Table of Contents

SQL FOR BEGINEERS

What is SQL

SQL stands for Structured Query Language. It is a standard programming language used to manage and manipulate relational databases. SQL allows users to perform tasks such as querying data, updating data, inserting data, deleting data, and creating or modifying database structures like tables, indexes, and views.

DBMS stands for Database Management System.

DBMS stands for Database Management System. It's software that allows users to interact with a database, providing methods for storing, organizing, retrieving, and manipulating data.

Types of DBMS:

1. **Relational DBMS (RDBMS):** Organizes data into tables with rows and columns, with relationships defined between them.
2. **NoSQL DBMS:** Designed for handling unstructured, semi-structured, or structured data, offering flexibility and scalability for modern applications.
3. **Object-Oriented DBMS (OODBMS):** Stores data in objects, integrating database capabilities with object-oriented programming concepts.
4. **Hierarchical DBMS:** Organizes data in a tree-like structure, with parent-child relationships.
5. **Network DBMS**: Similar to hierarchical DBMS but allows multiple parent-child relationships.
6. **Graph DBMS:** Designed for data modeled as graphs, with nodes, edges, and properties representing entities and relationships.

7. **Columnar DBMS:** Stores data in columns rather than rows, optimizing performance for analytical queries.
8. **Document DBMS:** Stores and retrieves semi-structured data in document formats like JSON, XML, BSON, etc.

The Select Statement

The SELECT statement in SQL is used to retrieve data from a database. It allows you to specify which columns you want to retrieve data from and which rows you want to retrieve based on specified conditions.

```
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

- **column1, column2, ...**: The columns you want to retrieve data from.
- **table_name**: The name of the table you want to retrieve data from.
- **condition**: Optional. It specifies the conditions that must be met for the records to be retrieved. If omitted, all rows will be selected.

Example:

Let's assume we have a table named "employees" with columns "employee_id", "first_name", "last_name", and "salary". We want to retrieve the first name and last name of all employees whose salary is greater than 50000.

```
SELECT first_name, last_name
FROM employees
WHERE salary > 50000;
```

In this example:

- **SELECT first_name, last_name**: Specifies that we want to retrieve data from the "first_name" and "last_name" columns.
- **FROM employees**: Specifies that we want to retrieve data from the "employees" table.
- **WHERE salary > 50000**: Specifies the condition that the salary must be greater than 50000 for the records to be retrieved.

The SELECT Clause

The SELECT clause is a fundamental part of the SQL query language. It specifies which columns of data you want to retrieve from a database table or tables.

```
SELECT column1, column2, ...
FROM table_name;
```

You can list one or more columns separated by commas after the SELECT keyword to specify which columns' data you want to retrieve. If you want to retrieve all columns, you can use the asterisk (*) wildcard character.

Example

Example:

Let's say we have a table named "students" with columns "student_id", "first_name", "last_name", "age", and "grade".

To retrieve all columns from the "students" table:

```
SELECT * FROM students;
```

To retrieve only the first name and last name columns from the "students" table:

```
SELECT first_name, last_name FROM students;
```

The WHERE Clause

The WHERE clause in SQL is used to filter rows from a table based on specified conditions. It allows you to narrow down the result set to only those rows that meet the specified criteria.

Example:

Example:

Let's say we have a table named "employees" with columns "employee_id", "first_name", "last_name", and "department_id". We want to retrieve the first name and last name of employees who belong to the 'Marketing' department.

SQL code:

```
SELECT first_name, last_name
 FROM employees
WHERE department_id = 'Marketing';
```

In this example:

- **SELECT first_name, last_name**: Specifies that we want to retrieve data from the "first_name" and "last_name" columns.
- **FROM employees**: Specifies that we want to retrieve data from the "employees" table.
- **WHERE department_id = 'Marketing'**: Specifies the condition that the value in the "department_id" column must be 'Marketing' for the rows to be included in the result set.

The AND, OR and NOT Operations

The AND, OR, and NOT operators in SQL are used to combine multiple conditions in WHERE clauses to filter rows based on logical criteria.

- **AND**: Retrieves rows that satisfy all conditions specified.
- **OR**: Retrieves rows that satisfy at least one of the conditions specified.
- **NOT**: Retrieves rows that do not satisfy the condition specified.

Short Example:

Let's say we have a table named "students" with columns "student_id", "first_name", "last_name", and "age".

1. Using AND:

Sql code

```
SELECT * FROM students
WHERE age >= 18 AND age <= 25;
```

This query retrieves students who are aged between 18 and 25.

2. Using OR:

sql code

SELECT * FROM students

WHERE first_name = 'John' OR first_name = 'Emma';

This query retrieves students with either the first name 'John' or 'Emma'.

3. Using NOT:

sql code

SELECT * FROM students

WHERE NOT age >= 18;

This query retrieves students who are not aged 18 or older.

The IN Operators

The IN operator in SQL is used to specify multiple values in a WHERE clause. It allows you to retrieve rows where a specified column value matches any value in a list.

Example:

Let's say we have a table named "students" with columns "student_id", "first_name", "last_name", and "age".

Sql code

SELECT *

FROM students

WHERE age IN (18, 19, 20);

This query retrieves students whose age is either 18, 19, or 20.

The Between Operators

The BETWEEN operator in SQL is used to specify a range of values in a WHERE clause. It allows you to retrieve rows where a specified column value falls within a specified range, including both endpoints.

Example:

Let's say we have a table named "students" with a column named "age".

Sql code

SELECT *

FROM students

WHERE age BETWEEN 18 AND 25;

This query retrieves students whose age is between 18 and 25, inclusive.

The Like Operator

The LIKE operator in SQL is used to search for a specified pattern in a column. It's commonly used with the WHERE clause to filter rows based on partial matches or patterns.

Example:

Let's say we have a table named "employees" with a column named "last_name".

Sql code

SELECT * FROM employees

WHERE last_name LIKE 'Sm%';

This query retrieves employees whose last names start with "Sm".

- **%** is a wildcard character in SQL that matches zero or more characters.
- **Sm%** specifies that the last name should start with "Sm" followed by any characters.

Point:

- The LIKE operator is useful for pattern matching, allowing for flexible and powerful searches.
- It supports two wildcard characters: **%** (matches zero or more characters) and **_** (matches any single character).

- It's case-insensitive by default in most database systems, but you can specify case sensitivity if needed.

The REGEXP Operator

The REGEXP operator, short for Regular Expression, is used in SQL to perform pattern matching using regular expressions. It allows for more complex and flexible pattern matching than the LIKE operator.

Example:

Let's say we have a table named "products" with a column named "product_name".

Sql code

```
SELECT * FROM products

WHERE product_name REGEXP '^S[0-9]{3}$';
```

This query retrieves products whose names start with 'S' followed by exactly three digits.

- **^** indicates the start of the string.
- **S** is a literal character.
- **[0-9]** matches any digit.
- **{3}** specifies that the preceding character (in this case, **[0-9]**) must appear exactly three times.
- **$** indicates the end of the string.

Point:

- The REGEXP operator provides powerful pattern matching capabilities using regular expressions.
- It allows for more complex and precise pattern matching compared to the LIKE operator.
- Regular expressions can be used to search for patterns such as specific character sequences, ranges, repetitions, and more.

The IS NULL Operator

The IS NULL operator in SQL is used to filter rows where a specified column contains a NULL value.

Example:

Let's say we have a table named "employees" with a column named "email".

Sql code

```
SELECT *
FROM employees
WHERE email IS NULL;
```

This query retrieves employees whose email addresses are NULL.

Point:

The IS NULL operator is used to specifically check for NULL values in a column.

It's particularly useful when you want to find records where certain data is missing.

It's important to note that NULL is not the same as an empty string ('') or a zero value. It represents the absence of a value.

The ORDER BY Clause

The ORDER BY clause in SQL is used to sort the result set of a query based on one or more columns. It arranges the rows returned by a SELECT statement in ascending or descending order.

Example:

Let's say we have a table named "students" with columns "student_id", "first_name", and "last_name".

Sql code

```
SELECT *
FROM students
ORDER BY last_name ASC;
```

This query retrieves all students from the "students" table and sorts them in ascending order based on their last names.

Points:

The ORDER BY clause is used to sort query results.

It can sort the results in either ascending (ASC) or descending (DESC) order, with ASC being the default.

You can sort the results based on one or multiple columns. If multiple columns are specified, the sorting occurs based on the first column, and if there are ties, it proceeds to the next column.

ORDER BY is typically used with SELECT statements but can also be used with other statements like UNION.

Sorting can be done on numeric, string, date, or other data types.

The LIMIT CLAUSE

The LIMIT clause in SQL is used to restrict the number of rows returned by a query. It's particularly useful when dealing with large datasets, allowing you to fetch only a subset of the rows that meet your criteria.

Example:

Let's say we have a table named "students" with columns "student_id", "first_name", and "last_name".

Sql code

SELECT *

FROM students

LIMIT 10;

This query retrieves the first 10 rows from the "students" table.

Points:

- The LIMIT clause is used to limit the number of rows returned by a query.
- It's typically used to improve performance by reducing the amount of data retrieved from the database.
- The LIMIT clause is often combined with the ORDER BY clause to retrieve the top or bottom N rows based on certain criteria.

The syntax may vary slightly depending on the SQL database system. For example, in some databases like PostgreSQL and MySQL, you can specify an optional offset (e.g., LIMIT 10 OFFSET 5) to skip the first N rows before returning the subsequent rows.

Although LIMIT is widely supported, not all SQL databases may support it, or they may have variations in syntax.

INNER JOINS

An INNER JOIN in SQL is used to combine 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.

Example:

Let's say we have two tables: "employees" with columns "employee_id", "first_name", "last_name", and "department_id", and "departments" with columns "department_id" and "department_name". We want to retrieve the names of employees along with their department names.

Sql code

```
SELECT employees.first_name, employees.last_name, departments.department_name
 FROM employees
INNER JOIN departments ON employees.department_id = departments.department_id;
```

In this query:

- **employees** and **departments** are the two tables we're joining.
- **employees.department_id = departments.department_id** specifies the condition for the join. It matches rows in the "employees" table with rows in the "departments" table where the "department_id" values are the same.
- We then select the desired columns from both tables.

Imp Points:

- INNER JOINs are commonly used to retrieve data from multiple related tables.
- They require a matching condition specified in the ON clause to determine which rows from each table should be combined.

- INNER JOINs return only the rows where there is a match in both tables. If there are no matching rows, those rows are not included in the result set.

Joining Across Data Bases

In SQL, joining across databases involves combining data from tables that reside in different databases. This is typically done using fully qualified table names, where you specify both the database and the table name when referencing a table.

Example:

Here's a general example of how you might join tables across databases:

Sql code

```
SELECT t1.column1, t1.column2, t2.column3

FROM database1.table1 AS t1

 INNER JOIN database2.table2 AS t2 ON t1.key_column = t2.key_column;
```

In this example:

- **database1.table1** and **database2.table2** refer to tables in different databases.
- **AS t1** and **AS t2** alias the tables, allowing us to reference them more concisely in the SELECT statement.
- **t1.key_column** and **t2.key_column** specify the column(s) used for joining the tables. These columns must have matching values across the databases.

Points to note:

- To perform joins across databases, the databases must be accessible from the same database management system (DBMS).
- Depending on the DBMS you're using, you might need appropriate permissions to access tables in different databases.
- Be mindful of performance implications when joining across databases, especially if the databases reside on different servers.
- It's essential to ensure data consistency and integrity, as joining across databases can introduce complexity and potential issues.

SELF JOIN

A self join in SQL is when a table is joined with itself. This can be useful when you have a table with hierarchical or recursive data, or when you need to compare rows within the same table.

Example:

Let's say we have a table named "employees" with columns "employee_id", "first_name", "last_name", and "manager_id". The "manager_id" column contains the ID of the manager for each employee, where the manager is also an employee in the same table.

To retrieve the names of employees along with the names of their managers:

Sql code

```
SELECT e1.first_name AS employee_first_name, e1.last_name AS employee_last_name,
        e2.first_name AS manager_first_name, e2.last_name AS manager_last_name
        FROM employees e1
        INNER JOIN employees e2 ON e1.manager_id = e2.employee_id;
```

In this query:

- We use aliases (e1 and e2) to distinguish between the two instances of the "employees" table.
- We join the table with itself based on the relationship between an employee and their manager (where the manager_id in one row matches the employee_id in another).
- By joining the table with itself, we can retrieve the names of employees along with the names of their managers.

Points to note:

- Self joins can be useful for hierarchical or recursive data structures, such as organizational charts.
- Be cautious when using self joins, as they can lead to complex queries and potentially degrade performance if not used wisely.
- Always ensure that you have appropriate indexing and data structure optimizations in place when working with self joins on large datasets

Joining Multiple Tables

Joining multiple tables in SQL allows you to combine data from more than two tables based on related columns. This is commonly done using various types of JOINs such as INNER JOIN, LEFT JOIN, RIGHT JOIN, or FULL JOIN.

Example:

Here's a general example of how you might join three tables:

Sql code

```
SELECT t1.column1, t2.column2, t3.column3
FROM table1 AS t1
```

INNER JOIN table2 AS t2 ON t1.key_column = t2.key_column

INNER JOIN table3 AS t3 ON t2.another_key_column = t3.another_key_column;

In this example:

- **table1**, **table2**, and **table3** are the three tables being joined.
- **AS t1**, **AS t2**, and **AS t3** alias the tables for easier reference.
- We use INNER JOINs to match rows between the tables based on specified key columns.

You can also use other types of JOINs as needed, such as LEFT JOIN, RIGHT JOIN, or FULL JOIN, depending on your requirements for including or excluding unmatched rows from the joined tables.

Example with LEFT JOIN:

Sql code

SELECT t1.column1, t2.column2, t3.column3

FROM table1 AS t1

LEFT JOIN table2 AS t2 ON t1.key_column = t2.key_column

LEFT JOIN table3 AS t3 ON t2.another_key_column = t3.another_key_column;

This query retrieves all rows from **table1** and includes matching rows from **table2** and **table3**, if any.

Points to note:

- When joining multiple tables, ensure that you specify the join conditions correctly to avoid unintended Cartesian products or data duplication.
- Understand the relationships between the tables and choose the appropriate type of JOIN (INNER, LEFT, RIGHT, FULL) based on your requirements.
- Use table aliases to improve the readability of your SQL queries, especially when working with multiple tables.

Compound Join Conditions

Compound join conditions in SQL are used when you need to specify more than one condition for joining tables. These conditions can involve multiple columns or expressions and are combined using logical operators such as AND or OR.

Example:

Let's say we have two tables: "orders" and "customers". Each order has a customer associated with it, and we want to retrieve orders along with the corresponding customer information. However, we want to ensure that we match orders with customers based on both the customer ID and the country to which the order is shipped.

Sql code

```
SELECT o.order_id, o.order_date, c.customer_name, c.country
FROM orders o
INNER JOIN customers c
    ON o.customer_id = c.customer_id
    AND o.shipping_country = c.country;
```

In this example:

- We're using an INNER JOIN to combine the "orders" and "customers" tables.
- The ON clause specifies the join conditions. In this case, we have a compound join condition involving two conditions: **o.customer_id = c.customer_id** and **o.shipping_country = c.country**.
- This ensures that we only retrieve orders where both the customer ID matches and the shipping country matches the country of the customer.

Points to note:

- Compound join conditions allow for more precise matching between tables based on multiple criteria.
- They can involve multiple columns or expressions connected with logical operators like AND or OR.
- Be cautious when using compound join conditions to avoid overly complex queries that might be difficult to understand or maintain.

Implicit and explicit Join Syntax

Implicit join syntax refers to the older method of joining tables in SQL, where you use comma-separated table names in the FROM clause and specify join conditions in the WHERE clause. This method is considered less readable and less flexible compared to explicit join syntax, where you use JOIN and ON keywords to specify join conditions.

Example using implicit join syntax:

```
SELECT orders.order_id, customers.customer_name
FROM orders, customers
WHERE orders.customer_id = customers.customer_id;
```

In this example:

- We're selecting data from the "orders" and "customers" tables.
- We list both tables in the FROM clause separated by a comma.
- The join condition, **orders.customer_id = customers.customer_id**, is specified in the WHERE clause.

While this syntax is still valid in SQL, it's generally recommended to use explicit join syntax for better readability and maintainability of your SQL queries.

Example using explicit join syntax:

```
SELECT orders.order_id, customers.customer_name
FROM orders
JOIN customers ON orders.customer_id = customers.customer_id;
```

In this example, the JOIN and ON keywords make the join condition more explicit and easier to understand.

Points to note:

- Implicit join syntax is older and less preferred in modern SQL development.
- Explicit join syntax using the JOIN and ON keywords is generally considered more readable and clearer.
- Explicit join syntax also allows for different types of joins (INNER JOIN, LEFT JOIN, etc.) and makes it easier to add additional join conditions.

Outer Joins(LEFT and RIGHT)

Outer joins, specifically LEFT JOIN and RIGHT JOIN, are used in SQL to retrieve rows from one table even if there is no matching row in the other table being joined. This means that all rows from one specified table are returned, along with matching rows from the other table. If there is no match, NULL values are returned for the columns from the other table.

Here's an explanation of LEFT JOIN and RIGHT JOIN:

1. **LEFT JOIN**: Returns all rows from the left table (the first table specified in the JOIN clause), along with matching rows from the right table (the second table specified in the JOIN clause). If there is no matching row in the right table, NULL values are returned for the columns from the right table.

Example:

sql code

```
SELECT * FROM table1
 LEFT JOIN table2 ON table1.key_column = table2.key_column;
```

2. **RIGHT JOIN**: Returns all rows from the right table (the second table specified in the JOIN clause), along with matching rows from the left table (the first table specified in the JOIN clause). If there is no matching row in the left table, NULL values are returned for the columns from the left table.

Example:

Sql code

```
SELECT *
 FROM table1
 RIGHT JOIN table2 ON table1.key_column = table2.key_column;
```

In both cases, the JOIN condition specifies how the two tables are related, and the result set includes rows from both tables based on that relationship. However, the difference lies in which table's rows are guaranteed to be included in the result set: with LEFT JOIN, it's the left table, and with RIGHT JOIN, it's the right table.

Points to note:

- Outer joins are useful when you want to retrieve all records from one table, regardless of whether there is a matching record in the other table.
- LEFT JOIN and RIGHT JOIN can be used to handle situations where you need to include data from one table that might not have corresponding data in the other table.
- LEFT JOIN and RIGHT JOIN are complementary, meaning that you can often achieve the same result by swapping the order of the tables and using the other type of join.

Outer Join Between Multiple Tables

Performing an outer join involving multiple tables in SQL is similar to joining two tables, but with additional JOIN clauses for each additional table. You can use LEFT JOIN, RIGHT JOIN, or FULL OUTER JOIN to include all rows from one or more specified tables, along with matching rows from other tables.

Here's an example of an outer join involving three tables:

Sql code

```
SELECT t1.column1, t2.column2, t3.column3

FROM table1 AS t1

 LEFT JOIN table2 AS t2 ON t1.key_column = t2.key_column

LEFT JOIN table3 AS t3 ON t1.key_column = t3.key_column;
```

In this example:

- We're performing a LEFT JOIN between **table1** and **table2**, and then another LEFT JOIN between **table1** and **table3**.
- This query retrieves all rows from **table1**, along with matching rows from **table2** and **table3** based on the specified key column(s).
- If there's no match in **table2** or **table3**, NULL values are returned for the corresponding columns.

Similarly, you can perform a RIGHT JOIN or FULL OUTER JOIN if needed, depending on your requirements for including or excluding unmatched rows from the joined tables.

Points to note:

- When performing outer joins involving multiple tables, ensure that you specify the join conditions correctly to avoid unintended data duplication or exclusion.
- Use aliases for table names (**AS t1**, **AS t2**, etc.) to improve the readability of your SQL queries, especially when working with multiple tables.
- Understand the relationships between the tables and choose the appropriate type of outer join based on your requirements for including or excluding unmatched rows.

SELF OUTER JOINS

Performing a self outer join, also known as a self join, involves joining a table to itself. This can be useful when you want to compare rows within the same table or create hierarchical structures. However, SQL doesn't directly support self outer joins like it does with inner and outer joins. Instead, you typically use a combination of subqueries or common table expressions (CTEs) to achieve the desired result.

Here's an **example** of how you might perform a self outer join using a UNION ALL with a subquery:

Suppose we have a table named "employees" with columns "employee_id", "first_name", and "manager_id". We want to retrieve each employee along with their manager's name:

Sql code

```
SELECT e1.first_name AS employee_name, e2.first_name AS manager_name

FROM employees e1
```

```
LEFT JOIN employees e2 ON e1.manager_id = e2.employee_id

 UNION ALL

SELECT e1.first_name AS employee_name, 'No Manager' AS manager_name

FROM employees e1 WHERE e1.manager_id IS NULL;
```

In this example:

- The first part of the query performs a LEFT JOIN between the "employees" table aliased as "e1" and itself aliased as "e2" based on the manager_id.
- The second part of the query retrieves employees who do not have a manager (i.e., where manager_id is NULL).
- UNION ALL combines the results of both queries, ensuring that all employees, including those without managers, are included in the final result set.

Points to note:

- Self outer joins require careful consideration to avoid circular references or unintended results.
- Using subqueries or CTEs is a common approach to perform self outer joins in SQL.
- Ensure that your SQL database system supports the features used in your query, such as subqueries and UNION ALL.

The USING Clause

The USING clause in SQL is used in conjunction with JOIN operations to specify the columns used for joining tables when those columns have the same name in both tables. It provides a shorthand method for joining tables based on common column names.

Here's a basic syntax of using the USING clause with INNER JOIN:

Sql code

```
SELECT *

FROM table1

INNER JOIN table2 USING (common_column);
```

In this syntax:

table1 and table2 are the tables being joined.

common_column is the name of the column that exists in both tables and is used for joining.

Example:

Let's say we have two tables, "employees" and "departments", where both tables have a column named "department_id". We want to join these tables based on the "department_id" column:

Sql code

SELECT *

FROM employees

INNER JOIN departments USING (department_id);

In this query, the USING clause specifies that the join should be performed based on the "department_id" column, which exists in both the "employees" and "departments" tables.

Points to note:

The columns specified in the USING clause must have the same name in both tables.

The result of the join is the same as using the ON clause to specify the join condition explicitly, but the syntax is more concise when the column names match.

The USING clause only works with INNER JOINs and cannot be used with other types of joins such as LEFT JOIN or RIGHT JOIN.

Natural JOINS

A NATURAL JOIN in SQL is a type of JOIN operation that automatically joins tables based on columns with the same name. It eliminates the need to specify join conditions explicitly, as it automatically matches columns with identical names in both tables.

Example:

Suppose we have two tables, "employees" and "departments", both containing a column named "department_id". To perform a natural join between these tables:

Sql code

SELECT *

FROM employees

NATURAL JOIN departments;

In this example, the NATURAL JOIN operation automatically matches rows from the "employees" table with rows from the "departments" table where the "department_id" column values are equal.

Points to note:

NATURAL JOIN simplifies joining tables by automatically matching columns with the same name.

It's concise and convenient but may lead to unexpected results if columns with the same name exist but should not be used for joining.

Due to the potential for ambiguity, many SQL developers prefer to use explicit JOIN operations with ON or USING clauses instead of NATURAL JOIN.

CROS JOINS

A CROSS JOIN in SQL is a type of join operation that produces the Cartesian product of the two tables being joined. It combines every row from the first table with every row from the second table, resulting in a potentially large result set.

Example:

Suppose we have two tables, "employees" and "departments", and we want to combine every employee with every department:

Sql code

SELECT *

FROM employees

CROSS JOIN departments;

In this example, the CROSS JOIN operation combines every row from the "employees" table with every row from the "departments" table, resulting in a result set where each employee is paired with each department.

Points to note:

CROSS JOINs can produce a large result set, especially if the tables being joined are large.

They are often used when you need to generate combinations of data from different tables.

Be cautious when using CROSS JOINs, as they can easily result in performance issues and unintended data combinations.

Unlike other join types, CROSS JOINs do not require a join condition.

Unions

UNION in SQL is used to combine the results of two or more SELECT queries into a single result set. It removes duplicate rows from the combined result set by default.

Here's the basic syntax of using UNION:

Sql code

SELECT column1, column2, ...

FROM table1

UNION

SELECT column1, column2, ...

FROM table2;

In this syntax:

- Each SELECT query must have the same number of columns, and the data types of corresponding columns must be compatible.
- The result set includes all rows from both SELECT queries, excluding duplicates.
- If you want to include duplicates, you can use UNION ALL instead of UNION.

Example:

Suppose we have two tables, "students2019" and "students2020", both containing columns "student_id", "first_name", and "last_name". We want to combine the data from both tables into a single result set:

Sql code

```
SELECT student_id, first_name, last_name

FROM students2019

UNION

SELECT student_id, first_name, last_name

FROM students2020;
```

In this example, the UNION operation combines the data from "students2019" and "students2020" into a single result set, removing any duplicate rows.

Points to note:

- The columns in the SELECT queries must have the same data types and be listed in the same order.
- UNION removes duplicate rows by default. If you want to include duplicates, use UNION ALL.
- UNION can be used to combine the results of multiple SELECT queries, even if they are from different tables or have different conditions.

Column Attributes

Column attributes in SQL refer to the characteristics or properties of a column in a database table. These attributes define various aspects of the column, such as data type, size, constraints, and default values. Here are some common column attributes:

1. **Data Type:** Specifies the type of data that can be stored in the column, such as INTEGER, VARCHAR, DATE, etc.
2. **Size:** For character-based data types (e.g., VARCHAR), the size attribute specifies the maximum number of characters that can be stored in the column.
3. **Constraints:** Constraints enforce rules or conditions on the data stored in the column. Common constraints include:
    - NOT NULL: Ensures that the column cannot contain NULL values.
    - UNIQUE: Ensures that all values in the column are unique.
    - PRIMARY KEY: Combines the NOT NULL and UNIQUE constraints and identifies the column as the primary key for the table.

- FOREIGN KEY: Establishes a relationship between this column and the primary key column in another table.
- CHECK: Specifies a condition that must be satisfied for the data in the column.

4. **Default Value:** Specifies a default value that is assigned to the column if no value is explicitly provided during insertion.
5. **Identity/Sequence:** Some databases support an identity or sequence attribute for columns, which automatically generates unique values for the column.

Example of defining column attributes in a CREATE TABLE statement:

Sql code

```
CREATE TABLE employees (
   employee_id INT PRIMARY KEY,
   first_name VARCHAR(50) NOT NULL,
   last_name VARCHAR(50) NOT NULL,
   department_id INT,
   hire_date DATE DEFAULT CURRENT_DATE,
   CONSTRAINT fk_department FOREIGN KEY (department_id) REFERENCES
departments(department_id)
);
```

In this example:

- **employee_id** is an INTEGER data type with a PRIMARY KEY constraint.
- **first_name** and **last_name** are VARCHAR data types with a size of 50 characters and NOT NULL constraint.
- **department_id** is an INTEGER data type.
- **hire_date** is a DATE data type with a DEFAULT constraint set to the current date.
- A FOREIGN KEY constraint (**fk_department**) is defined on the **department_id** column, referencing the **department_id** column in the **departments** table.

Inserting into single row

To insert a single row into a table in SQL, you use the INSERT INTO statement. Here's the basic syntax:

Sql code

```
INSERT INTO table_name (column1, column2, ...)
```

VALUES (value1, value2, ...);

In this syntax:

- **table_name** is the name of the table into which you want to insert the data.
- **(column1, column2, ...)** specifies the columns into which you want to insert data. If you omit the column list, you need to provide values for all columns in the same order they are defined in the table.
- **VALUES (value1, value2, ...)** specifies the values to be inserted into the corresponding columns. Each value must match the data type of the corresponding column.

Example:

Suppose we have a table named "employees" with columns "employee_id", "first_name", "last_name", and "hire_date". We want to insert a new employee into this table:

Sql code

INSERT INTO employees (employee_id, first_name, last_name, hire_date)

VALUES (1, 'John', 'Doe', '2022-03-29');

In this example:

- We're inserting a new row into the "employees" table with the employee's information: employee_id = 1, first_name = 'John', last_name = 'Doe', and hire_date = '2022-03-29'.

Points to note:

- Make sure the values you're inserting adhere to any constraints defined on the table columns (e.g., NOT NULL constraints, UNIQUE constraints).
- Ensure that the values provided match the data types of the corresponding columns. If not, you may encounter errors during insertion.
- It's good practice to specify the column list explicitly in the INSERT statement to avoid unexpected behavior if the table structure changes in the future.

INSERTING Into Multiple Rows

To insert multiple rows into a table in SQL, you can use the INSERT INTO statement with a single VALUES clause containing multiple value sets, or you can use the INSERT INTO statement with a SELECT statement that retrieves the data you want to insert.

Here are both methods:

Using a single INSERT INTO statement with multiple value sets:

sql code

```
INSERT INTO table_name (column1, column2, ...)
VALUES
 (value1_1, value1_2, ...),
(value2_1, value2_2, ...),
 ...;
```

In this syntax:

- **table_name** is the name of the table into which you want to insert the data.
- **(column1, column2, ...)** specifies the columns into which you want to insert data.
- **VALUES** is followed by multiple sets of values enclosed in parentheses, separated by commas.

Example:

sql code

```
INSERT INTO employees (employee_id, first_name, last_name)
VALUES (1, 'John', 'Doe'),
        (2, 'Jane', 'Smith'),
    (3, 'Alice', 'Johnson');
```

Using a single INSERT INTO statement with a SELECT statement:

Sql code

```
INSERT INTO table_name (column1, column2, ...)
SELECT value1_1, value1_2, ...
UNION ALL
SELECT value2_1, value2_2, ...
UNION ALL
 ...;
```

In this syntax:

- **table_name** is the name of the table into which you want to insert the data.
- **(column1, column2, ...)** specifies the columns into which you want to insert data.
- The SELECT statement retrieves the data you want to insert.

Example:

Sql code

```
INSERT INTO employees (employee_id, first_name, last_name)
 SELECT 1, 'John', 'Doe'
UNION ALL
SELECT 2, 'Jane', 'Smith'
 UNION ALL
SELECT 3, 'Alice', 'Johnson';
```

Points to note:

- Ensure that the number of columns and their data types in the INSERT INTO statement match the number and types of columns in the target table.
- When using the SELECT statement method, ensure that the SELECT statement retrieves the correct data for insertion into the table.
- Both methods can be used to insert multiple rows efficiently into a table with a single SQL statement.

INSERTING Hierarchical Rows

Inserting hierarchical rows in SQL involves inserting data into a table that has a hierarchical relationship, such as a parent-child relationship. Typically, this is achieved by ensuring that the parent rows exist before inserting the child rows and maintaining referential integrity using foreign key constraints.

Here's a basic approach to inserting hierarchical rows:

1. Insert parent rows first.
2. Insert child rows, referencing the parent rows using foreign keys.

Let's illustrate this with an example of inserting hierarchical rows into tables representing employees and departments:

Assuming we have two tables:

1. "departments" table with columns:
    - department_id (Primary Key)
    - department_name
2. "employees" table with columns:
    - employee_id (Primary Key)
    - first_name

- last_name
- department_id (Foreign Key referencing department_id in the departments table)

We want to insert departments and employees hierarchically.

First, we insert departments:

Sql code

```
INSERT INTO departments (department_id, department_name)
VALUES
    (1, 'Engineering'),
    (2, 'Marketing');
```

Next, we insert employees, ensuring that each employee belongs to an existing department:

Sql code

```
INSERT INTO employees (employee_id, first_name, last_name, department_id)
VALUES
  (1, 'John', 'Doe', 1),  -- Employee John Doe belongs to Engineering department
  (2, 'Jane', 'Smith', 1), -- Employee Jane Smith belongs to Engineering department
  (3, 'Alice', 'Johnson', 2); -- Employee Alice Johnson belongs to Marketing department
```

In this example:

- We insert departments first to ensure that the department_id values exist when inserting employees.
- When inserting employees, we specify the department_id value corresponding to the department to which each employee belongs.

Points to note:

- Hierarchical data insertion ensures that referential integrity is maintained.
- You must ensure that parent rows (departments) exist before inserting child rows (employees) to avoid referential integrity violations.
- Foreign key constraints help enforce relationships between parent and child rows.

Creating a Copy of Table

To create a copy of a table in SQL, you can use the CREATE TABLE statement along with a SELECT query that retrieves all the data from the original table. This approach allows you to copy both the structure and data of the original table into the new table.

Here's a basic example:

Sql code

```
CREATE TABLE new_table AS
 SELECT *
 FROM original_table;
```

In this example:

- **new_table** is the name of the new table you want to create.
- **original_table** is the name of the table you want to copy.

This statement creates a new table (**new_table**) with the same structure and data as the original table (**original_table**). It copies all columns and rows from the original table into the new table.

If you want to copy only the structure of the original table without any data, you can use a WHERE clause that always evaluates to false in the SELECT query, ensuring that no rows are returned:

Sql code

```
CREATE TABLE new_table AS
SELECT *
FROM original_table
WHERE 1 = 0;
```

Points to note:

- Make sure that the new table name (**new_table**) does not already exist in the database to avoid conflicts.
- Ensure that the SELECT query retrieves the desired columns and rows from the original table.
- Column names, data types, constraints, and indexes are copied from the original table to the new table.

Updating a Single Row

To update a single row in a table in SQL, you use the UPDATE statement. Here's the basic syntax:

sql code

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

In this syntax:

- **table_name** is the name of the table you want to update.
- **column1**, **column2**, etc., are the columns you want to update.
- **value1**, **value2**, etc., are the new values you want to assign to the corresponding columns.
- **condition** specifies the row(s) that you want to update. If omitted, all rows in the table will be updated.

Example:

Suppose we have a table named "employees" with columns "employee_id", "first_name", and "last_name". We want to update the last name of the employee with ID 1 to "Johnson":

Sql code

```
UPDATE employees
 SET last_name = 'Johnson'
 WHERE employee_id = 1;
```

In this example:

- We use the UPDATE statement to modify the data in the "employees" table.
- We specify that we want to update the "last_name" column to the value 'Johnson'.
- We use the WHERE clause to specify the condition that restricts the update to the row where the "employee_id" is 1.

Points to note:

- Make sure to include a WHERE clause to target the specific row(s) you want to update. Without it, all rows in the table will be updated.
- Ensure that the condition in the WHERE clause accurately identifies the row(s) you want to modify.
- The UPDATE statement can modify multiple columns in a single row simultaneously.

Updating Multiple Rows

To update multiple rows in a table in SQL, you can use the UPDATE statement with a WHERE clause that specifies the condition for selecting the rows to be updated. Here's the basic syntax:

Sql code

```
UPDATE table_name

SET column1 = value1, column2 = value2, ...

WHERE condition;
```

In this syntax:

- **table_name** is the name of the table you want to update.
- **column1**, **column2**, etc., are the columns you want to update.
- **value1**, **value2**, etc., are the new values you want to assign to the corresponding columns.
- **condition** specifies the rows that you want to update. If omitted, all rows in the table will be updated.

Example:

Suppose we have a table named "employees" with columns "department_id" and "salary". We want to give a salary raise of 10% to all employees in the Engineering department (department_id = 1):

Sql code

```
UPDATE employees

SET salary = salary * 1.1

 WHERE department_id = 1;
```

In this example:

- We use the UPDATE statement to modify the data in the "employees" table.
- We specify that we want to update the "salary" column by multiplying the current salary by 1.1 to give a 10% raise.
- We use the WHERE clause to specify the condition that restricts the update to the rows where the "department_id" is 1 (i.e., Engineering department).

Points to note:

- Ensure that the condition in the WHERE clause accurately identifies the rows you want to modify.
- The UPDATE statement can modify multiple columns in a single row simultaneously.
- Always double-check the WHERE clause to avoid unintended updates to rows that you did not intend to modify.

USING Subqueries in Update

Using subqueries in an UPDATE statement allows you to dynamically determine the value to update based on the result of a subquery. Here's the basic syntax:

Sql code

```
UPDATE table_name

SET column1 = (SELECT subquery)

WHERE condition;
```

In this syntax:

- **table_name** is the name of the table you want to update.
- **column1** is the column you want to update.
- **subquery** is a SELECT statement that retrieves the value(s) you want to use for updating the column.
- **condition** specifies the rows that you want to update. If omitted, all rows in the table will be updated.

Example:

Suppose we have a table named "orders" with columns "order_id" and "total_amount", and we want to update the total amount of each order to be 10% higher than the average total amount of all orders:

Sql code

```
UPDATE orders

SET total_amount = total_amount * 1.1

WHERE order_id IN (

  SELECT order_id

  FROM orders

  WHERE total_amount > (SELECT AVG(total_amount) FROM orders)

);
```

In this example:

- We use a subquery **(SELECT AVG(total_amount) FROM orders)** to calculate the average total amount of all orders.
- We use another subquery **(SELECT order_id FROM orders WHERE total_amount > (SELECT AVG(total_amount) FROM orders))** to select the order IDs of orders with a total amount greater than the average.

- We update the total amount of each selected order by multiplying it by 1.1 (10% increase).

Points to note:

- Subqueries in UPDATE statements should return a single value or a single column.
- Ensure that the subquery returns the correct value(s) that you want to use for updating the column.
- The WHERE clause in the UPDATE statement restricts the update to the rows that meet certain conditions, if necessary.

Deleting Rows

To delete rows from a table in SQL, you use the DELETE statement. Here's the basic syntax:

Sql code

```
DELETE FROM table_name

WHERE condition;
```

In this syntax:

- **table_name** is the name of the table from which you want to delete rows.
- **condition** specifies the rows that you want to delete. If omitted, all rows in the table will be deleted.

Example:

Suppose we have a table named "employees" with columns "employee_id", "first_name", and "last_name", and we want to delete the employee with ID 1:

Sql code

```
DELETE FROM employees

WHERE employee_id = 1;
```

In this example:

- We use the DELETE statement to remove data from the "employees" table.
- We specify the condition **employee_id = 1** to target the row with the employee ID 1 for deletion.

Points to note:

- The DELETE statement removes entire rows from a table based on the specified condition.
- Be cautious when using the DELETE statement without a WHERE clause, as it will delete all rows in the table.

- Always double-check the WHERE clause to ensure that you are targeting the correct rows for deletion, especially in production environments where data loss can have serious consequences.

Restoring the Database

Restoring a database typically involves recovering a database to a previous state using a backup. The process can vary depending on the database management system (DBMS) you are using. Here's a general outline of the steps involved in restoring a database:

Ensure Backup Availability: Make sure you have a recent backup of the database that you want to restore. Backups can be taken using database management tools or utilities provided by the DBMS.

Stop Access to the Database: Before restoring the database, you may need to stop any processes or applications that are accessing or modifying the database to prevent data corruption during the restoration process.

Restore Database Backup: Use the appropriate method provided by your DBMS to restore the database from the backup. This typically involves using a command-line tool, a graphical user interface (GUI), or SQL commands to restore the backup file to the database server.

Verify Restoration: After restoring the database, verify that the restoration was successful by checking for any errors or inconsistencies in the restored data. You can also perform tests to ensure that the database is functioning correctly.

Restart Access to the Database: Once the database has been successfully restored and verified, restart any processes or applications that access the database to resume normal operations.

Specific steps and commands for restoring a database may vary depending on the DBMS you are using (e.g., MySQL, PostgreSQL, Microsoft SQL Server, Oracle). It's essential to consult the documentation or guidelines provided by your DBMS vendor for detailed instructions on restoring a database from a backup. Additionally, always ensure that you have appropriate permissions and access rights to perform database restoration operations.