM employees;
```

This example uses the UPPER function to convert the employee_name column values to uppercase for each row.

Date Functions:
ii. Date Functions:

add-months:
Purpose: Adds a specified number of months to a date.
Example:

```
SELECT add_months(hire_date, 3) AS three_months_later
FROM employees;
```

months-between:
Purpose: Calculates the number of months between two dates.
Example:

```
SELECT months_between(SYSDATE, hire_date) AS months_worked
FROM employees;
```

round:
Purpose: Rounds a date to the nearest specified unit (day, month, etc.).
Example:

```
SELECT round(hire_date, 'MONTH') AS rounded_hire_date
FROM employees;
```

truncate:
Purpose: Truncates a date to the specified unit (day, month, etc.).

Example:

```
SELECT truncate(hire_date, 'MONTH') AS truncated_hire_date
FROM employees;
```

Numeric Functions:
iii. Numeric Functions:

abs:
Purpose: Returns the absolute value of a numeric expression.
Example:

```
SELECT abs(salary - 50000) AS salary_difference
FROM employees;
```

power:
Purpose: Raises a number to the power of another.
Example:

```
SELECT power(salary, 2) AS salary_squared
FROM employees;
```

mod:
Purpose: Returns the remainder of a division.
Example:

```
SELECT mod(employee_id, 5) AS id_mod_5
FROM employees;
```

round:
Purpose: Rounds a numeric value to a specified number of decimal places.
Example:

```
SELECT round(salary, 2) AS rounded_salary
FROM employees;
```

trunc:
Purpose: Truncates a numeric value to a specified number of decimal places.
Example:

```
SELECT trunc(salary, 2) AS truncated_salary
FROM employees;
```

sqrt:
Purpose: Returns the square root of a number.
Example:

```
SELECT sqrt(salary) AS salary_sqrt
FROM employees;
```

Character Functions:
iv. Character Functions:

initcap:
Purpose: Converts the first letter of each word to uppercase.
Example:

```
SELECT initcap(employee_name) AS capitalized_name
FROM employees;
```

lower:
Purpose: Converts all letters in a string to lowercase.
Example:

```
SELECT lower(job_title) AS lowercase_title
FROM employees;
```

upper:
Purpose: Converts all letters in a string to uppercase.
Example:

```
SELECT upper(department_name) AS uppercase_department
FROM departments;
```

ltrim:
Purpose: Removes leading spaces from a string.
Example:

```
SELECT ltrim(employee_name) AS trimmed_name
FROM employees;
```

rtrim:
Purpose: Removes trailing spaces from a string.
Example:

```
SELECT rtrim(employee_name) AS trimmed_name
FROM employees;
```

replace:
Purpose: Replaces occurrences of a specified string with another string.
Example:

```
SELECT replace(job_title, 'Manager', 'Lead') AS modified_title
FROM employees;
```

substring:
Purpose: Extracts a substring from a string.
Example:

```
SELECT substring(employee_name, 1, 3) AS sub_name
```

FROM employees;

instr:
Purpose: Returns the position of a substring in a string.
Example:

SELECT instr(job_title, 'Analyst') AS position
FROM employees;

Conversion Functions:
v. Conversion Functions:

to-char:
Purpose: Converts a value of any datatype to a string representation.
Example:

SELECT to_char(hire_date, 'YYYY-MM-DD') AS formatted_date
FROM employees;

to-date:
Purpose: Converts a string or number to a date.
Example:

SELECT to_date('2022-02-14', 'YYYY-MM-DD') AS converted_date
FROM dual;

to-number:
Purpose: Converts a string or date to a number.
Example:

SELECT to_number('123.45') AS converted_number
FROM dual;

These SQL functions offer powerful tools for manipulating data in various ways, including modifying string formats, performing mathematical operations, and handling date and time values. They play a crucial role in crafting complex queries and transforming data to meet specific requirements.

Execute various SQL operators

SQL operators are used to perform operations on data in relational databases. These operators can be categorized into different types based on their functionality. Here are various SQL operators explained:

Arithmetic Operators:

Addition (+):
Adds values together.
Example:

```
SELECT salary + bonus AS total_income
FROM employees;
```

Subtraction (-):
Subtracts one value from another.
Example:

```
SELECT total_sales - expenses AS net_profit
FROM financial_data;
```

Multiplication (*):
Multiplies values.
Example:

```
SELECT quantity * unit_price AS total_cost
FROM orders;
```

Division (/):
Divides one value by another.
Example:

```
SELECT revenue / number_of_customers AS average_revenue
FROM sales_data;
```

Modulus (%):
Returns the remainder of a division.
Example:

```
SELECT employee_id % 2 AS odd_or_even
FROM employees;
```

Comparison Operators:

Equal to (=):
Checks if two values are equal.
Example:

```
SELECT product_name
FROM products
WHERE category = 'Electronics';
```

Not equal to (!= or <>):
Checks if two values are not equal.
Example:

```
SELECT customer_name
FROM customers
WHERE country <> 'USA';
```

Greater than (>):

Checks if one value is greater than another.
Example:

```
SELECT product_name
FROM products
WHERE price > 100;
```

Less than (<):
Checks if one value is less than another.
Example:

```
SELECT employee_name
FROM employees
WHERE years_of_experience < 5;
```

Greater than or equal to (>=):
Checks if one value is greater than or equal to another.
Example:

```
SELECT order_id
FROM orders
WHERE total_amount >= 500;
```

Less than or equal to (<=):
Checks if one value is less than or equal to another.
Example:

```
SELECT customer_name
FROM customers
WHERE age <= 30;
```

Logical Operators:

AND:
Combines conditions and returns true if both conditions are true.
Example:

```
SELECT product_name
FROM products
WHERE category = 'Electronics' AND price > 100;
```

OR:
Combines conditions and returns true if at least one condition is true.
Example:

```
SELECT employee_name
FROM employees
WHERE department = 'Sales' OR department = 'Marketing';
```

NOT:

Negates a condition, returning true if the condition is false, and vice versa.
Example:

```
SELECT customer_name
FROM customers
WHERE NOT country = 'USA';
```

These SQL operators are fundamental for constructing queries that filter, manipulate, and analyze data in relational databases. They enable users to perform a wide range of operations on data to extract meaningful information.

Perform and explain queries on ‘Group by’, ‘Having’ and ‘Order by’ clause

Group BY Clause:
The GROUP BY clause in SQL is used to group rows that have the same values in specified columns into summary rows, like "total sales per category" or "average salary per department."

Syntax:

```
SELECT column1, column2, ..., aggregate_function(column)
FROM table
GROUP BY column1, column2, ...;
```

Example:

```
-- Find the total sales per category
SELECT category, SUM(sales_amount) AS total_sales
FROM sales
GROUP BY category;
```

HAVING Clause:
The HAVING clause is used in combination with the GROUP BY clause to filter the results of a group based on a specified condition. It allows you to filter aggregated data.

Syntax:

```
SELECT column1, column2, ..., aggregate_function(column)
FROM table
GROUP BY column1, column2, ...
HAVING condition;
```

Example:

```
-- Find categories with total sales greater than 1000
SELECT category, SUM(sales_amount) AS total_sales
FROM sales
GROUP BY category
HAVING SUM(sales_amount) > 1000;
```

ORDER BY Clause:
The ORDER BY clause is used to sort the result set of a query in ascending (ASC) or descending (DESC) order based on one or more columns.

Syntax:

```
SELECT column1, column2, ...
FROM table
ORDER BY column1 [ASC | DESC], column2 [ASC | DESC], ...;
```

Example:

```
-- Retrieve employee names and salaries, ordered by salary in descending order
SELECT employee_name, salary
FROM employees
ORDER BY salary DESC;
```

Example Queries:

Group BY:

Find the total number of orders per customer.

```
SELECT customer_id, COUNT(order_id) AS total_orders
FROM orders
GROUP BY customer_id;
```

Having:
Find customers who have made more than 3 orders.

```
SELECT customer_id, COUNT(order_id) AS total_orders
FROM orders
GROUP BY customer_id
HAVING COUNT(order_id) > 3;
```

Order BY:
Retrieve the top 5 highest-paid employees.

```
SELECT employee_name, salary
FROM employees
ORDER BY salary DESC
LIMIT 5;
```

These queries showcase the usage of GROUP BY, HAVING, and ORDER BY clauses in various scenarios. GROUP BY is used for aggregation, HAVING filters the aggregated data, and ORDER BY sorts the final result set.

Joins: Simple, Equi-join, Nonequi, Self-Joins, Outer-joins.

Joins in SQL are used to combine rows from two or more tables based on a related column between them. Here are explanations for different types of joins:

1. Inner Join (Simple Join / Equi-Join):
Description: An inner join returns only the rows that have matching values in both tables. It is the default type of join.
Syntax:

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

Example:

```
SELECT employees.employee_id, employees.employee_name,
departments.department_name
FROM employees
INNER JOIN departments
ON employees.department_id = departments.department_id;
```

2. Non-Equi Join (Nonequi-Join):
Description: A non-equi join is a join condition using operators other than equals.
Syntax:

```
SELECT columns
FROM table1
JOIN table2
ON table1.column < table2.column;
```

Example:

```
SELECT employees.employee_id, employees.employee_name,
departments.department_name
FROM employees
JOIN departments
ON employees.salary > departments.average_salary;
```

3. Self-Join:
Description: A self-join is a regular join, but the table is joined with itself. It is useful when dealing with hierarchical data or relationships within a single table.
Syntax:

```
SELECT columns
FROM table t1
JOIN table t2
```

ON t1.column = t2.column;

Example:

```
SELECT e1.employee_name AS employee, e2.employee_name AS manager
FROM employees e1
JOIN employees e2
ON e1.manager_id = e2.employee_id;
```

4. Outer Joins:
Description: Outer joins return all rows from one table and the matching rows from another table. If no match is found, NULL values are returned for columns from the table without a match.
Syntax (LEFT OUTER JOIN):

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

Syntax (RIGHT OUTER JOIN):

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

Syntax (FULL OUTER JOIN):

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

Example:

```
-- LEFT OUTER JOIN example
SELECT employees.employee_id, employees.employee_name,
departments.department_name
FROM employees
LEFT OUTER JOIN departments
ON employees.department_id = departments.department_id;
```

Joins are essential for combining data from different tables based on related columns. The type of join used depends on the specific requirements of the query and the desired relationship between the tables.

Subqueries - Multiple, Correlated:

Multiple Subqueries:
Description: A multiple subquery is a SQL query that contains more than one subquery. Multiple subqueries can be nested within the WHERE, FROM, or SELECT clauses.

Example:

```
SELECT employee_name
FROM employees
WHERE department_id IN (SELECT department_id FROM departments WHERE location_id = 1)
  AND salary > (SELECT AVG(salary) FROM employees WHERE department_id = 2);
```

Correlated Subqueries:
Description: A correlated subquery is a subquery that references columns from the outer query. The inner query depends on the results of the outer query.

Example:

```
SELECT employee_name, salary
FROM employees e1
WHERE salary > (SELECT AVG(salary) FROM employees e2 WHERE e2.department_id = e1.department_id);
```

Implementation of Queries using SQL Set operators: Union, union all, Intersect, Minus

UNION:
Description: The UNION operator combines the result sets of two or more SELECT statements, removing duplicate rows.

Example:

```
SELECT employee_id, employee_name FROM department1
UNION
SELECT employee_id, employee_name FROM department2;
```

UNION ALL:
Description: The UNION ALL operator combines the result sets of two or more SELECT statements, including duplicate rows.

Example:

```
SELECT employee_id, employee_name FROM department1
UNION ALL
SELECT employee_id, employee_name FROM department2;
```

INTERSECT:
Description: The INTERSECT operator returns the common rows between two SELECT statements, excluding duplicates.

Example:

```
SELECT employee_id, employee_name FROM department1
INTERSECT
SELECT employee_id, employee_name FROM department2;
```

MINUS (or EXCEPT in some databases):
Description: The MINUS operator returns the rows that exist in the first SELECT statement but not in the second SELECT statement.

Example:

```
SELECT employee_id, employee_name FROM department1
MINUS
SELECT employee_id, employee_name FROM department2;
```

These set operators allow you to perform set operations on the result sets of two or more SELECT statements. They are useful for combining, comparing, and analyzing data from different tables or queries. The choice between UNION, UNION ALL, INTERSECT, or MINUS depends on the specific requirements of the query and the desired outcome.

Unit– III Database Integrity Constraints & Objects

Describe with examples Referential Integrity constraint

Referential Integrity Constraint:

Referential Integrity is a relational database concept that ensures relationships between tables are maintained. This is typically achieved through the use of foreign keys. A foreign key in a table refers to the primary key in another table. The Referential Integrity Constraint ensures that relationships between tables remain consistent, preventing actions that would leave orphaned rows or violate the defined relationships.

Example:
Consider two tables: employees and departments.

Create the "departments" table:

```
CREATE TABLE departments (
    department_id INT PRIMARY KEY,
    department_name VARCHAR(50) NOT NULL
);
```

Create the "employees" table with a foreign key:

```
CREATE TABLE employees (
    employee_id INT PRIMARY KEY,
    employee_name VARCHAR(50) NOT NULL,
    department_id INT,
    FOREIGN KEY (department_id) REFERENCES departments(department_id)
);
```

In this example, the employees table has a foreign key (department_id) that references the primary key (department_id) in the departments table.

Insert data into the "departments" table:

```
INSERT INTO departments (department_id, department_name)
VALUES (1, 'HR'), (2, 'IT'), (3, 'Finance');
```

Insert data into the "employees" table:

```
-- This will work because department_id 1 exists in the "departments" table
INSERT INTO employees (employee_id, employee_name, department_id)
VALUES (101, 'John Doe', 1);

-- This will fail because department_id 4 does not exist in the "departments" table
INSERT INTO employees (employee_id, employee_name, department_id)
```

VALUES (102, 'Jane Smith', 4);

The second INSERT statement will result in a Referential Integrity Constraint violation because it attempts to insert a row into the employees table with a department_id (4) that does not exist in the departments table.

Referential Integrity helps maintain the relationships between tables, ensuring that foreign keys in one table correspond to valid primary key values in another table. This prevents data inconsistencies and ensures the integrity of the relational database model.

Views – Create, Alter, Drop views

A view in SQL is a virtual table based on the result of a SELECT query. It is a stored query that can be referenced like a table but does not store the actual data. Views provide a way to simplify complex queries, encapsulate logic, and restrict access to specific columns or rows of a table. Here's how you can create, alter, and drop views:

1. Create a View:
Syntax:

```
CREATE VIEW view_name AS
SELECT column1, column2, ...
FROM table
WHERE condition;
```

Example:

```
CREATE VIEW employee_names AS
SELECT employee_id, employee_name
FROM employees
WHERE department_id = 1;
```

2. Alter a View:
Views in most databases are read-only, so altering a view often involves dropping and recreating it.
Syntax (Drop and Recreate):

```
DROP VIEW view_name;

CREATE VIEW view_name AS
SELECT new_column1, new_column2, ...
FROM new_table
WHERE new_condition;
```

Example:

```
DROP VIEW employee_names;
```

```
CREATE VIEW employee_names AS
SELECT employee_id, employee_name
FROM employees
WHERE department_id = 2;
```

3. Drop a View:
Syntax:

```
DROP VIEW view_name;
```

Example:

```
DROP VIEW employee_names;
```

Additional Points:

Views can be used to encapsulate complex queries and present a simplified interface to users or applications.
Views can join multiple tables, apply filtering conditions, and aggregate data, allowing for more efficient querying.
Views do not store data themselves but provide a dynamic representation of the data stored in the underlying tables.
When you alter a view, you may need to ensure that the underlying tables and columns remain accessible and that the new definition makes sense in the context of the original purpose of the view.

Remember that the ability to alter views depends on the database system being used, and some databases may have specific syntax or restrictions. Always consult the documentation for the specific database you are working with.

Synonym: Create, Drop synonym

Synonym in SQL:

A synonym in SQL is an alias or alternative name for a database object, such as a table, view, sequence, or another synonym. Synonyms provide a way to simplify SQL statements and hide the underlying structure and location of the referenced object. They can be used to make database operations more concise and to facilitate changes to the database schema without affecting application code. Here's how you can create and drop synonyms:

1. Create a Synonym:
Syntax:

```
CREATE [PUBLIC] SYNONYM synonym_name FOR object_name;
```

Example:

```
CREATE SYNONYM emp FOR employees;
```
In this example, the synonym "emp" is created for the "employees" table.

2. Drop a Synonym:
Syntax:

DROP [PUBLIC] SYNONYM synonym_name;

Example:

DROP SYNONYM emp;

This removes the synonym "emp" from the database.

Additional Points:

The optional PUBLIC keyword allows the synonym to be accessible by all users. If omitted, the synonym is private to the schema of the user who creates it.
Synonyms are often used to simplify queries or to provide abstraction layers, allowing for changes to the database structure without affecting application code.
When creating a synonym, ensure that the referenced object exists and is accessible by the user creating the synonym.
Dropping a synonym removes the association between the synonym and the referenced object, but it does not remove the referenced object itself.

Example Use Case:
Suppose you have a complex table name like "employee_data_from_corporate_system," and you want to simplify queries for your application developers:

```
-- Without Synonym
SELECT employee_name, salary
FROM employee_data_from_corporate_system
WHERE department = 'IT';

-- With Synonym
CREATE SYNONYM emp_data FOR employee_data_from_corporate_system;

SELECT employee_name, salary
FROM emp_data
WHERE department = 'IT';
```

In this example, the synonym "emp_data" provides a cleaner and more readable way to reference the underlying table. If the table name changes or if there are multiple tables with similar data, only the synonym needs to be updated, minimizing the impact on application code.

Sequences in SQL:

A sequence in SQL is an object used to generate unique numeric values incrementally. Sequences are often employed to generate primary key values for tables automatically. They provide a way to generate unique identifiers that are typically used in scenarios where a unique, increasing number is required, such as for primary keys in tables. Here's how you can create, alter, and drop sequences:

1. Create a Sequence:
Syntax:

```
CREATE SEQUENCE sequence_name
START WITH start_value
INCREMENT BY increment_value
MINVALUE min_value
MAXVALUE max_value
CYCLE | NO CYCLE;
```

Example:

```
CREATE SEQUENCE emp_id_sequence
START WITH 1
INCREMENT BY 1
MINVALUE 1
MAXVALUE 1000
NO CYCLE;
```

In this example, a sequence named "emp_id_sequence" is created, starting from 1, incrementing by 1, with a minimum value of 1 and a maximum value of 1000. The NO CYCLE option indicates that the sequence will not cycle back to the minimum value after reaching the maximum.

2. Alter a Sequence:
Syntax:

```
ALTER SEQUENCE sequence_name
[INCREMENT BY increment_value]
[MINVALUE min_value]
[MAXVALUE max_value]
[RESTART [WITH restart_value] | NO RESTART];
```

Example:

```
ALTER SEQUENCE emp_id_sequence
INCREMENT BY 2
MAXVALUE 2000
RESTART WITH 100;
```

This alters the "emp_id_sequence" by changing the increment to 2, the maximum value to 2000, and restarting the sequence with a new value of 100.

3. Drop a Sequence:
Syntax:

DROP SEQUENCE sequence_name;

Example:

DROP SEQUENCE emp_id_sequence;

This removes the "emp_id_sequence" from the database.

Additional Points:

Sequences are typically used with the NEXTVAL and CURRVAL functions to retrieve the next sequence value or the current sequence value, respectively.
Sequences are independent of transactions, providing a unique number even if a transaction is rolled back.
Be cautious when altering sequences, especially if they are actively used in the database, as changes may impact existing data.

Example Use Case:

```
-- Create a sequence for employee IDs
CREATE SEQUENCE emp_id_sequence
START WITH 1001
INCREMENT BY 1
MINVALUE 1001
MAXVALUE 9999
NO CYCLE;

-- Use the sequence to insert new employee records
INSERT INTO employees (employee_id, employee_name)
VALUES (emp_id_sequence.NEXTVAL, 'John Doe');

-- Alter the sequence to restart from a higher value
ALTER SEQUENCE emp_id_sequence
RESTART WITH 2000;

-- Drop the sequence when it's no longer needed
DROP SEQUENCE emp_id_sequence;
```

In this example, a sequence is created, used to insert employee records, altered to restart from a higher value, and finally dropped when it's no longer needed. Sequences provide an efficient way to generate unique, incremental values for various purposes in a database.

Indexes in SQL:

An index is a database object that provides a fast and efficient way to look up records in a table based on the values in one or more columns. Indexes improve query performance by allowing the database engine to quickly locate and retrieve data. There are various types of indexes, and two common types are unique indexes and composite indexes.

1. Unique Index:
Description: A unique index ensures that no two rows in a table have the same values in the indexed columns. It enforces the uniqueness constraint.

Syntax (Create):

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

Syntax (Drop):

```
DROP INDEX index_name;
```

Example (Create):

```
CREATE UNIQUE INDEX unique_email_index
ON employees (email);
```

Example (Drop):

```
DROP INDEX unique_email_index;
```

2. Composite Index:
Description: A composite index involves multiple columns and is created on more than one column in a table. It can be beneficial for queries that involve conditions on multiple columns.

Syntax (Create):

```
CREATE INDEX index_name
ON table_name (column1, column2, ...);
```

Syntax (Drop):

```
DROP INDEX index_name;
```

Example (Create):

```
CREATE INDEX composite_index
```

ON employees (department_id, job_title);

Example (Drop):

DROP INDEX composite_index;

Additional Points:

Indexes come with a trade-off: while they significantly speed up query performance, they can slightly slow down data modification operations (such as INSERT, UPDATE, and DELETE), as the indexes need to be updated.
Unique indexes are often used to enforce primary key constraints, ensuring that each row in a table has a unique identifier.
Composite indexes are useful when queries involve conditions on multiple columns, and the order of columns in the index can impact query optimization.

Example Use Case:
Consider a scenario where the "employees" table has a large number of records, and you frequently query based on the department and job title. Creating a composite index on the "department_id" and "job_title" columns can improve the performance of such queries:

```
-- Create a composite index on department_id and job_title
CREATE INDEX composite_index
ON employees (department_id, job_title);

-- Query using the composite index
SELECT employee_name, salary
FROM employees
WHERE department_id = 1 AND job_title = 'Manager';

-- Drop the composite index when it's no longer needed
DROP INDEX composite_index;
```

In this example, the composite index is created to optimize queries that involve conditions on both the "department_id" and "job_title" columns. The index is then dropped when it is no longer needed or if the data distribution changes.