Unit-3 SQL

3.1 Characterstics of SQL (Structured Query Language)

- SQL is an ANSI and ISO standard computer language for creating and manipulating database systems.
- I allows the user to create, update and retreive data from a database.
- Simple and easy to learn.
- It works with database programs like MS Access, DB2, Informix, MS SQL Server, Oracle, Sybase, etc.
- It is a declarative language, not a procedural language.
- It is a case sensitive language.

3.2 Advantages of SQL

- **High speed:-** SQL queries can be used to retrieve large amounts of records from a database quickly and efficiently.
- **No Coding Required:-** SQL is a declarative language, not a procedural language. This means that the user only needs to specify what data is required, not how to get it.
- **Portable:-** SQL is portable. It means that the user can use it on any computer system with the required hardware and software.

3.3 SQL Data Types and Literals

- **char(n)** Fixed length character string, with user-specified length n.
- **varchar(n)** Variable length character strings, with user-specified maximum length n.
- **Boolean** Accepts value true or false.
- **int. Integer (a finite subset of the integers that is machine-dependent).
- **smallint** Small integer (a machine-dependent subset of the integer domain type).
- **decimal(p,d)** Fixed point number, with user-specified precision of p digits, with d digits to the right of decimal point. (ex., decimal(3,1), allows 44.5 to be stored exactly, but not 444.5 or 0.32)
- **Double(p,d)** Floating point and double-precision floating point numbers, with machine- dependent precision. Decimal precision can go to 53 places for a DOUBLE.
- f**loat(p,d)** Floating point number, with user-specified precision of at least n digits. Decimal precision can go to 24 places for a FLOAT.

3.4 SQL Commands

- **Data Definition Language (DDL)** commands are used to create, modify, and remove database objects such as tables, indexes, and users.
- **Data Manipulation Language (DML)** commands are used to insert, update, and delete data from database tables.
- **Data Control Language (DCL)** commands are used to grant and revoke access rights to database users and roles.
- **Transaction Control (TCL)** commands are used to manage the changes made by DML statements.

3.5 Data Definition Language (DDL)

- create tables
- alter tables
- drop tables
- rename tables
- truncate tables

3.6 Integrity constraints

There are three types of `integrity constraints`:-

- Domain Integrity Constraints
- Entity Integrity Constraints
- Refrential Integrity Constraints

3.6.1 Domain Integrity Constriants

- NOT NULL (data value can not be null)
- UNIQUE (data should not be repeated)
- DEFAULT (provides a default value)
- CHECK (ensures that all the values in a column satisfies certain conditions)

3.6.2 Entity Integrit Constraints

- Primary Key: A primary key is a column or a set of columns that uniquely identifies each row in a table.

3.6.3 Refrential Integrity Constraints

- Foreign Key: A foreign key is a column or a set of columns in a table whose values correspond to the values of the primary key in another table.
- Cascade Delete: When a row in the parent table is deleted, the corresponding rows in the child table are automatically deleted.
- Cascade Update: When a row in the parent table is updated, the corresponding rows in the child table are automatically updated.

3.7 Data Manipulation Language (DML)

- insert
- update
- delete
- select

3.8 Data Control Language (DCL)

- grant
- revoke

3.8.1 Grant

- GRANT SELECT ON table_name TO user_name;

- GRANT SELECT, INSERT ON table_name TO user_name;
- GRANT ALL ON table_name TO user_name;
- GRANT SELECT ON table_name TO PUBLIC;
- GRANT SELECT ON table_name TO PUBLIC WITH GRANT OPTION;
- GRANT SELECT ON table_name TO user_name WITH GRANT OPTION;

3.8.2 Revoke

- REVOKE SELECT ON table_name FROM user_name;
- REVOKE SELECT, INSERT ON table_name FROM user_name;
- REVOKE ALL ON table_name FROM user_name;
- REVOKE SELECT ON table_name FROM PUBLIC;
- REVOKE SELECT ON table_name FROM PUBLIC WITH GRANT OPTION;
- REVOKE SELECT ON table_name FROM user_name WITH GRANT OPTION;
- REVOKE ALL PRIVILEGES ON table_name FROM user_name;
- REVOKE ALL PRIVILEGES ON table_name FROM PUBLIC;
- REVOKE ALL PRIVILEGES ON table_name FROM user_name WITH GRANT OPTION;

3.9 Transaction Control (TCL)

- commit
- rollback
- savepoint

3.9.1 Commit

- COMMIT;
- COMMIT WORK;
- COMMIT TRANSACTION;

3.9.2 Rollback

- ROLLBACK;
- ROLLBACK WORK;
- ROLLBACK TRANSACTION;

3.9.3 Savepoint

- SAVEPOINT savepoint_name;
- ROLLBACK TO SAVEPOINT savepoint_name;
- RELEASE SAVEPOINT savepoint_name;

3.10 SQL Operators

- Arithmetic Operators
- Comparison Operators
- Logical Operators

3.10.1 Arithmetic Operators

- Addition (+): Adds two values.
- Subtraction (-): Subtracts one value from another.
- Multiplication (*): Multiplies two values.
- Division (/): Divides one value by another.

3.10.2 Comparison Operators

- Equal to (=): Checks if the values of two operands are equal or not. If yes, then the condition becomes true.
- Not equal to (<>): Checks if the values of two operands are equal or not. If the values are not equal, then the condition becomes true.
- Greater than (>): Checks if the value of the left operand is greater than the value of the right operand. If yes, then the condition becomes true.
- Less than (<): Checks if the value of the left operand is less than the value of the right operand. If yes, then the condition becomes true.
- Greater than or equal to (>=): Checks if the value of the left operand is greater than or equal to the value of the right operand. If yes, then the condition becomes true.
- Less than or equal to (<=): Checks if the value of the left operand is less than or equal to the value of the right operand. If yes, then the condition becomes true.

3.10.3 Logical Operators

- `AND`: If both the operands are true, then the condition becomes true.
- `OR`: If any of the two operands are true, then the condition becomes true.
- `NOT`: Reverses the logical state of its operand. If a condition is true, then the NOT operator will make it false.
- `BETWEEN`: Checks if a value is within a range of values.
- `IN`: Checks if a value is in a list of values.
- `LIKE`: Checks if a value matches a pattern.
- `EXISTS`: Checks if a subquery returns any rows.
- `ALL`: Checks if the condition is true for all values in the column.
- `ANY`: Checks if the condition is true for any value in the column.
- `SOME`: Checks if the condition is true for some values in the column.

3.11 Alter Command

- **ADD** - Adds a column in an existing table.
- **DROP** - Deletes a column from an existing table.
- **MODIFY** - Modifies the data type of a column in an existing table.
- **RENAME** - Renames a column in an existing table.
- **ALTER** - Used to modify the structure of an existing table.
- **DROP** - Deletes an existing table.
- **RENAME** - Renames an existing table.
- **TRUNCATE** - Deletes all records from an existing table.

3.12 SQL Queries

- **SELECT** - extracts data from a database

- **UPDATE** - updates data in a database
- **DELETE** - deletes data from a database
- **INSERT INTO** - inserts new data into a database

The FROM Clause: The FROM clause specifies the tables accessed. The FROM clause is required in every SELECT statement.

The WHERE Clause: The WHERE clause specifies which rows to retrieve. If you omit the WHERE clause, all rows are retrieved.

NULL values: A NULL value is different from zero (0) or an empty string ('').

3.13 SQL Functions

- `Single Row Functions`: Single row functions return a single result row for every row of a queried table or view.
- `Multi Row Functions`: Multi row functions work with more than one row of a table or view at a time.
- `String Functions`:

    1. char_length(): Returns the length of a string in bytes.
    2. concat(): Joins two or more strings together.
    3. lower(): Converts a string to lowercase.
    4. ltrim(): Removes all spaces from the left side of a string.
    5. left(): Extracts a substring from a string (starting from left).
    6. substring(): Extracts a substring from a string (starting from any position).
    7. locate(): Returns the position of a substring within a string.
    8. strcmp(): Compares two strings.
    9. instr(): Returns the position of a substring within a string.

- `Date Function`:

    1. current_date(): Returns the current date.
    2. current_time(): Returns the current time.
    3. current_timestamp(): Returns the current date and time.
    4. extract(): Extracts a part of a date.
    5. date_add(): Adds a specified time interval to a date.
    6. date_sub(): Subtracts a specified time interval from a date.
    7. datediff(): Returns the number of days between two dates.
    8. dayname(): Returns the name of the day of the week.
    9. dayofmonth(): Returns the day of the month.
    10. dayofweek(): Returns the day of the week.
    11. dayofyear(): Returns the day of the year.
    12. hour(): Returns the hour.
    13. minute(): Returns the minute.
    14. month(): Returns the month.
    15. monthname(): Returns the name of the month.
    16. now(): Returns the current date and time.

17. quarter(): Returns the quarter.
18. second(): Returns the second.
19. week(): Returns the week number.
20. year(): Returns the year.

- `Single Row Function`:

    1. avg(): Returns the average value.
    2. count(): Returns the number of rows.
    3. max(): Returns the maximum value.
    4. min(): Returns the minimum value.
    5. sum(): Returns the sum of all or distinct values.

- `Ordering Function`:

    1. asc(): Sorts the result set in ascending order.
    2. desc(): Sorts the result set in descending order.

- `Aggregate Function`:

    1. avg(): Returns the average value.
    2. count(): Returns the number of rows.
    3. max(): Returns the maximum value.
    4. min(): Returns the minimum value.
    5. sum(): Returns the sum of all or distinct values.

3.14 SQL Joins

In SQL, joins are used to combine rows from two or more tables based on a related column between them. Joins allow you to retrieve data from multiple tables by specifying how the tables are related to each other.

Here are the different types of joins commonly used in SQL:

INNER JOIN: Returns only the rows that have matching values in both tables. Example:

```
'''sql
SELECT Orders.order_id, Customers.customer_name
FROM Orders
INNER JOIN Customers ON Orders.customer_id = Customers.customer_id;
'''
```

This query joins the Orders and Customers tables based on the matching customer_id column. It retrieves the order_id from the Orders table and the customer_name from the Customers table for the matching rows.

LEFT JOIN: Returns all the rows from the left table and the matching rows from the right table. If there is no match, NULL values are returned for the right table columns. Example:

```
'''sql
SELECT Customers.customer_name, Orders.order_id
FROM Customers
LEFT JOIN Orders ON Customers.customer_id = Orders.customer_id;
'''
```

This query performs a left join between the Customers and Orders tables. It retrieves the customer_name from the Customers table and the order_id from the Orders table. All rows from the Customers table are returned, and the matching rows from the Orders table are included. If there is no matching row in the Orders table, NULL values are returned for the order_id column.

RIGHT JOIN: Returns all the rows from the right table and the matching rows from the left table. If there is no match, NULL values are returned for the left table columns. Example:

```
'''sql
SELECT Customers.customer_name, Orders.order_id
FROM Customers
RIGHT JOIN Orders ON Customers.customer_id = Orders.customer_id;
'''
```

This query performs a right join between the Customers and Orders tables. It retrieves the customer_name from the Customers table and the order_id from the Orders table. All rows from the Orders table are returned, and the matching rows from the Customers table are included. If there is no matching row in the Customers table, NULL values are returned for the customer_name column.

FULL JOIN: Returns all the rows from both the left and right tables. If there is no match, NULL values are returned for the columns from the non-matching table. Example:

```
'''sql
SELECT Customers.customer_name, Orders.order_id
FROM Customers
FULL JOIN Orders ON Customers.customer_id = Orders.customer_id;
'''
```

This query performs a full join between the Customers and Orders tables. It retrieves the customer_name from the Customers table and the order_id from the Orders table. All rows from both tables are returned, and NULL values are used for the columns where there is no match.

These are the basic types of joins used in SQL. Joins are powerful tools for combining data from multiple tables based on their relationships, allowing you to perform complex queries and retrieve the desired results.

Natural Join: A natural join is a type of join in SQL that combines tables based on columns with the same name and data type. It automatically matches the columns with the same name in both tables and returns

the rows where those values are equal. It eliminates the need to explicitly specify the join condition.

Here's an example to illustrate the concept of a natural join:

Let's consider two tables: Employees and Departments.

Table: Employees '''sql +------+----------+---------+ | emp_id | emp_name | dept_id | +------+----------+---------+ | 1 | John | 101 | | 2 | Jane | 102 | | 3 | Mary | 101 | +------+----------+---------+ ''' Table: Departments

'''sql +---------+------------+ | dept_id | dept_name | +---------+------------+ | 101 | Sales | | 102 | Marketing | +---------+------------+ ''' To perform a natural join between these two tables:

'''sql SELECT * FROM Employees NATURAL JOIN Departments; '''

The result of the query would be:

'''sql +------+----------+---------+------------+ | emp_id | emp_name | dept_id | dept_name | +------+----------+---------+------------+ | 1 | John | 101 | Sales | | 2 | Jane | 102 | Marketing | | 3 | Mary | 101 | Sales | +------+----------+---------+------------+ '''

In this example, the natural join is performed between the Employees and Departments tables. The join operation matches the columns with the same name (dept_id) in both tables. The result is a combination of the matching rows from both tables, where the values in the dept_id column are equal. The resulting table includes all the columns from both tables, eliminating the duplicate column (dept_id) in the result.

It's important to note that natural joins rely solely on matching column names, and if there are additional columns with the same name in both tables, they will also be used for the join, potentially leading to unexpected results. Therefore, it's recommended to exercise caution when using natural joins and consider the structure and naming conventions of the tables involved.

Due to the potential for ambiguity and unexpected behavior, some database professionals prefer to use explicit join syntax with explicitly defined join conditions instead of relying on natural joins.

Cross Join: A cross join, also known as a Cartesian join, is a type of join in SQL that returns the Cartesian product of two tables. It combines every row from the first table with every row from the second table, resulting in a new table with the total number of rows equal to the product of the row counts of the two tables being joined.

The syntax for a cross join is as follows:

'''sql SELECT * FROM Table1 CROSS JOIN Table2; '''

Here's an example to illustrate the concept of a cross join:

Let's consider two tables: Colors and Sizes.

Table: Colors

'''sql +---------+ | Color | +---------+ | Red | | Green | | Blue | +---------+ ''' Table: Sizes

'''sql +---------+ | Size | +---------+ | Small | | Medium | | Large | +---------+ ''' To perform a cross join between these two tables:

'''sql SELECT * FROM Colors CROSS JOIN Sizes; '''

The result of the query would be:

'''sql +---------+---------+ | Color | Size | +---------+---------+ | Red | Small | | Red | Medium | | Red | Large | | Green | Small | | Green | Medium | | Green | Large | | Blue | Small | | Blue | Medium | | Blue | Large | +---------+---------+ '''

In this example, the cross join is performed between the Colors and Sizes tables. Since both tables have three rows, the resulting table contains a total of nine rows, which is the product of 3 (rows in Colors) and 3 (rows in Sizes). Each row from the Colors table is combined with every row from the Sizes table, resulting in all possible combinations.

Cross joins can be useful in certain scenarios, such as when you need to generate all possible combinations between two tables or when you want to create a temporary table for further processing or analysis. However, it's important to exercise caution when using cross joins, as they can result in a large number of rows and can have performance implications if used without proper consideration.

Both left outer join and right outer join are types of outer joins in SQL. They allow you to combine rows from two tables based on a related column, including unmatched rows from one of the tables.

LEFT OUTER JOIN: A left outer join returns all the rows from the left table and the matching rows from the right table. If there is no match, NULL values are returned for the right table columns. Syntax:

'''sql SELECT * FROM Table1 LEFT OUTER JOIN Table2 ON Table1.column = Table2.column; ''' Example: Let's consider two tables: Customers and Orders.

Table: Customers

'''sql +----+----------+ | ID | Customer | +----+----------+ | 1 | John | | 2 | Jane | | 3 | Mary | +----+----------+ ''' Table: Orders

'''sql +----+--------+ | ID | Order | +----+--------+ | 2 | Order1 | | 3 | Order2 | | 4 | Order3 | +----+--------+ ''' To perform a left outer join between these two tables:

'''sql SELECT * FROM Customers LEFT OUTER JOIN Orders ON Customers.ID = Orders.ID; ''' The result of the query would be:

'''sql +----+----------+--------+ | ID | Customer | Order | +----+----------+--------+ | 1 | John | NULL | | 2 | Jane | Order1 | | 3 | Mary | Order2 | +----+----------+--------+ '''

In this example, a left outer join is performed between the Customers and Orders tables. All rows from the Customers table are returned, and the matching rows from the Orders table are included. Since there is no match for the customer with ID 1 in the Orders table, a NULL value is returned for the Order column in that row.

RIGHT OUTER JOIN: A right outer join returns all the rows from the right table and the matching rows from the left table. If there is no match, NULL values are returned for the left table columns. Syntax:

'''sql SELECT * FROM Table1 RIGHT OUTER JOIN Table2 ON Table1.column = Table2.column; '''

Example: Using the same Customers and Orders tables from the previous example, we can perform a right outer join:

'''sql SELECT * FROM Customers RIGHT OUTER JOIN Orders ON Customers.ID = Orders.ID; '''

The result of the query would be:

'''sql +----+----------+--------+ | ID | Customer | Order | +----+----------+--------+ | 2 | Jane | Order1 | | 3 | Mary | Order2 | | 4 | NULL | Order3 | +----+----------+--------+ '''

In this example, a right outer join is performed between the Customers and Orders tables. All rows from the Orders table are returned, and the matching rows from the Customers table are included. Since there is no match for the order with ID 4 in the Customers table, a NULL value is returned for the Customer column in that row.

Left outer join and right outer join allow you to include unmatched rows from one table while combining data from multiple tables. The choice between them depends on which table's unmatched rows you want to include in the result.

An outer join, also known as a full outer join, is a type of join in SQL that combines the results of both the left and right outer joins. It returns all the rows from both tables, including unmatched rows, and fills in NULL values for the columns where there is no match.

The syntax for an outer join varies depending on the specific database system, but here's a general representation:

'''sql SELECT * FROM Table1 FULL OUTER JOIN Table2 ON Table1.column = Table2.column; ''' Example: Let's consider two tables: Customers and Orders.

Table: Customers

'''sql +----+----------+ | ID | Customer | +----+----------+ | 1 | John | | 2 | Jane | | 3 | Mary | +----+----------+ '''

Table: Orders

'''sql +----+--------+ | ID | Order | +----+--------+ | 2 | Order1 | | 3 | Order2 | | 4 | Order3 | +----+--------+ '''

To perform an outer join between these two tables:

'''sql SELECT * FROM Customers FULL OUTER JOIN Orders ON Customers.ID = Orders.ID; '''

The result of the query would be:

'''sql +----+----------+--------+ | ID | Customer | Order | +----+----------+--------+ | 1 | John | NULL | | 2 | Jane | Order1 | | 3 | Mary | Order2 | | 4 | NULL | Order3 | +----+----------+--------+ '''

In this example, an outer join is performed between the Customers and Orders tables. All rows from both tables are returned, and the matching rows are included. Since there is no match for the customer with ID 1 in the Orders table, a NULL value is returned for the Order column in that row. Similarly, since there is no match for the order with ID 4 in the Customers table, a NULL value is returned for the Customer column in that row.

The outer join allows you to retrieve all the rows from both tables, including unmatched rows, and is particularly useful when you want to combine data from multiple tables while keeping all the information intact, even for non-matching records.

Joins on Multiple Tables: When you need to join multiple tables in a SQL query, you can use the JOIN clause multiple times to specify the relationships between the tables. This allows you to retrieve data from multiple tables based on their shared columns. Here's an example to illustrate joining multiple tables:

Consider the following three tables: Orders, Customers, and Products.

Table: Orders

'''sql +----------+------------+----------+ | order_id | customer_id | product_id| +----------+------------+----------+ | 1 | 101 | 201 | | 2 | 102 | 202 | | 3 | 103 | 203 | +----------+------------+----------+ '''

Table: Customers

'''sql +------------+--------------+ | customer_id| customer_name| +------------+--------------+ | 101 | John | | 102 | Jane | | 103 | Mary | +------------+--------------+ ''' Table: Products

'''sql +----------+------------+ | product_id| product_name| +----------+------------+ | 201 | Phone | | 202 | Laptop | | 203 | Tablet | +----------+------------+ '''

To join these tables and retrieve information about orders, customer names, and product names, you can use the following SQL query:

'''sql SELECT Orders.order_id, Customers.customer_name, Products.product_name FROM Orders JOIN Customers ON Orders.customer_id = Customers.customer_id JOIN Products ON Orders.product_id = Products.product_id; '''

The result of the query would be:

'''sql +----------+--------------+--------------+ | order_id | customer_name| product_name | +----------+--------------+--------------+ | 1 | John | Phone | | 2 | Jane | Laptop | | 3 | Mary | Tablet | +----------+--------------+--------------+ '''

In this example, the query joins the Orders, Customers, and Products tables. The first JOIN clause connects the Orders and Customers tables based on the matching customer_id column. Then, the second JOIN clause connects the resulting table with the Products table based on the matching product_id column.

By using multiple JOIN clauses, you can combine data from multiple tables based on their relationships and retrieve the desired information from the joined tables.

Full Outer Join: Apologies for the confusion in my previous response. I made an error in representing the full outer join. However, the full outer join is not supported in all database systems, including MySQL. Instead, you can emulate the functionality of a full outer join using a combination of left outer join and right outer join.

Here's an example of emulating a full outer join between two tables, Table1 and Table2, using a combination of left outer join and right outer join:

'''sql SELECT * FROM Table1 LEFT OUTER JOIN Table2 ON Table1.column = Table2.column UNION SELECT* FROM Table1 RIGHT OUTER JOIN Table2 ON Table1.column = Table2.column WHERE Table1.column IS NULL; '''

In the first part of the query, a left outer join is performed between Table1 and Table2. This retrieves all the rows from Table1 and the matching rows from Table2.

In the second part of the query, a right outer join is performed between Table1 and Table2. This retrieves all the rows from Table2 and the matching rows from Table1.

The UNION operator combines the results of the left outer join and right outer join, effectively emulating the functionality of a full outer join.

The WHERE Table1.column IS NULL condition filters out the rows where the join with Table1 was not successful, ensuring that only the unmatched rows from Table2 are included in the final result.

It's important to note that the above approach assumes that the join condition and column names are consistent between the two tables being joined. Adjustments may be necessary based on the specific column names and join conditions in your scenario.

3.15 Subqueries

Subqueries, also known as nested queries or inner queries, are queries that are embedded within another query. They allow you to retrieve data from one table based on the results of another query. Subqueries can be used in various parts of a SQL statement, such as the SELECT, FROM, WHERE, and HAVING clauses.

Here's an example to illustrate the usage of subqueries:

Consider the following two tables: Orders and Customers.

Table: Orders

+----------+------------+---------+ | order_id | customer_id | amount | +----------+------------+---------+ | 1 | 101 | 100 | | 2 | 102 | 200 | | 3 | 103 | 150 | +----------+------------+---------+ Table: Customers

+------------+--------------+ | customer_id| customer_name| +------------+--------------+ | 101 | John | | 102 | Jane | | 103 | Mary | +------------+--------------+

Subquery in the WHERE clause: You can use a subquery in the WHERE clause to filter the results based on the results of the subquery. For example, to retrieve the orders with an amount greater than the average order amount, you can use the following query:

SELECT * FROM Orders WHERE amount > (SELECT AVG(amount) FROM Orders);

The result of the query would be:

+----------+------------+---------+ | order_id | customer_id | amount | +----------+------------+---------+ | 2 | 102 | 200 | | 3 | 103 | 150 | +----------+------------+---------+

In this example, the subquery (SELECT AVG(amount) FROM Orders) calculates the average order amount, and the main query retrieves the orders with an amount greater than the average.

Subquery in the FROM clause: You can use a subquery in the FROM clause to treat the result of the subquery as a temporary table. For example, to retrieve the customer names along with the total order amount for each customer, you can use the following query:

SELECT c.customer_name, o.total_amount FROM (SELECT customer_id, SUM(amount) AS total_amount FROM Orders GROUP BY customer_id) o JOIN Customers c ON o.customer_id = c.customer_id;

The result of the query would be:

+--------------+--------------+ | customer_name| total_amount | +--------------+--------------+ | John | 100 | | Jane | 200 | | Mary | 150 | +--------------+--------------+

In this example, the subquery (SELECT customer_id, SUM(amount) AS total_amount FROM Orders GROUP BY customer_id) calculates the total order amount for each customer, and the main query joins the subquery results with the Customers table to retrieve the customer names along with their respective total order amounts.

These are just a couple of examples of how subqueries can be used in SQL. Subqueries provide a powerful way to perform complex queries and retrieve data based on conditions derived from other queries. They allow for more dynamic and flexible queries by leveraging the results of intermediate queries within the main query.

3.16 Set Operations

Set operations in SQL allow you to combine and manipulate the result sets of multiple queries. There are three main set operations: UNION, INTERSECT, and EXCEPT (or MINUS). Here's an explanation of each set operation with an example:

UNION: The UNION operation combines the result sets of two or more SELECT statements into a single result set, removing any duplicate rows. Example:

Consider two tables: Customers and Suppliers.

Table: Customers

+----+-----------+ | ID | Customer | +----+-----------+ | 1 | John | | 2 | Jane | | 3 | Mary | +----+-----------+

Table: Suppliers

+----+-----------+ | ID | Supplier | +----+-----------+ | 1 | Apple | | 4 | Samsung | | 5 | Sony | +----+-----------+

To retrieve a combined list of customers and suppliers, you can use the UNION operation:

SELECT Customer AS Name FROM Customers UNION SELECT Supplier AS Name FROM Suppliers;

The result of the query would be:

+-----------+ | Name | +-----------+ | John | | Jane | | Mary | | Apple | | Samsung | | Sony | +-----------+

In this example, the UNION operation combines the result sets of the two SELECT statements, removing any duplicate names, and provides a single result set with the combined list of customers and suppliers.

INTERSECT: The INTERSECT operation returns the common rows that exist in the result sets of two or more SELECT statements. Example:

Consider two tables: Customers and Orders.

Table: Customers

+----+-----------+ | ID | Customer | +----+-----------+ | 1 | John | | 2 | Jane | | 3 | Mary | +----+-----------+
Table: Orders

+----+------------+ | ID | Order | +----+------------+ | 1 | Order1 | | 2 | Order2 | | 4 | Order3 | +----+------------+ To retrieve the customers who have placed orders, you can use the INTERSECT operation:

SELECT Customer FROM Customers INTERSECT SELECT Customer FROM Orders;

The result of the query would be:

+-----------+ | Customer | +-----------+ | John | | Jane | +-----------+

In this example, the INTERSECT operation compares the result sets of the two SELECT statements and returns the common customers who exist in both the Customers and Orders tables.

EXCEPT/ MINUS: The EXCEPT operation (sometimes referred to as MINUS) returns the distinct rows from the first SELECT statement that do not exist in the result set of the second SELECT statement. Example:

Consider two tables: Customers and Orders.

Table: Customers

+----+-----------+ | ID | Customer | +----+-----------+ | 1 | John | | 2 | Jane | | 3 | Mary | +----+-----------+
Table: Orders

+----+------------+ | ID | Order | +----+------------+ | 1 | Order1 | | 2 | Order2 | | 4 | Order3 | +----+------------+

To retrieve the customers who have not placed any orders, you can use the EXCEPT operation:

SELECT Customer FROM Customers EXCEPT SELECT Customer FROM Orders;

The result of the query would be:

+-----------+ | Customer | +-----------+ | Mary | +-----------+

In this example, the EXCEPT operation compares the result sets of the two SELECT statements and returns the customers who exist in the Customers table but do not have a corresponding entry in the Orders table.

These set operations provide flexibility in combining, comparing, and manipulating result sets in SQL, allowing you to perform more complex queries and retrieve the desired information from multiple tables.

3.17 Views

In SQL, a view is a virtual table derived from the result of a query. It acts as a stored query that you can reference and use like a regular table. Views provide several benefits, including simplifying complex queries, improving data security, and enhancing data consistency. Here's an overview of views with an example:

To create a view, you use the CREATE VIEW statement. Here's the general syntax:

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

Here's an example to illustrate the usage of views:

Consider a table called Employees:

Table: Employees

+----+----------+--------+---------+ | ID | Name | Salary | Department | +----+----------+--------+---------+ | 1 | John | 50000 | Sales | | 2 | Jane | 60000 | Marketing | | 3 | Mary | 70000 | Finance | +----+----------+--------+---------+

Suppose you frequently need to retrieve the names and salaries of employees with a salary greater than 55000. Instead of writing the same query repeatedly, you can create a view to simplify the task:

CREATE VIEW HighSalaryEmployees AS SELECT Name, Salary FROM Employees WHERE Salary > 55000;

Now, you can query the HighSalaryEmployees view as if it were a regular table:

'''sql SELECT * FROM HighSalaryEmployees; '''

The result of the query would be:

'''sql +------+--------+ | Name | Salary | +------+--------+ | Jane | 60000 | | Mary | 70000 | +------+--------+ ''' By using the view, you don't need to rewrite the query each time you want to retrieve the high-salary employees. The view encapsulates the query logic and allows you to query the data easily.

Views can also be used for data security and abstraction. For example, you can grant users access to views instead of underlying tables, controlling what data they can see and protecting sensitive information.

Additionally, views can be used to simplify complex queries by breaking them into smaller, more manageable parts. You can create views for commonly used joins, aggregations, or data transformations, and then combine those views in more complex queries.

It's important to note that views are not physical tables. They are stored queries that are dynamically executed when you query them. If the underlying data changes, the view will reflect those changes.

You can modify and drop views using the ALTER VIEW and DROP VIEW statements, respectively.

Views provide a powerful mechanism in SQL for organizing and manipulating data, enhancing query simplicity, security, and data abstraction.

Inserting a Tuple into a View: In SQL, you can insert a tuple into a view under certain conditions. Here are the requirements and considerations for inserting a tuple into a view:

1. The view must be updatable: Not all views are updatable. For a view to be updatable, it must meet certain criteria, such as:
    - The view must be derived from a single base table (not a result of joins, aggregations, etc.).
    - The view must not contain any computed columns.
    - The view must not use the DISTINCT keyword.

- The view must have all the NOT NULL columns of the base table included in the select list.

2. The inserted tuple must satisfy the constraints of the underlying table(s): When inserting a tuple into a view, the inserted data must satisfy any constraints defined on the base table(s) that the view is derived from. For example, if the underlying table has a NOT NULL constraint on a column, the inserted tuple must provide a value for that column.

If the view meets the criteria for updatable views and the inserted tuple satisfies the constraints, you can insert the tuple into the view using the standard INSERT INTO statement.

Here's an example assuming we have an updatable view called "EmployeeNames" derived from the "Employees" table, which includes only the "Name" column:

```
```sql
CREATE VIEW EmployeeNames AS
SELECT Name
FROM Employees;

INSERT INTO EmployeeNames (Name)
VALUES ('Alice');
```
```

In this example, we insert the name 'Alice' into the "EmployeeNames" view, which corresponds to the "Name" column of the "Employees" table. If the "EmployeeNames" view is updatable and the "Employees" table doesn't have any constraints that would be violated, the tuple will be inserted successfully.

It's important to note that when you insert a tuple into a view, the data is actually inserted into the underlying base table(s) that the view is derived from. The view acts as a convenient way to access and modify the data, but the changes are reflected in the base table(s).

However, if the view is not updatable or the inserted tuple violates any constraints, the insertion will fail, and you'll need to modify the view or the underlying table(s) to make it updatable or meet the required constraints.

Update of a View:

In SQL, updating a view involves modifying the underlying data through the view. To update a view, you need to ensure that the view is updatable, meaning it meets certain criteria and conditions. Here's how you can update a view:

1. Check if the view is updatable: Not all views are updatable. To determine if a view is updatable, you can query the `IS_UPDATABLE` column of the `INFORMATION_SCHEMA.VIEWS` system view. For example:

```sql
SELECT IS_UPDATABLE
FROM INFORMATION_SCHEMA.VIEWS
WHERE TABLE_NAME = 'YourViewName' AND TABLE_SCHEMA = 'YourSchemaName';
```

If the `IS_UPDATABLE` value is 'YES', then the view is updatable, and you can proceed with the update operation.

2. Identify the columns to update: Determine which columns of the view you want to modify. Make sure the underlying table(s) allow updates on those columns. If the view includes columns derived from expressions, aggregations, or joins, those columns may not be directly updatable.

3. Use the UPDATE statement: Once you have confirmed that the view is updatable and identified the columns to update, you can use the UPDATE statement to modify the data through the view. The syntax is similar to updating a regular table.

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

Replace `YourViewName` with the name of your view, and set `column1`, `column2`, and so on to the desired column names. Specify the new values with `value1`, `value2`, and so on. The `WHERE` clause is optional and allows you to apply conditions to update specific rows.

Note: When updating a view, the changes are actually performed on the underlying base table(s) that the view is derived from.

Here's an example to illustrate the update of a view:

Suppose we have a view called "HighSalaryEmployees" derived from the "Employees" table, which includes the "Name" and "Salary" columns. Let's update the salary of an employee named "John" through the view:

```
UPDATE HighSalaryEmployees
SET Salary = 55000
WHERE Name = 'John';
```

In this example, we modify the "Salary" column of the "HighSalaryEmployees" view, and the change will be reflected in the underlying "Employees" table.

Remember to ensure that the view is updatable and that the update operation complies with any constraints or rules defined on the underlying base table(s). If the view is not updatable or the update violates any constraints, you may need to modify the view or the underlying table(s) accordingly.

Dropping a View:

In SQL, dropping a view means removing the view definition and deleting it from the database. Dropping a view does not affect the underlying tables or data; it only removes the view itself. Here's how you can drop a view:

```
DROP VIEW view_name;
```

Replace `view_name` with the name of the view you want to drop. Once you execute this statement, the view will be deleted from the database.

Here's an example to illustrate the dropping of a view:

```
DROP VIEW EmployeeNames;
```

In this example, the view named "EmployeeNames" will be dropped, assuming it exists in the database. After executing this statement, the view will no longer be accessible.

It's important to note that dropping a view is a permanent action, and you cannot undo it. Therefore, exercise caution when using the `DROP VIEW` statement, and make sure you specify the correct view name.

Additionally, ensure that you have the necessary permissions to drop a view. Depending on the database system and your user privileges, you may need specific privileges or ownership rights to drop a view.

By dropping a view, you remove the view definition from the database, allowing you to manage your database schema and objects more effectively.

3.18 - SQL Indexes

In databases, an index is a data structure that improves the speed of data retrieval operations on database tables. It is created on one or more columns of a table to facilitate faster searching, sorting, and filtering of data. An index provides a way to access data more efficiently by creating a smaller, optimized version of the table that allows for faster lookup and retrieval of specific data.

Here are a few key points about indexes:

1. Purpose: The primary purpose of an index is to speed up query execution by reducing the number of disk I/O operations required to locate and retrieve data.
2. Structure: An index is typically implemented as a balanced tree data structure, such as a B-tree or a B+ tree. This structure organizes the indexed column(s) in a hierarchical manner, enabling efficient searching and traversal.
3. Columns: You can create an index on one or more columns of a table. When creating an index on multiple columns, the order of the columns can impact the efficiency of certain queries.
4. Performance Impact: While indexes improve read performance by speeding up data retrieval, they can slightly impact write performance as the indexes need to be updated whenever data is inserted, updated, or deleted in the table. Therefore, it's essential to carefully choose the columns to index, considering the balance between read and write operations.
5. Types of Indexes: Different database systems offer various types of indexes, including:
    - B-tree Index: The most common type of index that works well for a wide range of queries.
    - Hash Index: Suitable for exact match searches but not effective for range queries or pattern matching.

    - Bitmap Index: Useful for columns with a limited number of distinct values, such as boolean or categorical data.
    - Full-Text Index: Specifically designed for searching text-based data, allowing efficient searching of words and phrases within large blocks of text.

6. Index Creation: Indexes can be created explicitly by the database administrator or automatically by the database system, based on query patterns and statistics.

7. Index Maintenance: Indexes require regular maintenance to ensure optimal performance. This includes rebuilding or reorganizing indexes to improve their efficiency and updating statistics to help the query optimizer make accurate decisions.

Proper use of indexes can significantly enhance the performance of database queries, especially for tables with a large number of rows. However, it's important to carefully consider the columns to index, monitor index usage and performance, and update indexes when necessary to ensure optimal performance in your specific database environment.

Syntax and Example: The syntax for creating an index in SQL varies slightly depending on the database management system (DBMS) you are using. Here's a general syntax that covers the common aspects:

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

Let's break down the syntax:

- `CREATE INDEX` is the statement used to create an index.
- `index_name` is the name you assign to the index. Choose a descriptive name that reflects its purpose.
- `table_name` is the name of the table on which you want to create the index.
- `(column1, column2, ...)` specifies the column(s) on which the index will be created. You can specify one or more columns, separated by commas. The order of the columns can be significant, as it can impact the index's efficiency for certain queries.

Here's an example to illustrate the usage of the index creation syntax:

Let's say we have a table called `Customers` with columns `CustomerID`, `FirstName`, `LastName`, and `Email`. To create an index on the `LastName` column, you can use the following statement:

```
CREATE INDEX idx_Customers_LastName ON Customers (LastName);
```

In this example, we create an index called `idx_Customers_LastName` on the `LastName` column of the `Customers` table. This index will improve the performance of queries that involve searching, sorting, or filtering by the last name.

It's important to note that the specific syntax and options for index creation may vary depending on the database system you are using. Some DBMSs may provide additional options to customize the index behavior, such as specifying index type, storage options, or collation.

Before creating an index, it's essential to consider the performance implications and analyze the query patterns in your database environment. Not all columns may benefit from an index, and creating unnecessary indexes can introduce overhead in terms of storage and maintenance. Therefore, it's recommended to carefully evaluate the queries that will benefit from the index and choose the appropriate columns to index based on the specific requirements of your application.