Database Management Systems: SQL

Vikas Thammanna Gowda

01/14/2025

1 Introduction to SQL

1.1 What is SQL?

SQL (Structured Query Language) is a standard programming language for managing and manipulating relational databases. SQL is used to create, update, delete, and retrieve data from databases.

Example: Retrieving all records from a table named `employees`:

```
SELECT * FROM employees;
```

1.2 History and Evolution of SQL

- 1970s: Concept introduced by Edgar F. Codd in his relational model.
- 1974: SQL developed by IBM for their System R database.
- 1986: SQL became a standard by ANSI (American National Standards Institute).
- Evolution into multiple variants (MySQL, PostgreSQL, etc.) with extended features over time.

1.3 SQL Standards and Variants

- **MySQL:** Open-source, widely used for web development.
- **PostgreSQL:** Advanced open-source database with support for custom data types and functions.
- **SQL Server:** Microsoft's proprietary RDBMS.
- **Oracle Database:** Known for high scalability and enterprise features.

1.4 Key Features of SQL

1. Declarative Nature

SQL is a declarative language, meaning you specify what you want the database to do without describing how to do it. This contrasts with procedural languages where detailed steps are required.

2. Wide Adoption

SQL is supported by most relational database management systems (RDBMS), making it a universal tool for database management.

3. Flexibility

SQL can handle complex queries and is suitable for both small and large datasets.

4 ACID Compliance

SQL-based systems often support ACID (Atomicity, Consistency, Isolation, Durability) properties, ensuring reliable transactions.

1.5 Components of SQL

SQL is broadly divided into the following sublanguages:

1. Data Definition Language (DDL)

Used for defining and managing database structures like tables, schemas, and indexes.

Commands:

- `CREATE`: Creates a new database object (e.g., table, database, index).
- `ALTER`: Modifies an existing database object.
- `DROP`: Deletes database objects.
- `TRUNCATE`: Removes all records from a table without logging individual row deletions.

Example:

```
CREATE TABLE Students (
    StudentID INT PRIMARY KEY,
    Name VARCHAR(100),
    Age INT,
    Major VARCHAR(50)
);
```

2. Data Manipulation Language (DML)

Used for querying and modifying data in the database.

Commands:

- `SELECT`: Retrieves data from tables.
- `INSERT`: Adds new records.
- `UPDATE`: Modifies existing records.
- `DELETE`: Removes records.

Example:

```
INSERT INTO Students (StudentID, Name, Age, Major)
VALUES (1, ’Alice’, 20, ’Computer Science’);

SELECT * FROM Students WHERE Age > 18;
```

3. Data Query Language (DQL)

Subset of DML focused specifically on data retrieval.

Primary Command:

- `SELECT`

Example:

```
SELECT Name, Major FROM Students;
```

4. Data Control Language (DCL)

Used to manage permissions and access control.

Commands:

- GRANT: Grants privileges to users.
- REVOKE: Revokes privileges.

Example:

```
GRANT SELECT, INSERT ON Students TO 'user1';
```

5. Transaction Control Language (TCL)

Used for managing transactions.

Commands:

- COMMIT: Saves changes to the database.
- ROLLBACK: Reverts changes since the last commit.
- SAVEPOINT: Sets a point in a transaction to rollback to.

Example:

```
BEGIN TRANSACTION;
INSERT INTO Students (StudentID, Name, Age, Major) VALUES (2, 'Bob', 21, 'Math');
ROLLBACK;
```

1.6 Advantages of SQL

- Simple and easy to learn.
- Portable across systems.
- Efficient for handling large datasets.
- Scalable for growing applications.

1.7 Limitations of SQL

- Relational model constraints may not suit all use cases.
- Performance can degrade with poorly optimized queries.
- Limited support for unstructured data compared to NoSQL databases.

2 SQL Basics

2.1 Data Types

SQL provides a range of data types to store data efficiently.

1. **Numeric:**

   - **INT:** Integer values (e.g., 1, 2, 3).
   - **FLOAT/REAL:** Approximate decimal values (e.g., 3.14).
   - **DECIMAL/NUMERIC:** Exact decimal values (e.g., 100.50).

   Example:

   ```
   CREATE TABLE numbers (id INT, value DECIMAL(5, 2));
   ```

2. **Character:**

   - **CHAR:** Fixed-length string (e.g., `CHAR(5)` always stores 5 characters).
   - **VARCHAR:** Variable-length string (e.g., `VARCHAR(50)` can store up to 50 characters).
   - **TEXT:** Large strings.

   Example:

   ```
   CREATE TABLE names (name VARCHAR(50));
   ```

3. **Date and Time:**

   - **DATE:** Stores only the date (e.g., 2024-12-22).
   - **TIME:** Stores only the time (e.g., 14:30:00).
   - **TIMESTAMP:** Stores both date and time.

   Example:

   ```
   CREATE TABLE events (event_date DATE, event_time TIME, created_at TIMESTAMP);
   ```

4. **Miscellaneous:**

   - **BOOLEAN:** Stores `TRUE` or `FALSE`.
   - **BLOB:** Binary Large Objects for images, files, etc.

2.2 SQL Syntax

- **Case Sensitivity:** SQL keywords (e.g., `SELECT`, `FROM`) are case-insensitive, but identifiers (e.g., table names) may be case-sensitive depending on the database.

- **Statements and Semicolon:** Most SQL statements end with a semicolon.

Example:

```
SELECT name FROM students;
```

2.3 Basic Commands

- **SELECT:** Retrieves data.
- **FROM:** Specifies the table.
- **WHERE:** Filters data.

Example:

```
SELECT name, age FROM students WHERE age > 18;
```

3 Keys in SQL

In SQL, keys are fundamental components that establish relationships between tables and ensure the integrity and uniqueness of data within a relational database. Below is an overview of the various types of keys:

3.1 Super Key

A Super Key is a set of one or more attributes that can uniquely identify a tuple (row) in a relation (table). It may contain extra attributes that are not necessary for unique identification. Every relation has at least one Super Key, which is the set of all its attributes. Additionally, all Candidate Keys and the Primary Key are subsets of Super Keys. **Example:** In an employee table, the combination of {Employee_ID, Employee_Name} is a Super Key if Employee_ID alone can uniquely identify each employee.

3.2 Candidate Key

A Candidate Key is a minimal Super Key, meaning it has no redundant attributes. It uniquely identifies a tuple and removing any attribute would cause it to lose its unique identification property. A table can have multiple Candidate Keys, and all Candidate Keys are Super Keys, though not all Super Keys are Candidate Keys. **Example:** In the same employee table, both {Employee_ID} and {SSN} could be Candidate Keys if each uniquely identifies an employee.

3.3 Primary Key

A Primary Key is a Candidate Key chosen by the database designer to uniquely identify tuples within a table. It cannot contain NULL values, and there can be only one Primary Key per table. Primary Keys are selected based on criteria like simplicity or familiarity. **Example:** If {Employee_ID} is chosen as the Primary Key in an employee table, it will uniquely identify each employee.

3.4 Alternate Key

An Alternate Key is any Candidate Key that is not chosen as the Primary Key. These keys serve as alternative unique identifiers and can be used for indexing or enforcing uniqueness constraints. **Example:** If {Employee_ID} is the Primary Key, then {SSN} could be an Alternate Key in the employee table.

3.5 Foreign Key

A Foreign Key is an attribute or set of attributes in one table that references the Primary Key of another table. It establishes relationships between tables and ensures referential integrity by enforcing valid references. Foreign Keys can accept NULL values if the relationship is optional. **Example:** In a 'Department' table, {Manager_ID} could be a Foreign Key referencing {Employee_ID} in the 'Employee' table, indicating which employee manages the department.

3.6 Composite Key

A Composite Key is a key composed of two or more attributes that together uniquely identify a tuple in a table. It is necessary when a single attribute is not sufficient for unique identification. All attributes in the Composite Key are essential, and removing any would disrupt uniqueness. **Example:** In an 'Order_Details' table, the combination of {Order_ID, Product_ID} could serve as a Composite Key to uniquely identify each item in an order.

3.7 Surrogate Key

A Surrogate Key is a system-generated unique identifier for a tuple, typically a sequential number. It has no intrinsic meaning or business logic and is often used to simplify key structures, especially when natural keys are large or complex. **Example:** A 'Customers' table might use {Customer_ID} as a surrogate key while storing customer details like {Customer_Name} and {Contact_Info} in other columns.

3.8 Natural Key

A Natural Key is formed of attributes that already exist in the real world and have a logical relationship to the data. It has inherent business meaning and can be used directly as a Primary Key if it is unique and stable. However, changes in business logic can affect its suitability. **Example:** {Social_Security_Number} (SSN) in a 'Citizens' table is an example of a Natural Key, as it inherently identifies individuals in the real world.

3.9 Compound Key

A Compound Key is similar to a Composite Key, consisting of two or more attributes that together uniquely identify a tuple. All components are necessary for unique identification. Compound Keys are often used in junction tables for many-to-many relationships. **Example:** In a 'Student_Courses' table, the combination of {Student_ID, Course_ID} could serve as a Compound Key to uniquely identify each student's enrollment in a course.

3.10 Secondary Key

A Secondary Key is an attribute or set of attributes used for retrieving data but may not be unique. It is often indexed to improve search performance, though it does not enforce uniqueness. **Example:** {Author_Name} in a 'Books' table could be a Secondary Key used to retrieve all books written by a particular author.

3.11 Unique Key

A Unique Key is an attribute or set of attributes that enforces uniqueness for the column(s) it is defined on. It allows for NULL values and ensures data integrity by preventing duplicate entries. A table can have multiple Unique Keys. **Example:** {Email_Address} in a 'Users' table could be defined as a Unique Key to ensure no two users have the same email while allowing for NULL values.

3.12 Composite Primary Key

A Composite Primary Key is a Primary Key that consists of two or more attributes. It ensures uniqueness using the combination of attributes and is common in associative entities representing many-to-many relationships. **Example:** In a 'Registration' table, the combination of {Student_ID, Class_ID} could serve as a Composite Primary Key, uniquely identifying each student's enrollment in a specific class.

Understanding and appropriately implementing these keys in SQL is crucial for designing robust and efficient relational databases. They play a vital role in maintaining data integrity and establishing clear relationships between tables.

4 Understanding NULL Values in Databases

4.1 What are NULL Values in a Database?

A NULL value in a database represents missing, undefined, or unknown data in a field. It is not the same as:

- **Zero (0):** A numerical value.
- **Empty String (‘’):** A defined, blank text value.

Instead, NULL indicates the absence of any value or that the value is unknown at the moment.

4.2 Why Do NULL Values Exist?

- **Incomplete Data:** When a record is partially created but lacks certain information. *Example:* A student’s phone number might not be available at the time of data entry.
- **Irrelevant Fields:** Some fields may not apply to certain rows. *Example:* An employee with no middle name.
- **Data Entry Errors:** Inadvertent omission of information.
- **Future Data Updates:** Placeholder for data that will be entered later.

4.3 Impact of NULL Values in Databases

1. Query Handling

Comparisons with NULL using standard operators (=, !=, etc.) do not work. SQL uses `IS NULL` or `IS NOT NULL` to check for NULL values.

```
SELECT * FROM Students WHERE PhoneNumber IS NULL;
```

2. Aggregate Functions

Most aggregate functions (e.g., `SUM`, `AVG`) ignore NULL values.

```
SELECT AVG(Salary) FROM Employees;
```

Explanation: If some employees have NULL salaries, they are excluded from the calculation.

3. NULL in Joins

Joins involving NULL values may fail to match records correctly.

4. Logic Ambiguity

NULL introduces three-valued logic in SQL: **TRUE, FALSE, UNKNOWN** (when involving NULL).

```
SELECT * FROM Students WHERE Email = NULL; -- Will not return results
```

4.4 Examples of NULL Value Issues

Example 1: Incomplete Data

StudentID	Name	PhoneNumber
1	Alice	123-456-7890
2	Bob	NULL

Query to find students without phone numbers:

```
SELECT Name FROM Students WHERE PhoneNumber IS NULL;
```

Result: Bob

Example 2: Problems with Aggregation

EmployeeID	Salary
1	5000
2	NULL
3	7000

Query for the average salary:

```
SELECT AVG(Salary) FROM Employees;
```

Result: (5000 + 7000) / 2 = 6000 (ignores NULL).

4.5 How to Handle NULL Values?

Default Values

Use a default value when defining the table schema to avoid NULL.

```
CREATE TABLE Students (
    StudentID INT,
    PhoneNumber VARCHAR(15) DEFAULT ’Unknown’
);
```

Data Validation

Enforce mandatory fields using `NOT NULL` constraints.

```
CREATE TABLE Students (
    StudentID INT,
    Name VARCHAR(50) NOT NULL
);
```

Replacing NULL in Queries

Use the `COALESCE` function to replace NULL with a default value.

```
SELECT Name, COALESCE(PhoneNumber, ’No Phone’) AS Contact FROM Students;
```

Avoid NULLs in Logic

Use NULL-safe logic and query patterns.

4.6 Best Practices

- Use NULL only when necessary; prefer default values or optional fields for clarity.
- Design tables with normalization to minimize the need for NULL.
- Understand the behavior of NULL in SQL operations to avoid logic and aggregation errors.

NULL values are an essential feature of databases but require careful handling to maintain data integrity and avoid ambiguous or incorrect results.

5 Creating and Managing Tables

5.1 Defining Tables

5.1.1 CREATE TABLE

To define the structure of a table, use the `CREATE TABLE` statement.

Example: Create a `students` table with specific columns:

```
CREATE TABLE students (
    student_id INT PRIMARY KEY,
    name VARCHAR(50) NOT NULL,
    age INT CHECK (age > 0),
    grade CHAR(1) DEFAULT ‘C’,
    enrollment_date DATE NOT NULL
);
```

Explanation:

- **PRIMARY KEY:** Uniquely identifies each row.
- **NOT NULL:** Ensures a column cannot have `NULL` values.
- **CHECK:** Adds a condition (e.g., `age` must be greater than 0).
- **DEFAULT:** Assigns a default value if none is provided.

Result: A table named `students` is created with the specified structure.

5.1.2 Primary Keys

Primary keys uniquely identify each record in a table. Only one primary key is allowed per table.

Example: The `student_id` column in the `students` table is a primary key:

```
CREATE TABLE example_students (
    student_id INT PRIMARY KEY,
    name VARCHAR(50)
);
```

5.1.3 Foreign Keys

Foreign keys enforce relationships between tables.

Example: Create a `courses` table where `instructor_id` references the `instructors` table:

```
CREATE TABLE instructors (
    instructor_id INT PRIMARY KEY,
    name VARCHAR(50)
);

CREATE TABLE courses (
    course_id INT PRIMARY KEY,
    course_name VARCHAR(100),
    instructor_id INT,
    FOREIGN KEY (instructor_id) REFERENCES instructors(instructor_id)
);
```

Explanation: The `instructor_id` in `courses` is a foreign key referencing `instructor_id` in `instructors`.

5.1.4 Constraints

Constraints enforce rules at the table level.

Constraint	Description
NOT NULL	Ensures a column cannot have `NULL` values.
UNIQUE	Ensures all values in a column are distinct.
CHECK	Validates data based on a condition.
DEFAULT	Assigns a default value if no value is provided during insertion.

5.2 Altering Tables

5.2.1 Adding/Removing Columns

Add or remove columns using the `ALTER TABLE` statement.

Example: Add a column `email` to the `students` table:

```
ALTER TABLE students
ADD email VARCHAR(100);
```

Example: Remove the `email` column:

```
ALTER TABLE students
DROP COLUMN email;
```

5.2.2 Modifying Columns

Modify the data type or constraints of a column.

Example: Change the `grade` column to `VARCHAR(2)`:

```
ALTER TABLE students
MODIFY grade VARCHAR(2);
```

5.2.3 Adding/Removing Constraints

Add or remove constraints after the table is created.

Example: Add a `UNIQUE` constraint on the `name` column:

```
ALTER TABLE students
ADD CONSTRAINT unique_name UNIQUE (name);
```

Example: Remove the `UNIQUE` constraint:

```
ALTER TABLE students
DROP CONSTRAINT unique_name;
```

5.3 Dropping Tables

5.3.1 DROP TABLE

Use `DROP TABLE` to permanently delete a table and its data.

Example:

```
DROP TABLE students;
```

Result: The `students` table no longer exists.

5.3.2 Cascade Deletions (CASCADE)

When a table has dependencies (e.g., foreign keys), use `CASCADE` to delete related data automatically.

Example: Drop the `instructors` table and all references to it:

```
DROP TABLE instructors CASCADE;
```

5.4 Practical Example with Results

1. Create Tables and Insert Data:

```
CREATE TABLE departments (
    dept_id INT PRIMARY KEY,
    dept_name VARCHAR(50) NOT NULL
);

CREATE TABLE employees (
    emp_id INT PRIMARY KEY,
    name VARCHAR(50),
    dept_id INT,
    FOREIGN KEY (dept_id) REFERENCES departments(dept_id)
);

INSERT INTO departments VALUES
(1, ‘HR’),
(2, ‘Engineering’),
(3, ‘Marketing’);

INSERT INTO employees VALUES
(101, ‘Alice’, 1),
(102, ‘Bob’, 2),
(103, ‘Charlie’, 3);
```

Result:

Departments Table:

Dept_ID	**Dept_Name**
1	HR
2	Engineering
3	Marketing

Employees Table:

Emp_ID	**Name**	**Dept_ID**
101	Alice	1
102	Bob	2
103	Charlie	3

2. Alter Table: Add a column `salary` to the `employees` table:

```
ALTER TABLE employees
ADD salary DECIMAL(10, 2);
```

Result:

Emp_ID	**Name**	**Dept_ID**	**Salary**
(Salary column added, no data yet)			

3. Drop Table with CASCADE: Drop the `departments` table and cascade deletions.

```
DROP TABLE departments CASCADE;
```

Result: The `departments` table and all related rows in `employees` referencing `dept_id` are deleted.

6 Modifying Data

SQL allows modification of data in a database through `INSERT`, `UPDATE`, and `DELETE` commands.

6.1 Inserting Data

6.1.1 Single Row Insertion

Insert a single row into a table.

Example Table: `employees`

```
CREATE TABLE employees (
    employee_id INT PRIMARY KEY,
    name VARCHAR(50),
    position VARCHAR(50),
    salary DECIMAL(10, 2),
    hire_date DATE
);

-- Insert a single row
INSERT INTO employees (employee_id, name, position, salary, hire_date)
VALUES (1, ‘Alice’, ‘Manager’, 75000.00, ‘2023-01-15’);
```

Result:

Employee_ID	Name	Position	Salary	Hire Date
1	Alice	Manager	75000.00	2023-01-15

6.1.2 Multiple Rows Insertion

Insert multiple rows into a table in one statement.

```
INSERT INTO employees (employee_id, name, position, salary, hire_date)
VALUES
(2, ‘Bob’, ‘Engineer’, 65000.00, ‘2023-03-10’),
(3, ‘Charlie’, ‘Technician’, 55000.00, ‘2023-05-01’),
(4, ‘Diana’, ‘Analyst’, 70000.00, ‘2023-07-20’);
```

Result:

Employee_ID	Name	Position	Salary	Hire Date
1	Alice	Manager	75000.00	2023-01-15
2	Bob	Engineer	65000.00	2023-03-10
3	Charlie	Technician	55000.00	2023-05-01
4	Diana	Analyst	70000.00	2023-07-20

6.2 Updating Data

6.2.1 Single Column Update

Update a single column for specific rows.

Example: Increase salary of Charlie by 10%.

```
UPDATE employees
SET salary = salary * 1.10
WHERE name = ‘Charlie’;
```

Result:

Employee_ID	Name	Position	Salary	Hire Date
3	Charlie	Technician	60500.00	2023-05-01

6.2.2 Multiple Column Update

Update multiple columns at once.

Example: Change Diana's position to Senior Analyst and increase her salary.

```
UPDATE employees
SET position = 'Senior Analyst', salary = 80000.00
WHERE name = 'Diana';
```

Result:

Employee_ID	Name	Position	Salary	Hire Date
4	Diana	Senior Analyst	80000.00	2023-07-20

6.2.3 Conditional Updates

Update data based on conditions.

Example: Increase salary by 5% for employees hired before 2023-04-01.

```
UPDATE employees
SET salary = salary * 1.05
WHERE hire_date < '2023-04-01';
```

Result:

Employee_ID	Name	Position	Salary	Hire Date
1	Alice	Manager	78750.00	2023-01-15
2	Bob	Engineer	68250.00	2023-03-10

6.3 Deleting Data

6.3.1 DELETE with Conditions

Delete specific rows based on a condition.

Example: Remove employees whose salary is below $60,000.

```
DELETE FROM employees
WHERE salary < 60000.00;
```

Result:

Employee_ID	Name	Position	Salary	Hire Date
1	Alice	Manager	78750.00	2023-01-15
2	Bob	Engineer	68250.00	2023-03-10
4	Diana	Senior Analyst	80000.00	2023-07-20

6.3.2 Deleting All Rows

Remove all rows from a table without deleting the structure.

Example:

```
TRUNCATE TABLE employees;
```

Result: The table `employees` is now empty:

Employee_ID	Name	Position	Salary	Hire Date
(No Rows)				

7 Data Retrieval

7.1 Basic Queries

- **Selecting Columns:** Retrieve specific columns from a table. **Example:**

  ```
  SELECT name, age FROM students;
  ```

- **Renaming Columns (AS):** Use aliases to rename columns in the output. **Example:**

  ```
  SELECT name AS student_name, age AS student_age FROM students;
  ```

7.2 Filtering Data

- **Comparison Operators:**
  - =: Equal
  - ! =: Not Equal
  - <, >, <=, >=: Less than, Greater than, etc.

  Example:

  ```
  SELECT * FROM students WHERE age >= 18;
  ```

- **Logical Operators:**
  - **AND:** Combines multiple conditions; all must be true.
  - **OR:** Any condition must be true.
  - **NOT:** Negates a condition.

  Example:

  ```
  SELECT * FROM students WHERE age >= 18 AND grade = ‘A’;
  ```

- **Pattern Matching (LIKE, ILIKE):**
  - **LIKE:** Case-sensitive pattern matching.
  - **ILIKE:** Case-insensitive pattern matching.

  Example:

  ```
  SELECT * FROM students WHERE name LIKE ‘A%’;
  ```

- **Null Handling:**
  - **IS NULL:** Checks for null values.
  - **IS NOT NULL:** Checks for non-null values.

  Example:

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

7.3 Sorting and Limiting Results

- **ORDER BY:** Sorts query results.
  - Default: Ascending order.
  - Use DESC for descending.

  Example:

  ```
  SELECT * FROM students ORDER BY age DESC;
  ```

- **LIMIT, OFFSET:** Limits the number of rows and skips a specified number of rows.
  Example:

  ```
  SELECT * FROM students ORDER BY age LIMIT 5 OFFSET 2;
  ```

7.4 Table Example

Create a table students and populate it with data:

```
CREATE TABLE students (
    student_id INT PRIMARY KEY,
    name VARCHAR(50),
    age INT,
    grade CHAR(1),
    enrollment_date DATE
);

INSERT INTO students VALUES
(1, ‘Alice’, 20, ‘A’, ‘2023-09-01’),
(2, ‘Bob’, 22, ‘B’, ‘2022-09-01’),
(3, ‘Charlie’, 19, ‘C’, ‘2023-06-01’),
(4, ‘Diana’, 21, ‘A’, ‘2021-09-01’),
(5, ‘Eve’, 20, ‘B’, ‘2023-01-01’);
```

7.4.1 Resulting Table

Here is the resulting table for the provided SQL statements:

Student_ID	Name	Age	Grade	Enrollment_Date
1	Alice	20	A	2023-09-01
2	Bob	22	B	2022-09-01
3	Charlie	19	C	2023-06-01
4	Diana	21	A	2021-09-01
5	Eve	20	B	2023-01-01

7.5 Query Examples

7.5.1 Selecting Columns

Retrieve the name and grade columns from the students table:

```
SELECT name, grade FROM students;
```

Result:

Name	Grade
Alice	A
Bob	B
Charlie	C
Diana	A
Eve	B

7.5.2 Renaming Columns

Use `AS` to rename `enrollment_date` as `Joined`:

```
SELECT name, enrollment_date AS Joined FROM students;
```

Result:

Name	Joined
Alice	2023-09-01
Bob	2022-09-01
Charlie	2023-06-01
Diana	2021-09-01
Eve	2023-01-01

7.5.3 Filtering Data

Retrieve students aged 20 or older with grade `A`:

```
SELECT name, age, grade FROM students
WHERE age >= 20 AND grade = ‘A’;
```

Result:

Name	Age	Grade
Alice	20	A
Diana	21	A

7.5.4 Sorting Data

Sort students by age in descending order:

```
SELECT name, age FROM students
ORDER BY age DESC;
```

Result:

Name	Age
Bob	22
Diana	21
Alice	20
Eve	20
Charlie	19

7.5.5 Limiting Results

Retrieve the youngest 2 students:

```
SELECT name, age FROM students
ORDER BY age ASC
LIMIT 2;
```

Result:

Name	Age
Charlie	19
Alice	20

8 Advanced Data Retrieval

8.1 Aggregate Functions

Aggregate functions perform calculations on a set of values and return a single value.

Function	Description	Example Use
COUNT	Counts rows or non-null values.	`COUNT(*)`
SUM	Computes the sum of numeric values.	`SUM(column)`
AVG	Calculates the average of numeric values.	`AVG(column)`
MIN	Finds the minimum value.	`MIN(column)`
MAX	Finds the maximum value.	`MAX(column)`

8.1.1 Example Table: sales

```
CREATE TABLE sales (
    sale_id INT PRIMARY KEY,
    product VARCHAR(50),
    quantity INT,
    price_per_unit DECIMAL(10, 2),
    sale_date DATE
);

INSERT INTO sales VALUES
(1, ‘Laptop’, 2, 1000.00, ‘2023-12-01’),
(2, ‘Phone’, 5, 600.00, ‘2023-12-02’),
(3, ‘Tablet’, 3, 400.00, ‘2023-12-03’),
(4, ‘Laptop’, 1, 950.00, ‘2023-12-04’),
(5, ‘Phone’, 2, 620.00, ‘2023-12-05’);
```

8.2 Resulting Table

Here is the resulting table for the provided SQL statements:

Sale_ID	Product	Quantity	Price Per Unit	Sale Date
1	Laptop	2	1000.00	2023-12-01
2	Phone	5	600.00	2023-12-02
3	Tablet	3	400.00	2023-12-03
4	Laptop	1	950.00	2023-12-04
5	Phone	2	620.00	2023-12-05

Examples:

1. **COUNT:** Count the number of sales records:

   ```
   SELECT COUNT(*) AS total_sales FROM sales;
   ```

   Result:

Total Sales
5

2. **SUM:** Calculate total revenue:

   ```
   SELECT SUM(quantity * price_per_unit) AS total_revenue FROM sales;
   ```

   Result:

Total Revenue
7200.00

3. **AVG:** Average price per unit:

   ```
   SELECT AVG(price_per_unit) AS avg_price FROM sales;
   ```

   Result:

Average Price
714.00

4. **MIN/MAX:** Find the minimum and maximum prices:

   ```
   SELECT MIN(price_per_unit) AS min_price, MAX(price_per_unit) AS max_price FROM sales;
   ```

   Result:

Min Price	Max Price
400.00	1000.00

8.3 Grouping Data

GROUP BY groups rows with the same value in specified columns, often used with aggregate functions to summarize data.

Example: Total quantity sold for each product:

```
SELECT product, SUM(quantity) AS total_quantity
FROM sales
GROUP BY product;
```

Result:

Product	Total Quantity
Laptop	3
Phone	7
Tablet	3

8.4 Filtering Groups with HAVING

HAVING filters groups created by GROUP BY. Unlike WHERE, which filters rows, HAVING filters aggregated data.

Example: Products with total sales greater than $2000:

```
SELECT product, SUM(quantity * price_per_unit) AS total_sales
FROM sales
GROUP BY product
HAVING total_sales > 2000;
```

Result:

Product	Total Sales
Laptop	2950.00
Phone	4220.00

8.5 Joins

Joins combine data from two or more tables based on related columns.

8.5.1 Example Tables

```
CREATE TABLE customers (
    customer_id INT PRIMARY KEY,
    name VARCHAR(50),
    city VARCHAR(50)
);

INSERT INTO customers VALUES
(1, ‘Alice’, ‘New York’),
(2, ‘Bob’, ‘Los Angeles’),
(3, ‘Charlie’, ‘Chicago’);

CREATE TABLE orders (
    order_id INT PRIMARY KEY,
    customer_id INT,
    order_date DATE,
    amount DECIMAL(10, 2),
    FOREIGN KEY (customer_id) REFERENCES customers(customer_id)
);

INSERT INTO orders VALUES
(101, 1, ‘2023-12-01’, 500.00),
(102, 2, ‘2023-12-02’, 150.00),
(103, 3, ‘2023-12-03’, 300.00),
(104, 1, ‘2023-12-04’, 250.00);
```

8.6 Resulting Tables

Here are the resulting tables for the provided SQL statements:
Customers Table

Customer_ID	Name	City
1	Alice	New York
2	Bob	Los Angeles
3	Charlie	Chicago

Orders Table

Order_ID	Customer_ID	Order Date	Amount
101	1	2023-12-01	500.00
102	2	2023-12-02	150.00
103	3	2023-12-03	300.00
104	1	2023-12-04	250.00

Examples:

1. **Inner Join:** Retrieve orders with customer details:

    ```
    SELECT customers.name, orders.order_date, orders.amount
    FROM customers
    INNER JOIN orders ON customers.customer_id = orders.customer_id;
    ```

Result:

Name	Order Date	Amount
Alice	2023-12-01	500.00
Bob	2023-12-02	150.00
Charlie	2023-12-03	300.00
Alice	2023-12-04	250.00

2. **Outer Join:** Retrieve all customers and their orders (use `LEFT JOIN`):

```
SELECT customers.name, orders.order_date, orders.amount
FROM customers
LEFT JOIN orders ON customers.customer_id = orders.customer_id;
```

Result:

Name	Order Date	Amount
Alice	2023-12-01	500.00
Bob	2023-12-02	150.00
Charlie	2023-12-03	300.00
Alice	2023-12-04	250.00

8.7 Subqueries

- **Scalar Subqueries:** Return a single value. Example: Find the highest order amount:

```
SELECT name, city
FROM customers
WHERE customer_id = (
    SELECT customer_id FROM orders WHERE amount = (SELECT MAX(amount) FROM orders)
);
```

Result:

Name	City
Alice	New York

- **Correlated Subqueries:** Depend on outer query. Example: Customers who have placed orders above their city's average:

```
SELECT name
FROM customers c
WHERE EXISTS (
    SELECT 1
    FROM orders o
    WHERE o.customer_id = c.customer_id AND o.amount > (
        SELECT AVG(amount) FROM orders o2 WHERE o2.customer_id = o.customer_id
    )
);
```

Result:

Name
Alice

9 Indexing and Optimization

Indexes improve query performance by allowing faster data retrieval. Proper indexing and optimization can significantly enhance database efficiency.

9.1 Creating Indexes

Indexes are created on columns to speed up data retrieval.

9.1.1 Single Column Index

An index on a single column allows faster lookups based on that column.

Example: Create an index on the `name` column of the `employees` table:

```
CREATE INDEX idx_name ON employees(name);
```

Explanation: Speeds up queries like:

```
SELECT * FROM employees WHERE name = ‘Alice’;
```

9.1.2 Composite Index

A composite index includes multiple columns and is used for queries filtering by more than one column.

Example: Create a composite index on `dept_id` and `name`:

```
CREATE INDEX idx_dept_name ON employees(dept_id, name);
```

Explanation: Useful for queries like:

```
SELECT * FROM employees WHERE dept_id = 1 AND name = ‘Alice’;
```

9.1.3 Unique Index

A unique index ensures all values in the indexed column(s) are distinct.

Example: Create a unique index on the `email` column:

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

Explanation:

- Prevents duplicate values in the `email` column.
- Throws an error for duplicate insertion:

```
INSERT INTO employees (emp_id, name, email) VALUES (104, ‘Diana’, ‘diana@example.com’);
```

9.2 Deleting Indexes

Indexes can be deleted when they are no longer needed.

9.2.1 DROP INDEX

Deletes a specific index from a table.

Example: Delete the `idx_name` index:

```
DROP INDEX idx_name;
```

Explanation: After deletion, queries involving `name` will no longer benefit from indexing.

9.3 Query Optimization

9.3.1 Understanding Query Plans (EXPLAIN)

Use EXPLAIN to analyze how a query is executed and determine whether indexes are used.

Example: Analyze a query retrieving employees by name:

```
EXPLAIN SELECT * FROM employees WHERE name = ‘Alice’;
```

Result:

ID	Select Type	Table	Type	Possible Keys	Key	Rows	Extra
1	SIMPLE	employees	ref	idx_name	idx_name	1	Using index

Explanation:

- **Key:** Shows which index is used.
- **Rows:** Shows the number of rows scanned.
- **Extra:** Indicates if the index was fully utilized.

9.3.2 Using Indexes Efficiently

- Sort your data retrieval conditions to match the index order: Composite indexes are most effective when queries filter on their leading columns.
- Avoid wrapping indexed columns with functions: Queries like WHERE LOWER(name) = ‘alice’ won’t use the index.

Example of Efficient Query: If idx_dept_name exists:

```
SELECT * FROM employees WHERE dept_id = 1 AND name = ‘Alice’;
```

9.3.3 Avoiding Common Performance Pitfalls

- Avoid using SELECT *: Fetch only required columns to reduce data transfer.

  ```
  SELECT name, salary FROM employees WHERE dept_id = 1;
  ```

- Index unused columns cautiously: Indexing rarely queried columns increases storage and maintenance costs.

9.4 Practical Example with Results

9.4.1 Create a Table and Insert Data:

```
CREATE TABLE employees (
    emp_id INT PRIMARY KEY,
    name VARCHAR(50),
    dept_id INT,
    salary DECIMAL(10, 2),
    email VARCHAR(100)
);

INSERT INTO employees VALUES
(101, ‘Alice’, 1, 75000.00, ‘alice@example.com’),
(102, ‘Bob’, 2, 65000.00, ‘bob@example.com’),
(103, ‘Charlie’, 3, 55000.00, ‘charlie@example.com’),
(104, ‘Diana’, 1, 80000.00, ‘diana@example.com’);
```

9.4.2 Resulting Tables for the Provided SQL Statements

Emp_ID	Name	Dept_ID	Salary	Email
101	Alice	1	75000.00	alice@example.com
102	Bob	2	65000.00	bob@example.com
103	Charlie	3	55000.00	charlie@example.com
104	Diana	1	80000.00	diana@example.com

9.4.3 Create an Index and Query Efficiently:

Create an index on `dept_id` and `name`:

```
CREATE INDEX idx_dept_name ON employees(dept_id, name);
```

Query using the index:

```
SELECT name, salary FROM employees WHERE dept_id = 1 AND name = 'Alice';
```

Result:

Name	Salary
Alice	75000.00

9.4.4 Analyze Query Execution with EXPLAIN:

```
EXPLAIN SELECT name, salary FROM employees WHERE dept_id = 1 AND name = 'Alice';
```

Result:

ID	Select Type	Table	Type	Possible Keys	Key	Rows	Extra
1	SIMPLE	employees	ref	idx_dept_name	idx_dept_name	1	Using index

9.4.5 Drop an Unused Index:

Drop the `idx_email` index:

```
DROP INDEX idx_email;
```

Result: The `email` column is no longer indexed.

10 Transactions and Concurrency

Transactions and concurrency control ensure data consistency and integrity in multi-user environments.

10.1 Transactions

A transaction is a sequence of SQL operations that are executed as a single unit. Transactions ensure data consistency by following the **ACID properties**:

- **Atomicity:** All operations succeed or none are applied.
- **Consistency:** Transactions leave the database in a valid state.
- **Isolation:** Transactions do not interfere with each other.
- **Durability:** Changes persist even after a system failure.

10.1.1 Starting a Transaction (BEGIN)

Transactions begin explicitly using BEGIN.

Example:

```
BEGIN;
```

10.1.2 Committing a Transaction (COMMIT)

COMMIT saves all changes made during the transaction to the database.

Example:

```
BEGIN;
INSERT INTO accounts (account_id, balance) VALUES (1, 1000.00);
UPDATE accounts SET balance = balance - 200.00 WHERE account_id = 1;
COMMIT;
```

Explanation:

- The transaction inserts a new account and updates its balance.
- The changes are saved to the database only after COMMIT.

Result:

Account_ID	Balance
1	800.00

10.1.3 Rolling Back a Transaction (ROLLBACK)

ROLLBACK undoes all changes made during a transaction.

Example:

```
BEGIN;
INSERT INTO accounts (account_id, balance) VALUES (2, 500.00);
UPDATE accounts SET balance = balance + 300.00 WHERE account_id = 2;
ROLLBACK;
```

Explanation:

- The changes (inserting an account and updating the balance) are undone.
- The database remains unchanged.

Result: No new row is added, and existing rows remain unaffected.

10.2 Concurrency Control

Concurrency control ensures correct results when multiple transactions are executed simultaneously.

10.2.1 Locking Mechanisms

Locks prevent conflicts between transactions by restricting access to rows or tables.

- `LOCK TABLE`: Locks a table for exclusive access.
- `UNLOCK TABLES`: Releases the lock.

Example:

```
LOCK TABLE accounts WRITE;

UPDATE accounts SET balance = balance - 100.00 WHERE account_id = 1;

UNLOCK TABLES;
```

Explanation:

- The `accounts` table is locked for exclusive access during the update.
- Other transactions cannot read or write to `accounts` until the lock is released.

10.2.2 Isolation Levels

Isolation levels determine how transactions interact with each other. They control phenomena like dirty reads, non-repeatable reads, and phantom reads.

Isolation Level	Description
READ UNCOMMITTED	Transactions can read uncommitted changes (dirty reads).
READ COMMITTED	Transactions only read committed changes.
REPEATABLE READ	Ensures the same rows can be read multiple times without changes by other transactions.
SERIALIZABLE	Transactions are executed sequentially, ensuring complete isolation.

Setting Isolation Levels:

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
```

10.2.3 Examples for Isolation Levels

READ UNCOMMITTED Phenomenon: Dirty Read
– Transaction A

```
BEGIN;
UPDATE accounts SET balance = balance + 500.00 WHERE account_id = 1;
```

– Transaction B

```
SELECT balance FROM accounts WHERE account_id = 1;
```

– Reads uncommitted data

READ COMMITTED Phenomenon: Prevents Dirty Reads
– Transaction A

```
BEGIN;
UPDATE accounts SET balance = balance + 500.00 WHERE account_id = 1;
```

– Transaction B

```
SELECT balance FROM accounts WHERE account_id = 1;
```

– Reads original data until Transaction A commits

REPEATABLE READ Phenomenon: Prevents Dirty and Non-Repeatable Reads

```
-- Transaction A
BEGIN;
UPDATE accounts SET balance = balance + 500.00 WHERE account_id = 1;
```

– Transaction B

```
SELECT balance FROM accounts WHERE account_id = 1;
```

– Consistently reads the same data

SERIALIZABLE Phenomenon: Prevents Dirty Reads, Non-Repeatable Reads, and Phantom Reads

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

BEGIN;
SELECT SUM(balance) FROM accounts WHERE balance > 1000.00;
```

– Ensures no other transactions interfere

10.3 Practical Example with Results

10.3.1 Create and Populate a Table:

```
CREATE TABLE accounts (
    account_id INT PRIMARY KEY,
    balance DECIMAL(10, 2)
);

INSERT INTO accounts VALUES
(1, 1000.00),
(2, 500.00),
(3, 1500.00);
```

10.3.2 Resulting Tables for the Provided SQL Statements

Account_ID	Balance
1	1000.00
2	500.00
3	1500.00

10.3.3 Perform a Transaction: Transfer $200 from account 1 to account 2:

```
BEGIN;

UPDATE accounts SET balance = balance - 200.00 WHERE account_id = 1;
UPDATE accounts SET balance = balance + 200.00 WHERE account_id = 2;

COMMIT;
```

Result:

Account_ID	**Balance**
1	800.00
2	700.00
3	1500.00

10.3.4 Locking Mechanisms: Prevent other transactions from modifying accounts during an update:

```
LOCK TABLE accounts WRITE;

UPDATE accounts SET balance = balance - 100.00 WHERE account_id = 3;

UNLOCK TABLES;
```

11 Advanced Features in SQL

SQL provides advanced features like views, stored procedures, and functions to enhance modularity, reusability, and abstraction in database operations.

11.1 Views

A view is a virtual table based on a SQL query. Views do not store data themselves; they retrieve data dynamically from the underlying tables.

11.1.1 Creating Views (CREATE VIEW)

Example: Create a view high_salary_employees that displays employees earning more than $70,000.

```
CREATE VIEW high_salary_employees AS
SELECT emp_id, name, salary
FROM employees
WHERE salary > 70000.00;
```

Explanation:

- The view dynamically retrieves data where salary > 70,000.

Usage:

```
SELECT * FROM high_salary_employees;
```

Result:

Emp_ID	**Name**	**Salary**
104	Diana	80000.00

11.1.2 Modifying Views (CREATE OR REPLACE VIEW)

Example: Modify the high_salary_employees view to include department information.

```
CREATE OR REPLACE VIEW high_salary_employees AS
SELECT e.emp_id, e.name, e.salary, d.dept_name
FROM employees e
JOIN departments d ON e.dept_id = d.dept_id
WHERE e.salary > 70000.00;
```

Usage:

```
SELECT * FROM high_salary_employees;
```

Result:

Emp_ID	**Name**	**Salary**	**Dept_Name**
104	Diana	80000.00	HR

11.1.3 Dropping Views (DROP VIEW)

Drop a view using DROP VIEW.

Example:

```
DROP VIEW high_salary_employees;
```

Explanation: The high_salary_employees view is deleted from the database.

11.2 Stored Procedures and Functions

Stored procedures and user-defined functions encapsulate reusable SQL logic.

11.2.1 Stored Procedures

A stored procedure is a set of SQL statements that can be executed as a single callable unit.

Creating Stored Procedures Example: Create a stored procedure `transfer_funds` to transfer money between accounts.

```
DELIMITER $$

CREATE PROCEDURE transfer_funds(
    sender_id INT,
    receiver_id INT,
    amount DECIMAL(10, 2)
)
BEGIN
    UPDATE accounts SET balance = balance - amount WHERE account_id = sender_id;
    UPDATE accounts SET balance = balance + amount WHERE account_id = receiver_id;
END $$

DELIMITER ;
```

Explanation:

- The procedure deducts `amount` from `sender_id` and adds it to `receiver_id`.

Calling Stored Procedures Example: Transfer $100 from account 1 to account 2.

```
CALL transfer_funds(1, 2, 100.00);
```

Result:

Account_ID	Balance
1	700.00
2	800.00
3	1500.00

11.2.2 User-Defined Functions (UDFs)

A user-defined function (UDF) returns a value based on input parameters. Functions are used in queries and calculations.

Creating Functions Example: Create a function `calculate_bonus` that calculates a 10% bonus based on the salary.

```
DELIMITER $$

CREATE FUNCTION calculate_bonus(salary DECIMAL(10, 2))
RETURNS DECIMAL(10, 2)
DETERMINISTIC
BEGIN
    RETURN salary * 0.10;
END $$

DELIMITER ;
```

Explanation: The function takes `salary` as input and returns 10% of it.

Using Functions Example: Calculate bonuses for all employees.

```
SELECT name, calculate_bonus(salary) AS bonus
FROM employees;
```

Result:

Name	Bonus
Alice	7875.00
Bob	6825.00
Charlie	5500.00
Diana	8000.00

11.3 Practical Example with Results

11.3.1 Create and Populate Tables

```
CREATE TABLE employees (
    emp_id INT PRIMARY KEY,
    name VARCHAR(50),
    dept_id INT,
    salary DECIMAL(10, 2)
);

CREATE TABLE departments (
    dept_id INT PRIMARY KEY,
    dept_name VARCHAR(50)
);

INSERT INTO departments VALUES
(1, ‘HR’),
(2, ‘Engineering’),
(3, ‘Marketing’);

INSERT INTO employees VALUES
(101, ‘Alice’, 1, 75000.00),
(102, ‘Bob’, 2, 65000.00),
(103, ‘Charlie’, 3, 55000.00),
(104, ‘Diana’, 1, 80000.00);
```

11.3.2 Resulting Tables for the Provided SQL Statements

Departments Table

Dept_ID	Dept_Name
1	HR
2	Engineering
3	Marketing

Employees Table

Emp_ID	Name	Dept_ID	Salary
101	Alice	1	75000.00
102	Bob	2	65000.00
103	Charlie	3	55000.00
104	Diana	1	80000.00

11.3.3 Create and Query a View

Example: Create a view `engineering_employees` for employees in the Engineering department.

```
CREATE VIEW engineering_employees AS
SELECT name, salary
FROM employees
WHERE dept_id = 2;
```

Query the View:

```
SELECT * FROM engineering_employees;
```

Result:

Name	Salary
Bob	65000.00

11.3.4 Create and Call a Stored Procedure

Example: Transfer $200 between accounts.

```
CALL transfer_funds(101, 103, 200.00);
```

Result:

Emp_ID	Name	Salary
101	Alice	74800.00
103	Charlie	55200.00

11.3.5 Use a Function in a Query

Example: Calculate bonuses for employees.

```
SELECT name, calculate_bonus(salary) AS bonus
FROM employees;
```

Result:

Name	Bonus
Alice	7480.00
Bob	6500.00
Charlie	5520.00
Diana	8000.00