SQL

Created	@November 15, 2024 8:50 PM
Tags	

Introduction to Database

A database is an organized collection of data that can be easily accessed, managed, and updated. It is designed to store large amounts of information systematically to facilitate efficient retrieval and manipulation. A database is usually controlled by database management systems (DBMS).

- Example: A library database stores information about books, authors, borrowers, and borrowing history.

Database Management System (DBMS)

Database management system used to store, retrieve and run queries on data. It is an interface between an end user and a database, allowing users to create, read, update and delete data in the database. DBMS manage the data, the database engine and the database schema, allowing for data to be manipulated or extracted by users and other programs. This helps provides data security, concurrency and uniform data administration procedures.

Explain CRUD with Example

CRUD stands for Create, Read, Update and Delete, representing the basic operations performed on database records.

Example: Consider a table Users with fields UserID, Name, and Email :

Create: Insert a new user into the table.

Read: Retrieve user details.

Update: Modify user information.

Delete: Remove a user from the table.

Types of database

- Relational Databases: A Relational Database organizes data into structured tables consisting of rows and columns, where each row represents a unique record, and each column represents a specific attribute of that record. Relationships between tables are established using keys, such as primary and foreign keys. Relational databases follow the ACID properties (Atomicity, Consistency, Isolation, Durability) to ensure data integrity and reliability.
- NoSQL Database: A NoSQL Database (Not Only SQL) is a non-relational database designed to handle unstructured, semi-structured, or structured data. It is highly flexible and scalable, often used for large datasets and distributed systems. NoSQL databases do not rely on fixed schemas, and they are designed to accommodate changing data models.

What is SQL?

SQL (Structured Query Language) is a programming language used for managing and querying data in relational databases. It allows users to perform operations like inserting, updating, deleting, and retrieving data.

What are Database Queries?

Database queries are requests made to a database to perform specific operations, such as retrieving, updating, or deleting data. Queries are written in query languages like SQL.

Tables and Keys

Components of Tables:

A table in a database consists of several key components:

- Rows (Records or Tuples): Each row represents a single record or instance of the entity represented by the table. For example, in a "Students" table, each row represents a student.
- Columns (Attributes or Fields): Each column defines a specific attribute or characteristic of the entity. For instance, in a "Students" table, the columns might include StudentID, Name, Age, Grade, etc.
- Cell: A cell is the intersection of a row and column, containing a specific data value for that attribute in that record. For example, in the row for student 1, the cell under Name might contain "John Doe."

- Primary Key: A column or a combination of columns that uniquely identifies each row in the table. For example, StudentID could be the primary key in a "Students" table.
- Foreign Key: A column or set of columns that creates a relationship between two tables. It references the primary key in another table. For example, in a "Courses" table, StudentID might be a foreign key that links to the StudentID in the "Students" table.

Different Kinds of Keys:

- Primary Key:
  - The primary key is a unique identifier for each record in a table. It ensures that no two rows have the same value for the primary key column(s).
  - Example: In a "Students" table, the StudentID column could be the primary key since each student has a unique ID.
- Foreign Key:
  - A foreign key is a column (or a combination of columns) used to establish a link between the data in two tables. It refers to the primary key in another table, creating a relationship.
  - Example: In a "Courses" table, the StudentID column might be a foreign key that references the StudentID in the "Students" table, showing which student is enrolled in which course.
- Composite Key:
  - A composite key is formed by combining two or more columns to create a unique identifier for each record. The combination of values in these columns must be unique across the table.
  - Example: In a "CourseRegistrations" table, a combination of StudentID and CourseID might form a composite key, since a student can enroll in multiple courses, and a course can have multiple students.
- Surrogate key:
  - A surrogate key is a unique identifier for a record in a database table that is created artificially and has no meaningful relationship to the actual data.

It is typically used when there is no natural primary key or when using a natural key (like a social security number or email) is impractical or inefficient.

Example: Consider a Customers table where a natural key could be the Email address. However, emails can change, and using them as a primary key could lead to issues. Instead, you can use a surrogate key: CustomerID .

SQL Basics

SQL (Structured Query Language) is divided into several categories based on its function:

- **DQL (Data Query Language)**: This subset of SQL is used for querying and retrieving data from a database.
  - **Example:** `SELECT`
    - **Syntax:** `SELECT column_name FROM table_name;`
    - Retrieves data from the database.
- **DML (Data Manipulation Language)**: DML is used to manipulate the data in the database, such as inserting, updating, and deleting records.
  - **Examples:**
    - **INSERT INTO**: Used to add new rows.
      - **Syntax:** `INSERT INTO table_name (column1, column2) VALUES (value1, value2);`
    - **UPDATE**: Used to modify existing data.
      - **Syntax:** `UPDATE table_name SET column1 = value1 WHERE condition;`
    - **DELETE**: Removes data from the table.
      - **Syntax:** `DELETE FROM table_name WHERE condition;`
- **DDL (Data Definition Language)**: DDL is used to define and manage database structures like tables, schemas, and indexes.
  - **Examples:**
    - **CREATE TABLE**: Used to create a table.
      - **Syntax:** `CREATE TABLE table_name (column1 datatype, column2 datatype, ...);`

        - **ALTER TABLE**: Modifies an existing table structure.
            - **Syntax:** `ALTER TABLE table_name ADD column_name datatype;`
        - **DROP TABLE**: Deletes a table.
            - **Syntax:** `DROP TABLE table_name;`
- **DCL (Data Control Language)**: DCL is used for defining access controls and permissions on the database.
    - **Examples:**
        - **GRANT**: Gives a user specific privileges.
            - **Syntax:** `GRANT SELECT, INSERT ON table_name TO user;`
        - **REVOKE**: Removes previously granted privileges.
            - **Syntax:** `REVOKE SELECT ON table_name FROM user;`

Syntax for a Query

The general syntax of an SQL query is:

```
SELECT column1, column2, ...
FROM table_name
WHERE condition
ORDER BY column_name ASC|DESC
LIMIT number;
```

- `SELECT`: Specifies the columns you want to retrieve.
- `FROM`: Specifies the table to query from.
- `WHERE`: Filters records based on a condition.
- `ORDER BY`: Sorts the results in ascending or descending order.
- `LIMIT`: Limits the number of rows returned.

Example:

```
SELECT Name, Age
FROM Students
WHERE Age > 18
ORDER BY Name ASC
LIMIT 5;
```

Various Data Types

SQL supports a variety of data types, including:

- **INT**: Used to store integer values.
  - Example: `INT` (stores numbers like `1, 2, 100`).
- **DECIMAL**: Used to store exact numeric values with a specified precision and scale (e.g., for monetary values).
  - Example: `DECIMAL(10, 2)` (stores values like `12345.67`).
- **VARCHAR**: Stores variable-length strings.
  - Example: `VARCHAR(255)` (stores text like `John Doe`, `New York`).
- **BLOB (Binary Large Object)**: Used to store binary data, such as images, audio, or files.
  - Example: `BLOB` (stores binary data).
- **DATE**: Used to store date values (year, month, day).
  - Example: `DATE` (stores values like `2024-11-16`).
- **TIMESTAMP**: Stores both date and time values.
  - Example: `TIMESTAMP` (stores values like `2024-11-16 12:30:00`).

Steps to Create a Table in SQL

Syntax:

```
CREATE TABLE table_name (
column1 datatype,
column2 datatype,
...
);
```

Example:

```
CREATE TABLE Students (
StudentID INT PRIMARY KEY,
Name VARCHAR(100),
Age INT
);
```

PostgreSQL-specific commands:

- `\i`: This is used to run SQL commands from a file.
  - **Example:** `\i /path/to/file.sql`
- `\d`: Displays information about the table (like column names and types).
  - **Example:** `\d table_name`

Delete, Modify, and Inserting Data in a table

- **Delete a Table**: The `DROP TABLE` command deletes a table and all its data.

```
DROP TABLE table_name;
```

- **Modify Table (Add a Column)**: Use `ALTER TABLE` to modify a table.
  - **Adding a column**:

```
ALTER TABLE table_name ADD column_name datatype;
```

Example:

```
ALTER TABLE Students ADD Email VARCHAR(100);
```

- **Modify Table (Drop a Column)**: To remove a specific column from a table:

```
ALTER TABLE table_name DROP COLUMN column_name;
```

Example:

```
ALTER TABLE Students DROP COLUMN Age;
```

- **Inserting Data**: The `INSERT INTO` command is used to insert records into a table.

```
INSERT INTO table_name (column1, column2, ...) VALUES (value1, value2, ...);
```

Example:

```
INSERT INTO Students (StudentID, Name, Age)
VALUES (1, 'John Doe', 20);
```

Commands for Not Having NULL Values, UNIQUE Values, and Default Values

- **NOT NULL**: Ensures that a column cannot contain NULL values.
  - **Syntax:** `column_name datatype NOT NULL`
  - **Example:**

```
CREATE TABLE Students (
    StudentID INT NOT NULL,
    Name VARCHAR(100) NOT NULL
);
```

  In this example, both `StudentID` and `Name` cannot have NULL values.

- **UNIQUE**: Ensures that all values in a column are unique, meaning no two rows can have the same value for that column.
  - **Syntax:** `column_name datatype UNIQUE`
  - **Example:**

```
CREATE TABLE Students (
    StudentID INT PRIMARY KEY,
    Email VARCHAR(100) UNIQUE
);
```

  Here, the `Email` column will not allow duplicate values.

- **DEFAULT**: Provides a default value for a column if no value is specified when inserting a new row.
  - **Syntax:** `column_name datatype DEFAULT default_value`
  - **Example:**

```
CREATE TABLE Students (
```

```
    StudentID INT PRIMARY KEY,
    Name VARCHAR(100),
    EnrollmentDate DATE DEFAULT CURRENT_DATE
);
```

In this case, if no `EnrollmentDate` is provided, the default value will be the current date.

Serial

- **SERIAL**: Automatically generates unique integer values for a column, starting from 1.
  - **Syntax:**

```
CREATE TABLE Students (
    StudentID SERIAL PRIMARY KEY,
    Name VARCHAR(100)
);
```

In this example, `StudentID` will auto-increment each time a new record is inserted, starting from 1.

- **BIGSERIAL**: Similar to `SERIAL`, but it uses a larger integer type for auto-incrementing values.
  - **Syntax:**

```
CREATE TABLE Students (
    StudentID BIGSERIAL PRIMARY KEY,
    Name VARCHAR(100)
);
```

This is useful if you expect a very large number of records.

Comparison operators

Comparison operators are used to compare two values or expressions. They return a boolean result (TRUE or FALSE) based on whether the comparison is true or false. They are commonly used in SQL `WHERE` clauses to filter data based on specific conditions.

Common Comparison Operators:

- `=` **(Equal to)**: Checks if two values are equal.
  - Example: `SELECT * FROM Students WHERE Age = 20;`
- `<>` **(Not equal to)**: Checks if two values are not equal.
  - Example: `SELECT * FROM Students WHERE Age <> 20;`
- `>` **(Greater than)**: Checks if the left value is greater than the right.
  - Example: `SELECT * FROM Students WHERE Age > 18;`
- `<` **(Less than)**: Checks if the left value is less than the right.
  - Example: `SELECT * FROM Students WHERE Age < 25;`
- `>=` **(Greater than or equal to)**: Checks if the left value is greater than or equal to the right.
  - Example: `SELECT * FROM Students WHERE Age >= 18;`
- `<=` **(Less than or equal to)**: Checks if the left value is less than or equal to the right.
  - Example: `SELECT * FROM Students WHERE Age <= 25;`

Why We Use Them:

- **Filtering Data**: Comparison operators allow us to filter the records based on specific conditions in a query.
- **Precision**: They allow us to define exact conditions for querying and manipulating the data.
- **Flexibility**: They enable complex queries, allowing combinations of multiple conditions for advanced filtering.

Explain AND, OR

`AND` and `OR` are **logical operators** used to combine multiple conditions in a SQL query's `WHERE` clause. They help to refine the search by combining multiple comparisons.

- **AND**: Returns TRUE only if **all** conditions specified are TRUE.
  - Example:This query returns all students who are older than 18 **and** male. Both conditions must be true for the record to be included.

```
SELECT * FROM Students WHERE Age > 18 AND Gender = 'Male';
```

- **OR**: Returns TRUE if **any** of the conditions specified is TRUE.
  - Example:This query returns all students who are either older than 18 **or** female. If one condition is true, the row is included.

```
SELECT * FROM Students WHERE Age > 18 OR Gender = 'Female';
```

Using AND and OR Together:

- You can combine `AND` and `OR` to create more complex conditions. Use parentheses `()` to group conditions properly and ensure the correct order of evaluation.

```
SELECT * FROM Students
WHERE (Age > 18 AND Gender = 'Male') OR (Gender = 'Female' AND Enrol
lmentDate < '2023-01-01');
```

This will select students who are either male and older than 18 or female and enrolled before January 1st, 2023.

How to Update and Delete a Row, or Any Specific Information (SET and WHERE)

- **UPDATE**: Used to modify existing records in a table.
  - **Syntax:**

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

  - **Example** (update a specific student's age):

```
UPDATE Students
SET Age = 21
WHERE StudentID = 1;
```

    This query updates the `Age` of the student with `StudentID = 1` to 21.
  - **With Conditions**: You can specify a condition using `WHERE` to update only certain rows.

```
UPDATE Students
SET Age = 22
WHERE Name = 'John Doe';
```

    This updates the age to 22 only for the student named 'John Doe'.
- **DELETE**: Removes records from a table.
  - **Syntax:**

```
DELETE FROM table_name WHERE condition;
```

- **Example** (delete a specific student):

```
DELETE FROM Students
WHERE StudentID = 1;
```

  This query deletes the record where `StudentID` equals 1.

- **With Conditions**: Like `UPDATE`, `DELETE` can also be controlled with a `WHERE` clause to delete specific rows.

```
DELETE FROM Students
WHERE Age < 18;
```

  This deletes all students whose age is less than 18.

Important Notes:

- **SET**: Used with `UPDATE` to define what values to assign to which columns.
- **WHERE**: Defines the condition for updating or deleting specific rows. Without `WHERE`, **all rows** in the table will be affected (for `UPDATE` or `DELETE`), so be cautious.

Basic Queries

1. Fetch All Information from a Table

Query:

```
SELECT * FROM Students;
```

Explanation:

- `SELECT`: Specifies the columns to retrieve. `*` means all columns.
- `FROM`: Indicates the table to query data from (`Students`).

This query retrieves all rows and columns from the `Students` table.

2. Fetch Specific Columns

Query:

```
SELECT Name, Age FROM Students;
```

Explanation:

- `Name` and `Age`: Columns you want to retrieve.
- Retrieves only the `Name` and `Age` columns from the `Students` table.

3. Sort Data in Ascending or Descending Order

Query:

```
SELECT * FROM Students
ORDER BY Age DESC;
```

Explanation:

- `ORDER BY`: Specifies the column to sort data by.
- `Age`: The column to sort.
- `DESC`: Sorts data in descending order (newest to oldest). Replace with `ASC` for ascending order.

4. **Retrieve a Limited Number of Rows**

Query:

```
SELECT * FROM Students
LIMIT 3;
```

Explanation:

- `LIMIT`: Restricts the number of rows returned to 3.
- Retrieves the first 3 rows of the `Students` table.

5. **Retrieve Specific Data for a Student**

Query:

```
SELECT * FROM Students
WHERE StudentID = 1;
```

Explanation:

- `WHERE`: Filters rows based on the condition.
- `StudentID = 1`: Fetches rows where `StudentID` is 1.
- Retrieves all columns for the student whose ID is 1.

6. **Check for Specific Values (`IN`)**

Query:

```
SELECT * FROM Students
WHERE Age IN (18, 20, 22);
```

Explanation:

- Filters rows where the `Age` column matches any of the values in the list (18, 20, or 22).
- Example: Retrieves students who are either 18, 20, or 22 years old.

-Exclude Specific Values (`NOT IN`)

Query:

```
SELECT * FROM Students
```

```
WHERE Age NOT IN (18, 20, 22);
```

Explanation:

- Filters rows where the `Age` column does not match any of the values in the list.
- Example: Students aged other than 18, 20, or 22 are retrieved.

Comprehensive Guide to SQL Queries, Keys, Joins, and ER Diagrams

1. Creating a Foreign Key

A **foreign key** is a column or set of columns in a table that establishes a relationship with a primary key in another table. It ensures referential integrity by making sure the values in the foreign key column match the primary key values in the referenced table.

Syntax:

```
CREATE TABLE Orders (
    OrderID INT PRIMARY KEY,
    CustomerID INT,
    FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)
    ON DELETE SET NULL
);
```

Explanation:

- `FOREIGN KEY (CustomerID)`: Defines CustomerID in Orders as a foreign key.
- `REFERENCES Customers(CustomerID)`: Specifies the parent table (Customers) and column (CustomerID) being referenced.

Foreign Key Actions:

1. **ON DELETE SET NULL:**
    - If a referenced row is deleted in the parent table, the foreign key in the child table is set to `NULL`.

    Example:

```
CREATE TABLE Orders (
    OrderID INT PRIMARY KEY,
    CustomerID INT,
    FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)
    ON DELETE SET NULL
);
```

2. **ON DELETE CASCADE:**
    - If a referenced row is deleted in the parent table, all corresponding rows in the child table are also deleted.

    Example:

```
CREATE TABLE Orders (
    OrderID INT PRIMARY KEY,
    CustomerID INT,
    FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)
    ON DELETE CASCADE
);
```

2. Basic Queries

AS (Aliasing)

Query:

```
SELECT CustomerName AS Name, CustomerID AS ID FROM Customers;
```

Explanation:

- `AS` assigns temporary names (aliases) to columns or tables for better readability.

DISTINCT

Query:

```
SELECT DISTINCT Country FROM Customers;
```

Explanation:

- `DISTINCT` removes duplicate values, returning unique entries for the specified column.

SQL Functions

1. **Count**

   ```
   SELECT COUNT(*) FROM Orders;
   ```

   - Returns the total number of rows.

2. **AVG**

   ```
   SELECT AVG(Price) FROM Products;
   ```

   - Calculates the average of numeric column values.

3. **SUM**

   ```
   SELECT SUM(Price) FROM Products;
   ```

   - Returns the sum of numeric column values.

4. **MIN**

   ```
   SELECT MIN(Price) FROM Products;
   ```

   - Finds the smallest value.

5. **MAX**

   ```
   SELECT MAX(Price) FROM Products;
   ```

   - Finds the largest value.

GROUP BY

Syntax:

```
SELECT column_name, aggregate_function(column_name)
FROM table_name
GROUP BY column_name;
```

- `column_name`: Specifies the column by which the data will be grouped.
- `aggregate_function`: Performs calculations on the grouped data (e.g., `SUM`, `AVG`).

Query:

```
SELECT Country, COUNT(CustomerID) FROM Customers
GROUP BY Country;
```

Explanation:

- The `GROUP BY` statement in SQL is used to organize data into groups based on one or more columns. It is typically used in conjunction with aggregate functions like `COUNT`, `SUM`, `AVG`, `MIN`, and `MAX` to perform operations on each group.

3. Wildcards (LIKE)

Wildcards are used with the `LIKE` operator for pattern matching.

1. `%`: Represents zero or more characters.

   Query:

   ```
   SELECT * FROM Customers
   WHERE CustomerName LIKE 'A%';
   ```

   - Retrieves all customers whose names start with 'A'.

2. `_`: Represents a single character.

   Query:

```
SELECT * FROM Customers
WHERE CustomerName LIKE 'Jo_n';
```

- Retrieves all customers with names like 'John' or 'Joan'.

4. UNION

Combines results from two or more `SELECT` queries. The number of columns and their types must match.

Query:

```
SELECT City FROM Customers
UNION
SELECT City FROM Suppliers;
```

Explanation:

- Merges unique cities from both tables.

5. Joins

Joins combine rows from two or more tables based on a related column.

1. **LEFT JOIN Query:**

   ```
   SELECT Customers.CustomerName, Orders.OrderID
   FROM Customers
   LEFT JOIN Orders ON Customers.CustomerID = Orders.CustomerID;
   ```

   Explanation:

   - Returns all rows from the left table (`Customers`) and matching rows from the right table (`Orders`). Non-matching rows in the right table are `NULL`.

2. **RIGHT JOIN Query:**

   ```
   SELECT Orders.OrderID, Customers.CustomerName
   FROM Orders
   RIGHT JOIN Customers ON Orders.CustomerID = Customers.CustomerID;
   ```

Explanation:

- Returns all rows from the right table (`Customers`) and matching rows from the left table (`Orders`). Non-matching rows in the left table are `NULL`.

3. **FULL OUTER JOIN Query:**

```
SELECT Customers.CustomerName, Orders.OrderID
FROM Customers
FULL OUTER JOIN Orders ON Customers.CustomerID = Orders.CustomerI
D;
```

Explanation:

- Returns all rows when there is a match in either table. Non-matching rows are filled with `NULL`.

ON vs USING

- `ON`: Explicitly defines the condition.
- `USING`: Specifies the column name that both tables share.

6. Nested Queries

Subqueries are queries within queries.

Query:

```
SELECT CustomerName
FROM Customers
WHERE CustomerID IN (
   SELECT CustomerID FROM Orders WHERE OrderDate > '2024-01-01'
);
```

Explanation:

- Retrieves customers who placed orders after January 1, 2024.

ER Diagram with Relationship Cardinality

An **Entity-Relationship (ER) Diagram** is a graphical representation of entities and their relationships within a database system. It is a key part of database design, visually outlining the data structure and its logical connections.

Key Components of ER Diagrams:

1. **Entities:**
    - Represent objects or concepts in the database.
    - Depicted as rectangles.
    - Types:
        - **Strong Entity:** Exists independently (e.g., `Customer`).
        - **Weak Entity:** Depends on a strong entity for its existence (e.g., `OrderItem`).
2. **Attributes:**
    - Properties or characteristics of an entity.
    - Depicted as ovals connected to their entities.
    - Types:
        - **Primary Key:** Unique identifier for an entity.
        - **Composite Attribute:** An attribute composed of multiple sub-parts (e.g., `FullName` as `FirstName` + `LastName`).
        - **Derived Attribute:** Can be calculated (e.g., `Age` from `DateOfBirth`).
3. **Relationships:**
    - Describe how entities are associated.
    - Depicted as diamonds connecting entities.
    - Types:
        - **One-to-One (1:1):** Each instance of Entity A is related to one instance of Entity B and vice versa.
        - **One-to-Many (1:N):** One instance of Entity A is related to many instances of Entity B.

- **Many-to-Many (M:N):** Many instances of Entity A are related to many instances of Entity B.

Cardinality defines the number of instances of one entity that can be associated with instances of another entity. It determines the "multiplicity" of relationships.

Types of Cardinality:

1. One-to-One (1:1):

- **Definition:** Each instance of Entity A is associated with exactly one instance of Entity B.
- **Example:** A `Person` has exactly one `Passport`.
- **Diagram Representation:**

```
Person ——— 1:1 ——— Passport
```

2. One-to-Many (1:N):

- **Definition:** One instance of Entity A is associated with multiple instances of Entity B, but each instance of Entity B is associated with only one instance of Entity A.
- **Example:** A `Customer` places multiple `Orders`, but each `Order` belongs to one `Customer`.
- **Diagram Representation:**

```
Customer ——— 1:N ——— Order
```

3. Many-to-Many (M:N):

- **Definition:** Many instances of Entity A are associated with many instances of Entity B.
- **Example:** Students can enroll in multiple courses, and each course can have multiple students.
- **Diagram Representation:**

```
Student ——— M:N ——— Course
```

- **Implementation:** Represented using a **junction table** (e.g., `Enrollment` table with `StudentID` and `CourseID`).

Example:

- **Entities**: `Customers`, `Orders`.
- **Relationships**:
  - A customer can place multiple orders (1:N).
  - Each order is linked to one customer (N:1).

Diagram Example:

Customers Table:

CustomerID	CustomerName
1	Alice
2	Bob
3	Charlie

Orders Table:

OrderID	OrderDate	CustomerID
101	2024-11-20	1
102	2024-11-21	2
103	2024-11-22	1

- Customer **Alice** (CustomerID = 1) has placed **two orders**: Order #101 and Order #103.
- Customer **Bob** (CustomerID = 2) has placed **one order**: Order #102.
