What is SQL?

SQL stands for Structured Query Language. **It is a standard programming language used for managing and manipulating relational databases**. SQL allows users to perform various operations on databases, such as querying data to retrieve specific information, updating data, inserting new data, and deleting data. It is widely used in database management systems (**DBMS**) like **MySQL**, **PostgreSQL**, **SQL Server**, **OracleDB**, and others.

SQL is essential for anyone working with databases, including developers, data analysts, database administrators, and data scientists.

Feature	MySQL	SQL
Definition	MySQL is an open-source relational database management system (RDBMS).	SQL (Structured Query Language) is a standard language for managing and manipulating relational databases.
Type	Database management system (DBMS)	Query language used to communicate with relational databases.
Usage	Used to store and manage databases.	Used to perform operations such as querying, inserting, updating, and deleting data in a database.
Flexibility	Offers both free and enterprise editions.	SQL is universally used by all relational databases like MySQL, SQL Server, PostgreSQL, etc.
Vendor	Developed and maintained by Oracle Corporation.	Not a product; a language standard defined by ANSI (American National Standards Institute).
Primary Function	Manages databases using SQL queries.	Provides commands like SELECT, INSERT, UPDATE, DELETE for interacting with databases.
Platform	Runs on Linux, Windows, macOS, and other operating systems.	SQL is used across different relational database platforms.
ACID Compliance	Fully ACID-compliant for transaction management.	SQL itself is not ACID-compliant; it depends on the RDBMS implementation.
Support for Foreign Keys	Yes, supports foreign key constraints for relational integrity.	SQL supports foreign keys, but enforcement depends on the RDBMS used.
Examples	MySQL, MariaDB, Percona	PostgreSQL, Oracle, SQL Server, MySQL, SQLite, etc.

Summary:

- **MySQL** is a **relational database management system** that uses **SQL** to manage databases.
- **SQL** is a language standard used across all relational database systems for querying and managing data.

7 Reasons Why You Should Learn SQL :

1. **A Programming Language Not Only Limited to Programming Tasks**

   SQL is not just for developers. It is widely used by professionals in marketing, sales, and finance to query and analyze data for insights. This versatility makes it a valuable skill across domains.

2. **Almost Every Data Technology Supports SQL or a Flavor of It**

   SQL serves as the foundation for many data technologies. Whether you're working with databases or advanced data platforms, SQL is a universal tool that integrates seamlessly with most systems.

3. **Helpful in Handling Huge Amounts of Structured Data**

   SQL allows efficient storage, retrieval, and manipulation of large datasets, saving time and effort in managing data-heavy operations.

4. **A Useful Tool for Data Analytics**

   SQL is essential for data analysis, enabling professionals to process, filter, and interpret data to make strategic decisions and derive actionable insights.

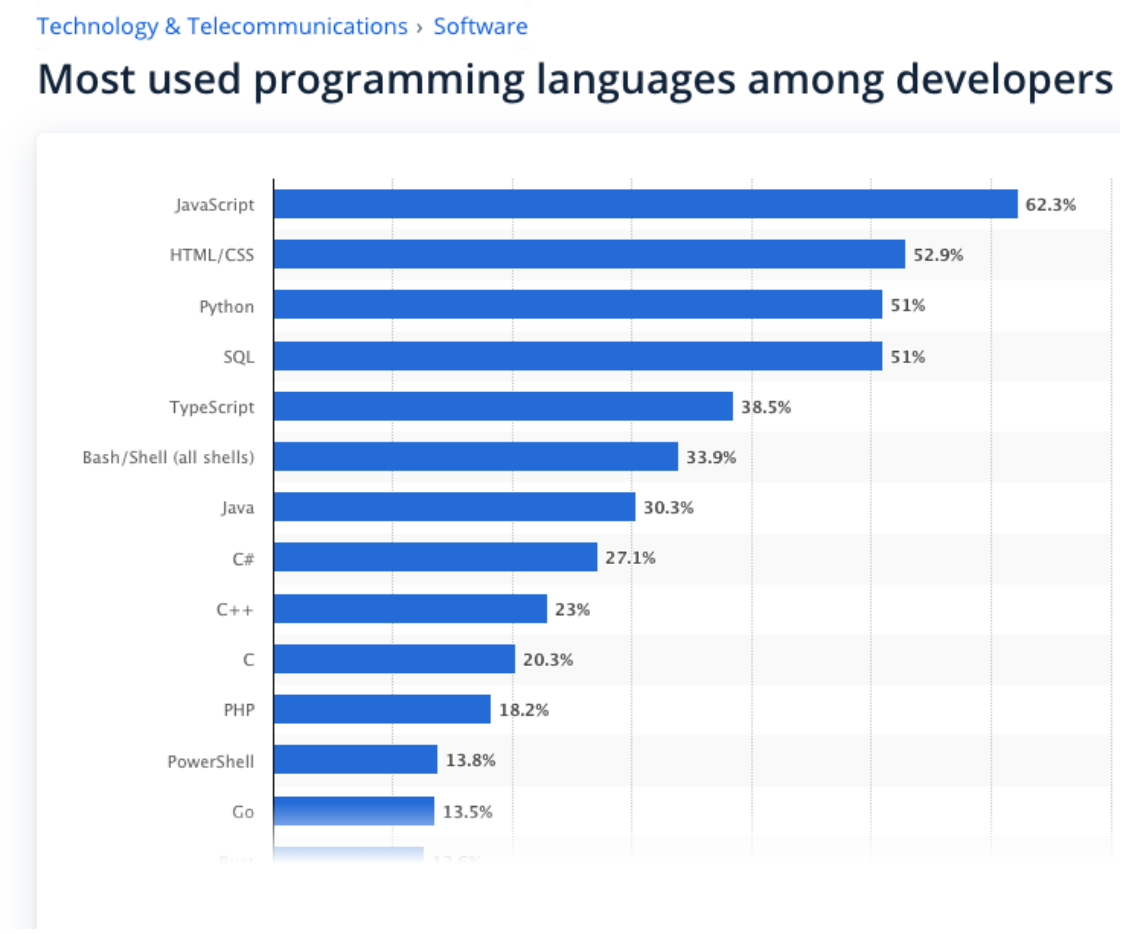
5. **The Cherry on a Cake for Data Mining**

6. **Helpful in Pursuing Data Science as a Career**

7. **In-Demand Skill**

   SQL remains one of the most sought-after skills across industries. Its simplicity, robustness, and versatility make it a staple for developers, analysts, and managers ali

MOST POPULAR LANGUAGE :



WHY MYSQL :

Top 5 Reasons to Learn MySQL

1. **Widely Used and Industry Standard**

   MySQL is one of the most popular relational database management systems, trusted by top companies for its reliability and scalability.

2. **Free and Open Source**

   MySQL offers a free, open-source version, making it accessible for learners and businesses without significant investment.

3. **High Performance for Data Management**

   It handles large-scale databases efficiently, making it ideal for web applications, data analysis, and large enterprises.

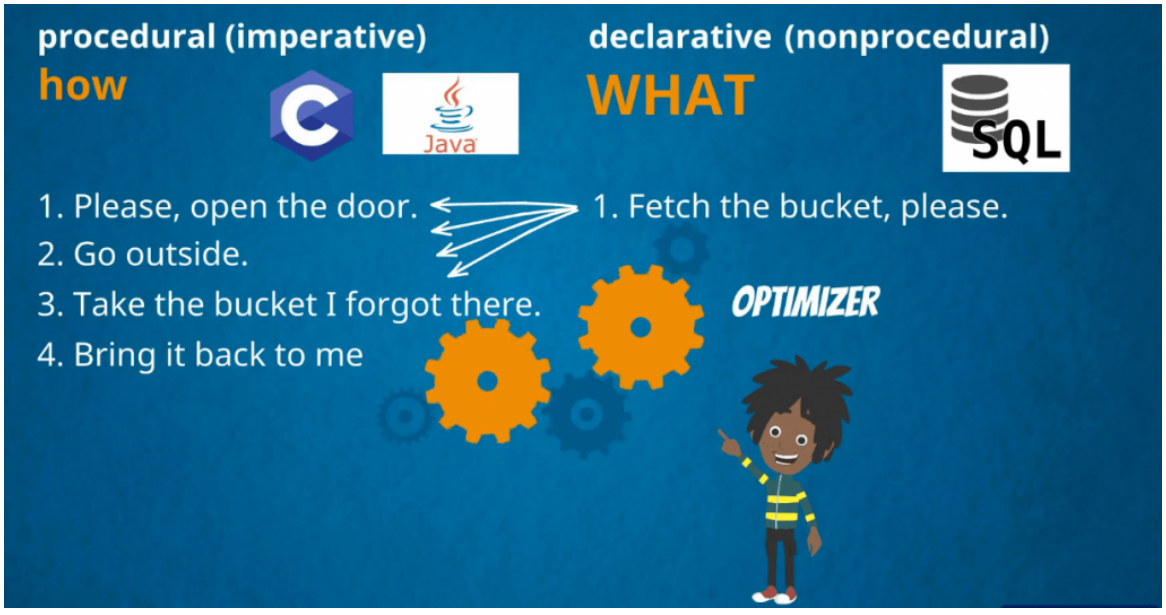
4. **Cross-Platform Compatibility**

   MySQL supports multiple operating systems and integrates seamlessly with various programming languages and tools.

5. **Strong Community and Documentation**

MySQL has extensive documentation and a supportive community, ensuring resources are readily available for troubleshooting and learning.

5 . SQL as a declarative language



Why is SQL Non-Procedural?

In **non-procedural** programming languages, you specify **what** you want to do (the desired outcome) rather than **how** to do it (the steps to achieve that outcome). SQL is considered non-procedural because:

- You define the **result** (e.g., retrieve data, update data) without specifying the steps or procedures the system should follow to get that result.
- For example, an SQL query like `SELECT * FROM users WHERE age > 30;` simply asks for the data you want (all users older than 30) without defining the specific operations that should occur to retrieve that data.

Procedural vs Non-Procedural:

Feature	Procedural Programming	Non-Procedural Programming
Focus	Focuses on how to achieve the desired outcome (step-by-step).	Focuses on what result you want to achieve, not how to do it.
Control	The programmer defines the sequence of operations to be performed.	The programmer specifies what needs to be done, and the system decides how to perform it.
Example Languages	C, Java, Python, Fortran, etc.	SQL, HTML, CSS, etc.
Example	In procedural programming, you would write a loop to search through records to find certain data.	In SQL, you would write a query like `SELECT * FROM users WHERE age > 30;` to get the data.

INTRODUCTION TO DATABASE :

Table: customers

customer_id	first_name	last_name	phone	country
1	John	Doe	817-646-8833	USA
2	Robert	Luna	412-862-0502	USA
3	David	Robinson	208-340-7906	UK
4	John	Reinhardt	307-242-6285	UK
5	Betty	Taylor	806-749-2958	UAE

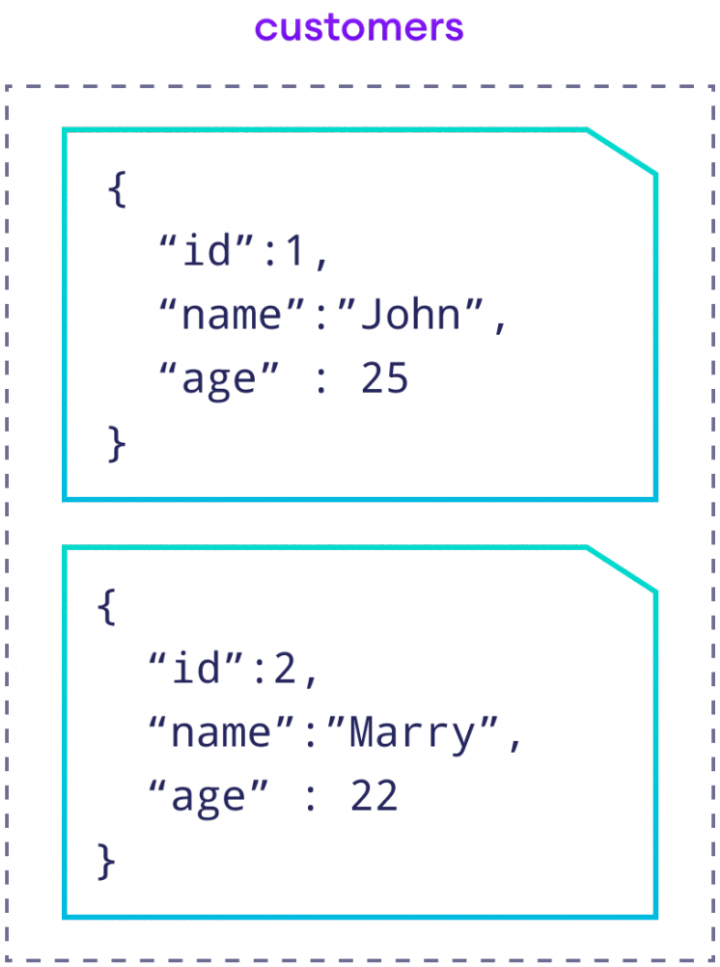


Table: orders

order_id	product	total	customer_id
1	Paper	500	5
2	Pen	10	2
3	Marker	120	3
4	Books	1000	1
5	Erasers	20	4

Table: customers

customer_id	first_name	last_name	phone	country
1	John	Doe	817-646-8833	USA
2	Robert	Luna	412-862-0502	USA
3	David	Robinson	208-340-7906	UK
4	John	Reinhardt	307-242-6285	UK
5	Betty	Doe	806-749-2958	UAE

What are Databases?

Databases are collections of information stored on digital devices, including mobile phones, digital watches, electronic tablets, and computers.

Why are they called "Relational" Databases?

The term **"relational"** comes from the concept of **relations** in mathematics. In relational databases:

- **Data is organized into tables** (also called relations).
- Each table consists of **rows** (called tuples) and **columns** (called attributes).
- Tables are related to one another through **keys**, such as primary keys and foreign keys.

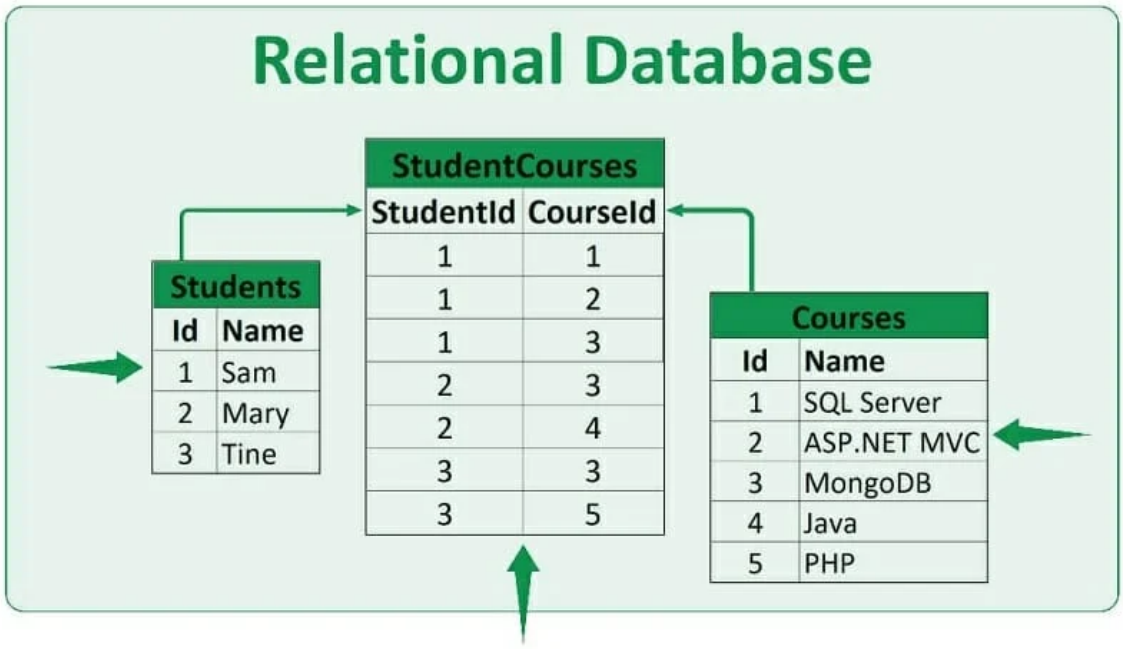
Relational Database

A relational database houses other major types of databases that are the opposite of NoSQL databases. Information is well structured and arranged in relational databases. It is based on the relational data model.

The relational model uses tables that relate to one another using two critical columns: primary key and foreign key. Relational databases focus on a model that stores data in rows and columns, collectively forming a table.

Key Features of Relational Databases:

- **Tables (Relations)**: Data is stored in tables, which are related to each other through keys.
- **SQL (Structured Query Language)**: A language used to interact with the database, retrieve data, and perform operations.
- **ACID Properties**: Ensures reliable transaction management (Atomicity, Consistency, Isolation, Durability).



There are four common properties of relational databases.

- **Atomicity:** This feature takes the ‘all or nothing’ strategy; every data operation will be completed regardless of its outcome. Atomicity means no neutral stance; it is either success or failure.
- **Consistency:** The value of data before and after every operation should remain the same. For instance, the account statement before and after transactions should resonate.
- **Isolation:** Data in relational databases should remain isolated because you have concurrent users accessing the database from different locations. When there are multiple operations, the after-effect of one does not affect other operations in the database.
- **Durability:** Durability ensures that after every operation, the state of data sets remains permanent.

Pros

- Speed and accuracy
- Security
- Simplicity
- Accessibility
- Multi-user feature
- Easy modification of database entries

Cons

- Complexity
- High costs
- Physical storage
- Information loss

9. Most Commonly Used Objects in Relational Databases :

What is Meant by "Object" in SQL?

In the context of SQL and relational databases, **objects** are the components or entities that you create, manage, and interact with within a database. They represent the **building blocks of a database system** and are used to store, organize, and manipulate data.

Key Features of Database Objects:

1. **Purpose**: Objects are used to store, retrieve, modify, or organize data.

2. **Type**: Different types of objects serve specific purposes (e.g., storing data, optimizing queries, automating tasks).
3. **Interaction**: SQL commands operate on these objects to perform database operations.

Examples of Objects in SQL

Here's a breakdown of commonly used SQL objects:

1. Tables

- **Description**: A structured collection of data stored in rows and columns.
- **Purpose**: Main storage object for data in a relational database.
- **Example**:

```
sql
Copy code
CREATE TABLE employees (
    id INT PRIMARY KEY,
    name VARCHAR(50),
    age INT
);
```

2. Views

- **Description**: A virtual table based on a SQL query.
- **Purpose**: Simplifies complex queries and provides a security layer by restricting access to specific data.
- **Example**:

```
sql
Copy code
CREATE VIEW active_employees AS
SELECT id, name FROM employees WHERE active = 1;
```

3. Indexes

- **Description**: Data structures that improve the speed of data retrieval.
- **Purpose**: Optimize query performance by creating quick access paths for columns.
- **Example**:

```
sql
Copy code
CREATE INDEX idx_name ON employees (name);
```

4. Keys

- **Primary Key**: Uniquely identifies each row in a table.
- **Foreign Key**: Links tables together by referencing a primary key in another table.
- **Example**:

```
sql
Copy code
ALTER TABLE orders ADD FOREIGN KEY (customer_id) REFERENCES customers(i
d);
```

5. Constraints

- **Description**: Rules applied to table columns to ensure data integrity.
- **Purpose**: Prevent invalid or duplicate data entry.
- **Example**:

```
sql
Copy code
CREATE TABLE students (
    id INT PRIMARY KEY,
    name VARCHAR(50) NOT NULL,
    email VARCHAR(100) UNIQUE
);
```

6. Stored Procedures

- **Description**: Predefined SQL code that can be executed repeatedly.
- **Purpose**: Automates repetitive tasks or operations.
- **Example**:

```
sql
Copy code
CREATE PROCEDURE GetEmployeeCount()
```

```
AS
BEGIN
    SELECT COUNT(*) FROM employees;
END;
```

7. Triggers

- **Description**: Automated actions executed when specific events occur on a table.
- **Purpose**: Enforce business rules or maintain audit logs.
- **Example**:

```
sql
Copy code
CREATE TRIGGER log_update
AFTER UPDATE ON employees
FOR EACH ROW
INSERT INTO audit_log (employee_id, change_date) VALUES (OLD.id, NOW());
```

8. Functions

- **Description**: SQL code that performs a calculation or operation and returns a value.
- **Purpose**: Encapsulates reusable logic.
- **Example**:

```
sql
Copy code
CREATE FUNCTION CalculateAge(birthdate DATE)
RETURNS INT
AS
BEGIN
    RETURN YEAR(CURDATE()) - YEAR(birthdate);
END;
```

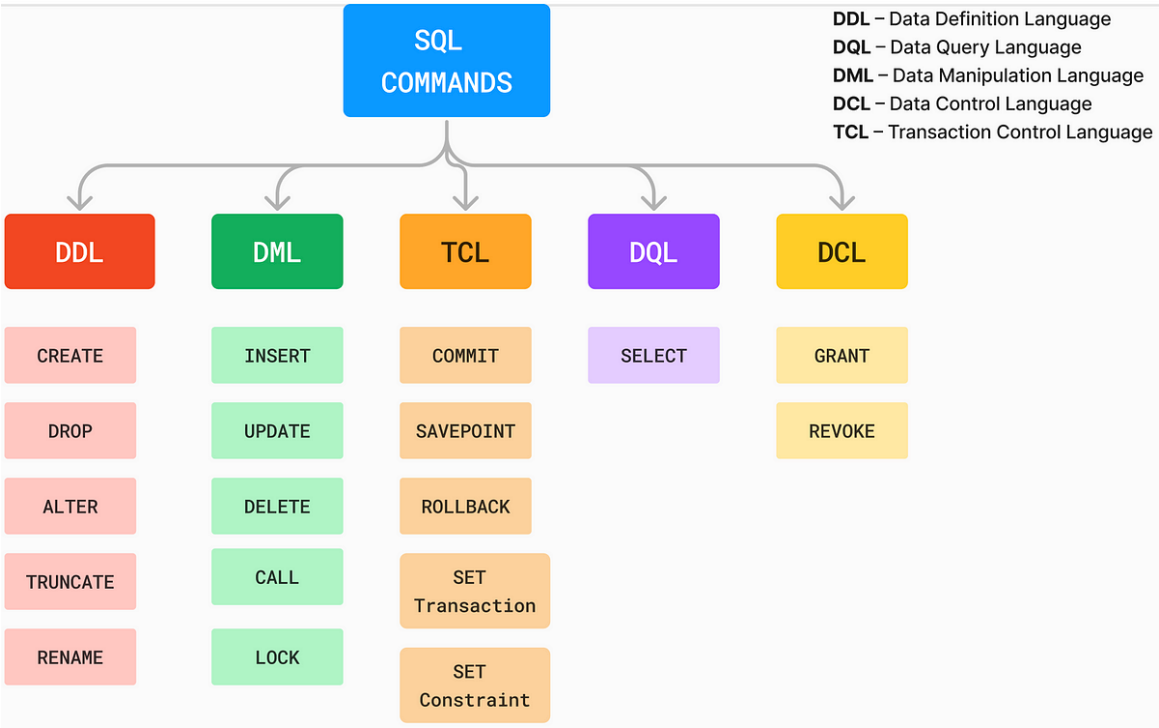
Mapping Commands to Objects

These commands can be applied to various objects like **tables**, **views**, **indexes**, and **constraints**:

1. **Tables**:
    - `CREATE TABLE`, `ALTER TABLE`, `DROP TABLE`, `INSERT`, `UPDATE`, `DELETE`, `SELECT`.

2. **Views**:
   - `CREATE VIEW`, `DROP VIEW`, `SELECT`.
3. **Indexes**:
   - `CREATE INDEX`, `DROP INDEX`.
4. **Keys and Constraints**:
   - `ALTER TABLE` (to add or modify keys/constraints), `DROP` (to remove them).
5. **Stored Procedures and Functions**:
   - `CREATE PROCEDURE`, `DROP PROCEDURE`, `CALL`.
6. **Triggers**:
   - `CREATE TRIGGER`, `DROP TRIGGER`.

6. SQL Syntax /COMMANDS :

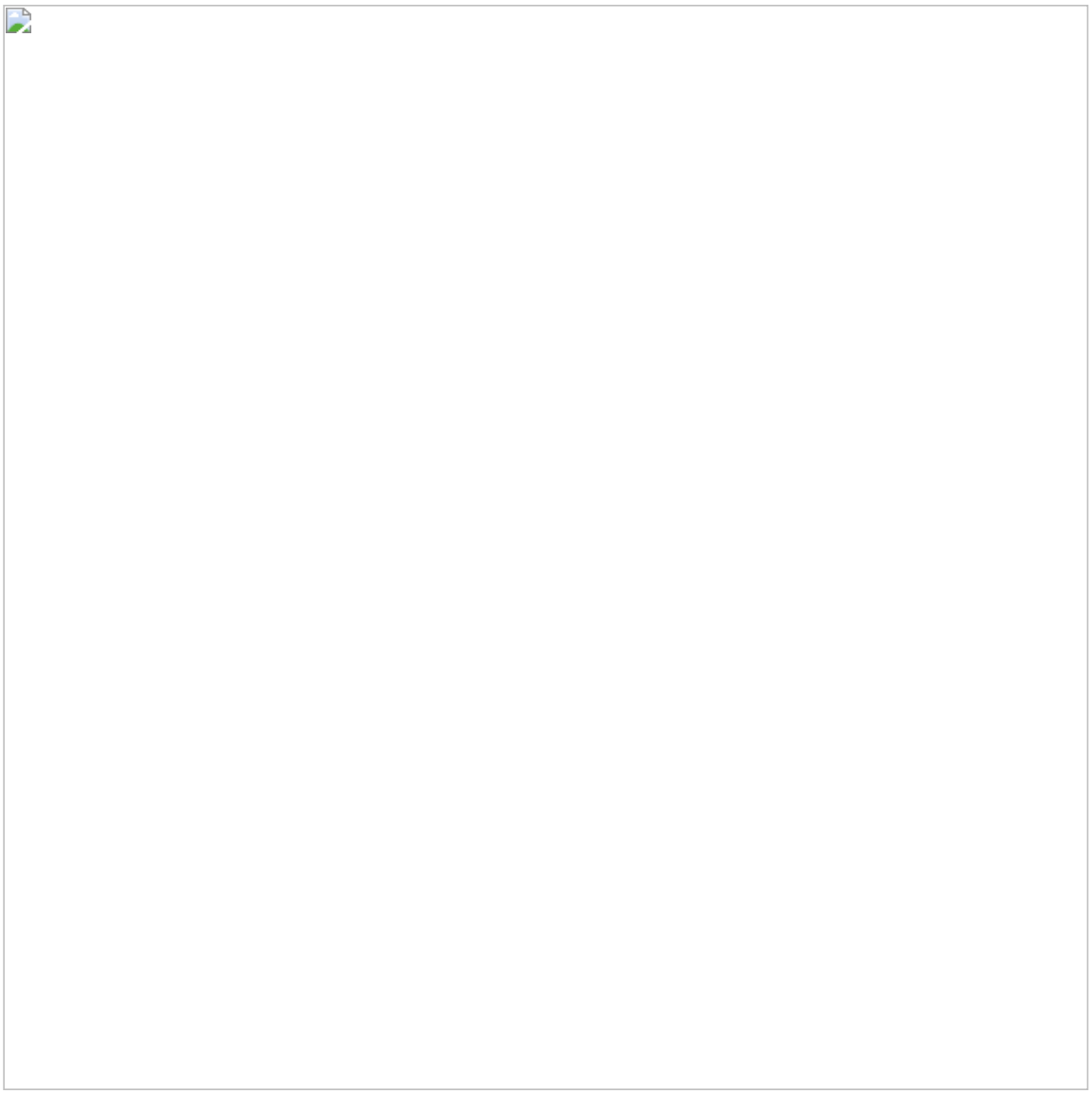


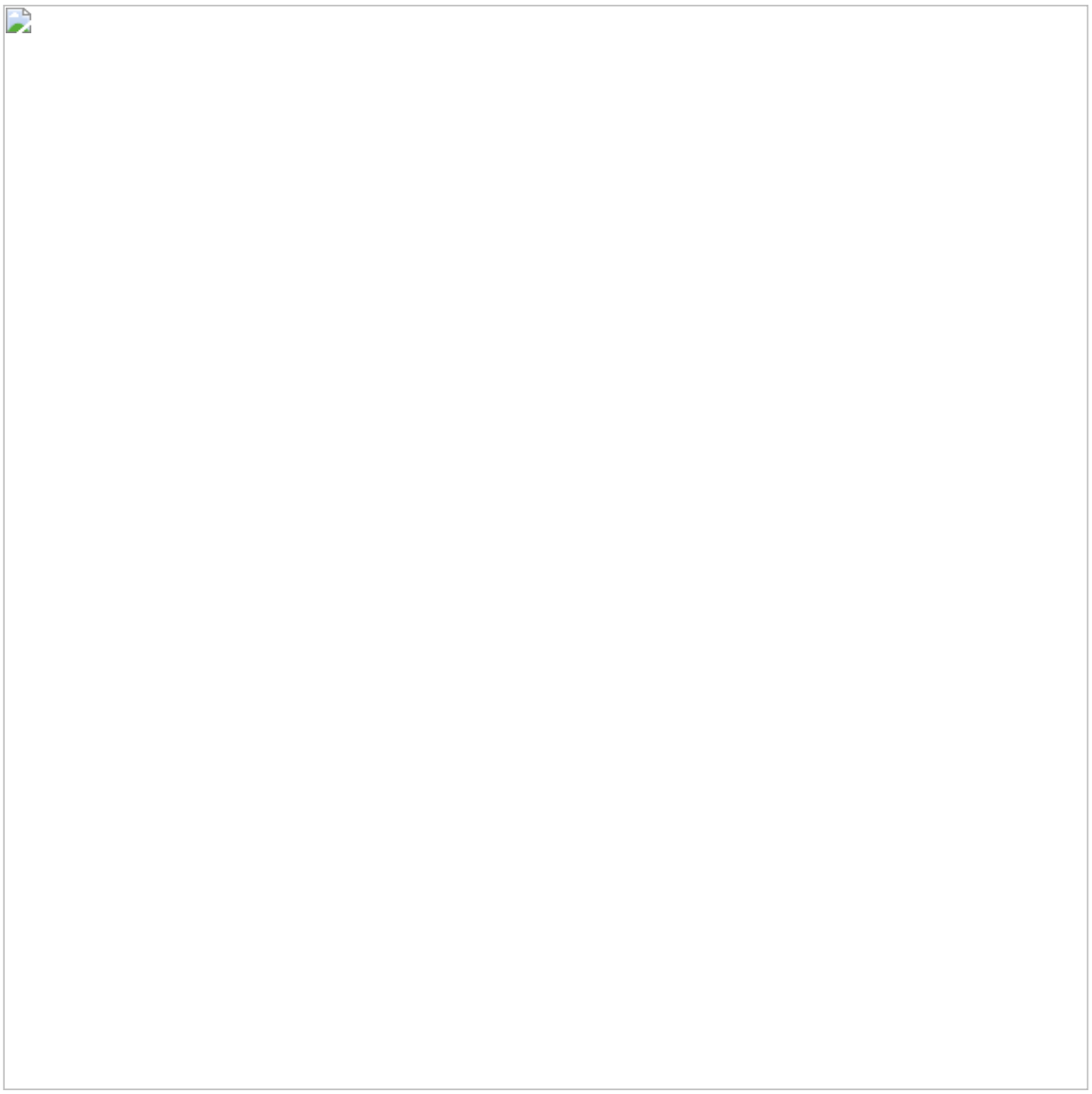
7 . Data definition language (DDL) :

The Data Definition Language (DDL) encompasses a series of SQL commands specifically aimed at designing and shaping the structure of a database. Its primary function is to define and manage the schema, or blueprint, of database objects. DDL includes SQL statements that are responsible for creating, altering, and removing database structures, distinct from the data they contain. These commands are generally reserved for database administrators or developers, as they directly impact the database's architecture. DDL commands are:

1. **CREATE:** Utilized for the creation of databases and various database objects such as tables, views, indexes, stored procedures, functions, and triggers.
2. **DROP:** Employed to remove objects from a database. Deleting tables, views, indexes, or other database structures when they are no longer needed.
3. **ALTER:** Used to modify the design or structure of an existing database. Changing attributes of database elements, like altering table schemas or modifying column properties.
4. **TRUNCATE:** Employed to delete all records from a table, including freeing up space allocated for those records. Quickly remove all data from a table while retaining the table structure for future use.
5. **COMMENT:** Used for adding comments to the data dictionary. Documenting the database by adding descriptive text to database objects, enhancing readability and maintainability.
6. **RENAME:** Utilized for changing the name of an existing database object. Updating names of database elements to better reflect their purpose or to comply with naming conventions.

Below is a table that outlines the key Data Definition Language (DDL) commands in SQL, along with their syntax and examples:





8.Data Query Language (DQL)

DQL is a subset of SQL focusing on retrieving data from databases using the **SELECT** statement. It is crucial for data analysis, helping retrieve and organize data from one or more tables.

Example 1: Retrieve All Records from a Table

```
SELECT * FROM employees;
```

This command retrieves all columns for every record in the `employees` table.

Example 2: Retrieve Specific Columns

```
SELECT name, department FROM employees;
```

This fetches only the `name` and `department` columns from the `employees` table.

Example 3: Retrieve Data with a Condition

```
SELECT name FROM employees WHERE department = 'Sales';
```

This retrieves the `name` of employees working in the **Sales** department.

Example 4: Sort Retrieved Data

```
SELECT name, salary FROM employees ORDER BY salary DESC;
```

This retrieves the `name` and `salary` of all employees, sorted in descending order by salary.

9. Data Manipulation Language (DML) Commands

SQL commands that focus on modifying or handling data within a database fall under the category of Data Manipulation Language (DML). This segment encompasses a significant portion of SQL operations. DML commands are:

1. **INSERT**

- **Purpose:** Adds new rows to a table.
- **Syntax:**

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

- **Example:**

```
INSERT INTO employees (name, department) VALUES ('John Doe', 'Marketin
g');
```

2. **UPDATE**
   - **Purpose:** Modifies existing data in a table.
   - **Syntax:**

```
UPDATE table_name SET column1 = value1 WHERE condition;
```

   - **Example:**

```
UPDATE employees SET department = 'HR' WHERE id = 123;
```

3. **DELETE**
   - **Purpose:** Removes rows from a table based on a condition.
   - **Syntax:**

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

   - **Example:**

```
DELETE FROM employees WHERE department = 'Intern';
```

4. **MERGE**
    - **Purpose:** Inserts, updates, or deletes records based on a match between two tables.
    - **Syntax:**

```
MERGE INTO target_table USING source_table
ON condition
WHEN MATCHED THEN UPDATE SET column1 = value1
WHEN NOT MATCHED THEN INSERT (column1, column2) VALUES (value1, value
2);
```

    - **Example:**

```
MERGE INTO employees USING temp_employees
ON employees.id = temp_employees.id
WHEN MATCHED THEN UPDATE SET employees.salary = temp_employees.salary
WHEN NOT MATCHED THEN INSERT (id, name, salary) VALUES (temp_employee
s.id, temp_employees.name, temp_employees.salary);
```

5. **CALL**
    - **Purpose:** Executes a stored procedure.
    - **Example:**

```
CALL UpdateEmployeeSalary(101, 5000);
```

6. **LOCK TABLE**
    - **Purpose:** Restricts access to a table to ensure data consistency.
    - **Example:**

```
LOCK TABLE employees IN EXCLUSIVE MODE;
```

10 . Data Control Language (DCL) Examples :

Data Control Language (DCL) is a subset of SQL commands used to control access to data in a database. DCL is primarily concerned with permissions and deals with the rights and privileges of

the database users. The main commands in DCL are GRANT and REVOKE.

1. **GRANT**
   - **Purpose:** Provides specific privileges to a user or role.
   - **Example 1:** Granting the SELECT privilege to a user.

     ```
     GRANT SELECT ON employees TO user123;
     ```

   - **Example 2:** Granting multiple privileges (SELECT and UPDATE).

     ```
     GRANT SELECT, UPDATE ON employees TO user123;
     ```

2. **REVOKE**
   - **Purpose:** Removes specific privileges from a user or role.
   - **Example 1:** Revoking the SELECT privilege.

     ```
     REVOKE SELECT ON employees FROM user123;
     ```

   - **Example 2:** Revoking all privileges from a user.

     ```
     REVOKE ALL PRIVILEGES ON employees FROM user123;
     ```

---

11 .Transaction Control Language (TCL) Examples :

Transaction Control Language (TCL) in SQL is a set of commands specifically designed to manage the changes made by DML statements (like INSERT, UPDATE, DELETE). It allows control over transaction processes in a database:

1. **COMMIT**
   - **Purpose:** Permanently saves changes made during the current transaction.
   - **Example:**

     ```
     UPDATE employees SET salary = salary + 1000 WHERE id = 101;
     ```

```
COMMIT;
```

2. **ROLLBACK**
   - **Purpose:** Reverts changes made during the current transaction to the last committed state.
   - **Example:**

```
UPDATE employees SET salary = salary + 1000 WHERE id = 101;
ROLLBACK;
```

3. **SAVEPOINT**
   - **Purpose:** Creates a point within a transaction to which you can later roll back.
   - **Example:**

```
UPDATE employees SET salary = salary + 500 WHERE id = 102;
SAVEPOINT savepoint1;
UPDATE employees SET salary = salary + 1000 WHERE id = 103;
ROLLBACK TO savepoint1;
```

4. **SET TRANSACTION**
   - **Purpose:** Specifies transaction properties like isolation level.
   - **Example:**

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
BEGIN TRANSACTION;
-- Transaction statements go here
COMMIT;
```

5. **SET CONSTRAINTS**
   - **Purpose:** Manages the timing of constraint enforcement during transactions.
   - **Example:**

```
SET CONSTRAINTS ALL DEFERRED;
```

```
BEGIN TRANSACTION;
INSERT INTO orders (order_id, customer_id) VALUES (101, 'C01');
INSERT INTO customers (customer_id, name) VALUES ('C01', 'John Doe');
COMMIT;
```

Keywords in SQL :

Keywords in SQL are reserved words that have a predefined meaning and are used to perfom specific operations in SQL queries. They cannot be used as identifiers such as table names, column names, or other database objects. These keywords are the building blocks of SQL queries.

Common SQL Keywords with Examples:

1. **SELECT**
   - **Purpose:** Used to retrieve data from a database.
   - **Example:**This query retrieves the `name` and `age` columns from the `employees` table.

     ```
     SELECT name, age FROM employees;
     ```

2. **FROM**
   - **Purpose:** Specifies the table from which to retrieve data.
   - **Example:**Retrieves all columns from the `employees` table.

     ```
     SELECT * FROM employees;
     ```

3. **WHERE**
   - **Purpose:** Filters records based on a specified condition.
   - **Example:**Retrieves names of employees working in the HR department.

     ```
     SELECT name FROM employees WHERE department = 'HR';
     ```

4. **INSERT INTO**
   - **Purpose:** Adds new rows of data to a table.

- **Example:**Adds a new employee named Alice to the Finance department.

```
INSERT INTO employees (name, department) VALUES ('Alice', 'Finance');
```

5. **UPDATE**
   - **Purpose:** Modifies existing data in a table.
   - **Example:**Updates Alice's salary to 50,000.

```
UPDATE employees SET salary = 50000 WHERE name = 'Alice';
```

6. **DELETE**
   - **Purpose:** Removes data from a table.
   - **Example:**Deletes all employees in the Intern department.

```
DELETE FROM employees WHERE department = 'Intern';
```

7. **CREATE TABLE**
   - **Purpose:** Creates a new table in the database.
   - **Example:**Creates an `employees` table with three columns.

```
CREATE TABLE employees (
    id INT PRIMARY KEY,
    name VARCHAR(100),
    department VARCHAR(50)
);
```

8. **DROP TABLE**
   - **Purpose:** Deletes a table from the database.
   - **Example:**Deletes the `employees` table.

```
DROP TABLE employees;
```

```

```

9. **JOIN**
   - **Purpose:** Combines rows from two or more tables based on a related column.
   - **Example:**Retrieves employee names along with their department names.

```
SELECT employees.name, departments.department_name
FROM employees
JOIN departments ON employees.department_id = departments.id;
```

10. **ORDER BY**
    - **Purpose:** Sorts the results of a query in ascending or descending order.
    - **Example:**Retrieves employee names and salaries, sorted by salary in descending order.

```
SELECT name, salary FROM employees ORDER BY salary DESC;
```

11. **GROUP BY**
    - **Purpose:** Groups rows sharing a property into summary rows.
    - **Example:**Groups employees by department and counts the number of employees in each.

```
SELECT department, COUNT(*) AS num_employees
FROM employees
GROUP BY department;
```

12. **HAVING**
    - **Purpose:** Filters records after a `GROUP BY` operation.
    - **Example:**Displays departments with more than 5 employees.

```
SELECT department, COUNT(*) AS num_employees
FROM employees
GROUP BY department
```

```
HAVING COUNT(*) > 5;
```

13. **ALTER TABLE**
    - **Purpose:** Modifies the structure of an existing table.
    - **Example:**Adds a new `email` column to the `employees` table.

```
ALTER TABLE employees ADD COLUMN email VARCHAR(100);
```

14. **DISTINCT**
    - **Purpose:** Ensures that duplicate rows are excluded from the results.
    - **Example:**Retrieves unique department names from the `employees` table.

```
SELECT DISTINCT department FROM employees;
```

15. **LIMIT**
    - **Purpose:** Restricts the number of rows returned by a query.
    - **Example:**Retrieves the first 5 employee names.

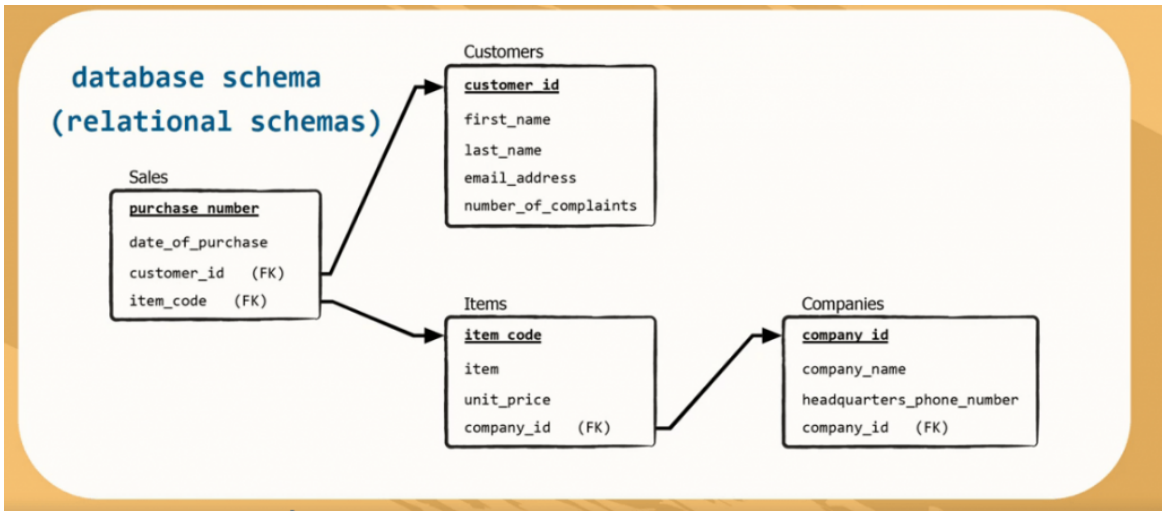
```
SELECT name FROM employees LIMIT 5;
```

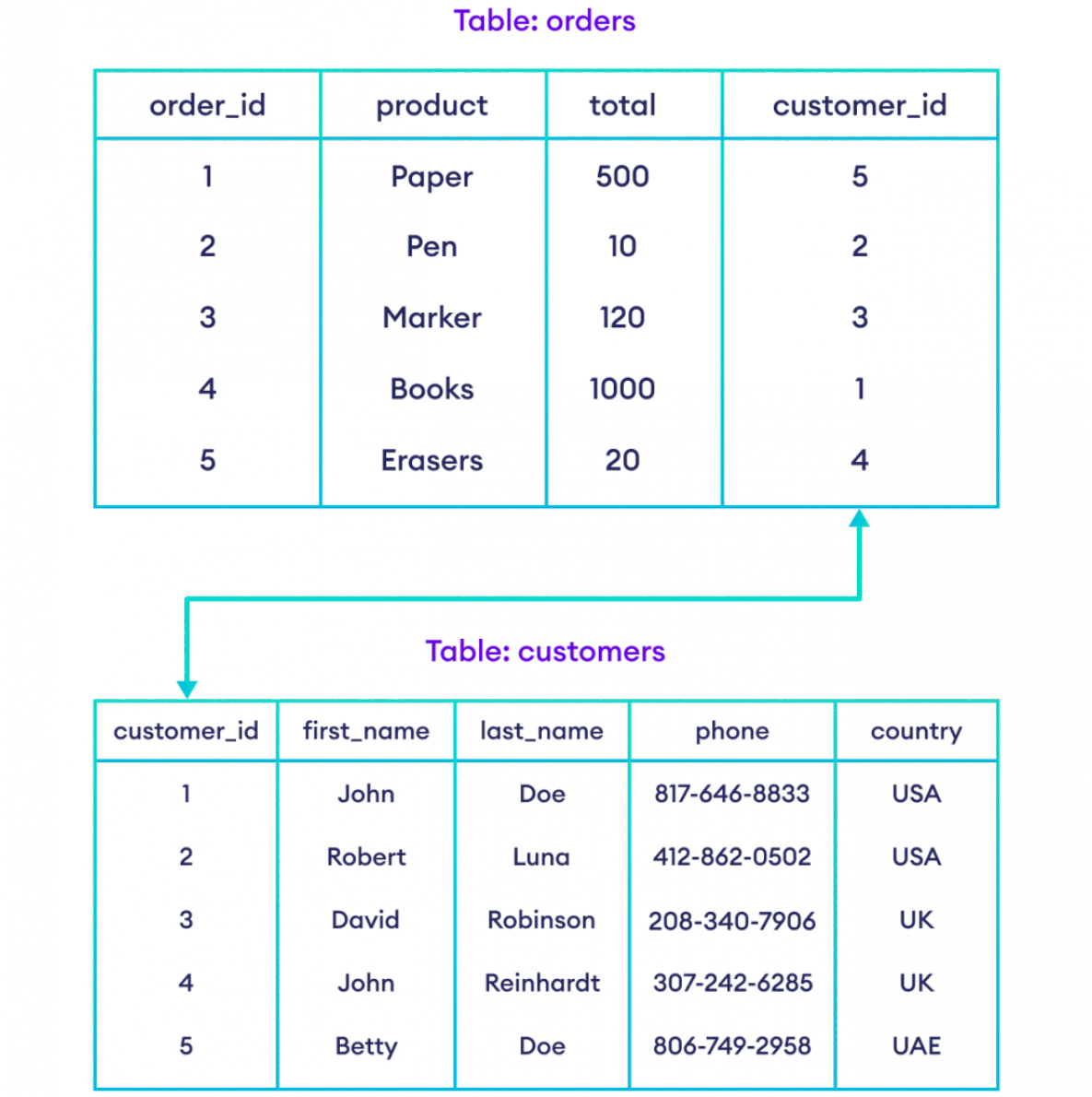
12. Differences Between Databases and Spreadsheets

Feature	Databases	Spreadsheets
Data Structure	Relational, structured tables	Flat, basic rows and columns
Scalability	High	Limited
Data Integrity	Strong	Weak, prone to errors
Concurrent Access	Multi-user support	Limited
Automation	SQL, advanced querying	Formulas, macros
Security	Advanced roles, encryption	Basic password protection
Relationships	Complex data relationships	None
Performance	Optimized for large data sets	Slows with large data
Backup	Advanced tools	Manual
Use Case	Complex, multi-user systems	Simple, individual tasks

Understanding these differences helps in selecting the right tool for specific tasks or projects.

13. Primary key and Relationship schema:





What is a Primary Key in SQL?

In SQL, a **Primary Key** is a column (or a combination of columns) that uniquely identifies each row in a table. A table can have only one primary key, and it ensures the following properties:

1. **Uniqueness**: No two rows in a table can have the same value in the primary key column(s).
2. **Non-Null**: A primary key column cannot contain `NULL` values.

The primary key is essential for maintaining data integrity and ensuring each record can be uniquely identified. It also helps in establishing relationships between tables in a database.

Key Characteristics of a Primary Key

1. **Uniqueness**: Ensures data in the column(s) is unique for every record.
2. **Non-Null Constraint**: Primary key values cannot be `NULL`.
3. **Indexing**: A primary key automatically creates a unique clustered index on the column(s).
4. **One per Table**: A table can have only one primary key.

How to Define a Primary Key

You can define a primary key in SQL:

1. **While creating a table**.
2. **On an existing table** using the `ALTER` statement.

Example of Primary Key

1. Defining a Primary Key While Creating a Table

```
CREATE TABLE Students (
    StudentID INT NOT NULL,
    FirstName VARCHAR(50),
    LastName VARCHAR(50),
    Age INT,
    PRIMARY KEY (StudentID)
);
```

- **Explanation**:
  - `StudentID` is the primary key.
  - Each student must have a unique `StudentID`.
  - `StudentID` cannot contain `NULL`.

Insert Data into the Table

```
INSERT INTO Students (StudentID, FirstName, LastName, Age)
VALUES (1, 'Alice', 'Smith', 20),
       (2, 'Bob', 'Johnson', 22),
       (3, 'Charlie', 'Brown', 19);
```

If you try to insert a duplicate or `NULL` value into the `StudentID` column:

```
-- This will fail because StudentID must be unique
INSERT INTO Students (StudentID, FirstName, LastName, Age)
VALUES (1, 'David', 'Miller', 23);

-- This will fail because StudentID cannot be NULL
INSERT INTO Students (StudentID, FirstName, LastName, Age)
VALUES (NULL, 'Eve', 'Davis', 21);
```

2. Defining a Composite Primary Key

A composite primary key is made up of two or more columns.

```
sql
Copy code
CREATE TABLE Orders (
    OrderID INT NOT NULL,
    ProductID INT NOT NULL,
    Quantity INT,
    PRIMARY KEY (OrderID, ProductID)
);
```

- **Explanation**:
  - The combination of `OrderID` and `ProductID` must be unique.
  - Useful when a single column is not enough to uniquely identify a record.

3. Adding a Primary Key to an Existing Table

```
sql
Copy code
ALTER TABLE Students
ADD CONSTRAINT PK_Student PRIMARY KEY (StudentID);
```

Why Use a Primary Key?

1. **Uniqueness**: Ensures that each row is uniquely identifiable.
2. **Data Integrity**: Prevents duplicate and null values in key columns.

3. **Relationships**: Used to create relationships between tables (e.g., Foreign Key references).
4. **Indexing**: Automatically indexes the primary key column for faster query performance.

Example: Relating Tables Using a Primary Key

Table: Students

StudentID	FirstName	LastName
1	Alice	Smith
2	Bob	Johnson

Table: Enrollments

EnrollmentID	StudentID	CourseName
101	1	Math
102	2	Science
103	1	Physics

- **Explanation**:
  - `StudentID` in the `Enrollments` table is a **Foreign Key** that references the `Primary Key` in the `Students` table.
  - This ensures each enrollment is associated with a valid student.

Key Notes

- A primary key cannot allow duplicate or `NULL` values.
- You can create composite primary keys for unique combinations of columns.
- A primary key helps maintain data integrity and enforces a logical structure in relational databases.

Foreign Key :

What is a Foreign Key in SQL?

A **Foreign Key** is a column or a set of columns in a table that establishes a relationship between data in two tables. It refers to the **Primary Key** in another table, ensuring data integrity by enforcing a link between the two tables.

A foreign key:

- Creates a **parent-child relationship** between two tables.
- Ensures that the values in the foreign key column(s) match the primary key values in the referenced table.
- Prevents invalid data from being inserted into the foreign key column.

Key Properties of a Foreign Key

1. **Relationship**: A foreign key links two tables by referencing the primary key in the parent table.
2. **Data Integrity**: Ensures that only valid data (that exists in the parent table) can be inserted in the child table.
3. **Referential Integrity**: Prevents actions that would break the link between the tables, such as deleting a referenced row.
4. **Multiple Foreign Keys**: A table can have more than one foreign key.
5. **Cascade Actions**: Defines what happens when the parent table's data is updated or deleted (e.g., `ON DELETE CASCADE`).

How to Define a Foreign Key

You can define a foreign key:

1. **When creating a table**.
2. **In an existing table** using the `ALTER TABLE` statement.

Example of a Foreign Key

1. Defining a Foreign Key While Creating Tables

Let's create two related tables: **Students** and **Enrollments**.

Parent Table: `Students`

```
CREATE TABLE Students (
    StudentID INT PRIMARY KEY,
    FirstName VARCHAR(50),
    LastName VARCHAR(50)
);
```

Child Table: `Enrollments`

```
sql
Copy code
```

```
CREATE TABLE Enrollments (
    EnrollmentID INT PRIMARY KEY,
    StudentID INT,
    CourseName VARCHAR(50),
    FOREIGN KEY (StudentID) REFERENCES Students(StudentID)
);
```

- **Explanation**:
  - `Students.StudentID` is the **Primary Key** in the `Students` table.
  - `Enrollments.StudentID` is a **Foreign Key** that references `Students.StudentID`.
  - Each `StudentID` in `Enrollments` must exist in the `Students` table.

Insert Data

```
-- Insert into Students (Parent Table)
INSERT INTO Students (StudentID, FirstName, LastName)
VALUES (1, 'Alice', 'Smith'),
       (2, 'Bob', 'Johnson');

-- Insert into Enrollments (Child Table)
INSERT INTO Enrollments (EnrollmentID, StudentID, CourseName)
VALUES (101, 1, 'Math'),
       (102, 2, 'Science');
```

Invalid Insert

If you try to insert a `StudentID` in `Enrollments` that does not exist in `Students`:

```
sql
Copy code
-- This will fail because StudentID 3 does not exist in Students table
INSERT INTO Enrollments (EnrollmentID, StudentID, CourseName)
VALUES (103, 3, 'Physics');
```

---

2. Defining a Foreign Key in an Existing Table

You can add a foreign key to an existing table using the `ALTER TABLE` command:

```
ALTER TABLE Enrollments
ADD CONSTRAINT FK_Student FOREIGN KEY (StudentID)
REFERENCES Students(StudentID);
```

Cascading Actions with Foreign Keys

Foreign keys can define actions when the parent table is updated or deleted:

1. **ON DELETE CASCADE**: Deletes the child rows if the referenced parent row is deleted.
2. **ON UPDATE CASCADE**: Updates the child rows when the referenced parent row is updated.
3. **ON DELETE SET NULL**: Sets the foreign key column to `NULL` if the referenced parent row is deleted.
4. **ON DELETE RESTRICT/NO ACTION**: Prevents deletion of the parent row if it is referenced by the child table.

Example with Cascading Actions

```
CREATE TABLE Students (
    StudentID INT PRIMARY KEY,
    FirstName VARCHAR(50),
    LastName VARCHAR(50)
);

CREATE TABLE Enrollments (
    EnrollmentID INT PRIMARY KEY,
    StudentID INT,
    CourseName VARCHAR(50),
    FOREIGN KEY (StudentID) REFERENCES Students(StudentID)
    ON DELETE CASCADE
    ON UPDATE CASCADE
);
```

- **ON DELETE CASCADE**: If a `StudentID` is deleted from the `Students` table, all related rows in the `Enrollments` table will also be deleted.
- **ON UPDATE CASCADE**: If the `StudentID` is updated in the `Students` table, it will automatically update the corresponding `StudentID` in the `Enrollments` table.

Test ON DELETE CASCADE

```
-- Delete a student
DELETE FROM Students WHERE StudentID = 1;

-- This will automatically delete the corresponding rows in Enrollments
```

Benefits of Using a Foreign Key

1. **Data Integrity**: Ensures data consistency between related tables.
2. **Referential Integrity**: Prevents orphaned rows in the child table.
3. **Logical Relationships**: Clearly defines relationships between tables.
4. **Simplifies Queries**: Helps in joining tables and retrieving related data.

Real-World Example

Tables: `Customers` and `Orders`

Customers Table

CustomerID	Name	Email
1	Alice	alice@email.com
2	Bob	bob@email.com

Orders Table

OrderID	CustomerID	Product	Quantity
101	1	Laptop	2
102	2	Headphones	1

- `CustomerID` in the `Orders` table is a foreign key referencing `CustomerID` in the `Customers` table.

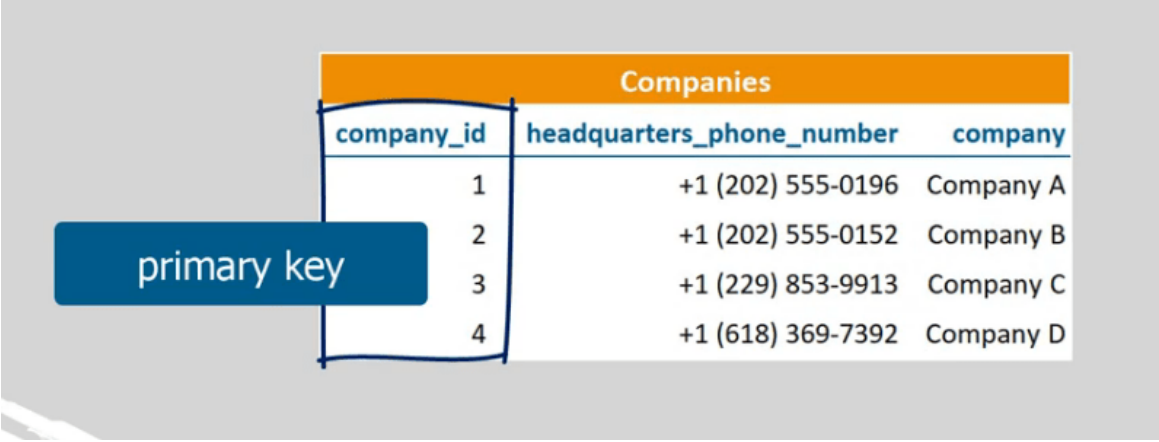
Key Differences: Primary Key vs. Foreign Key

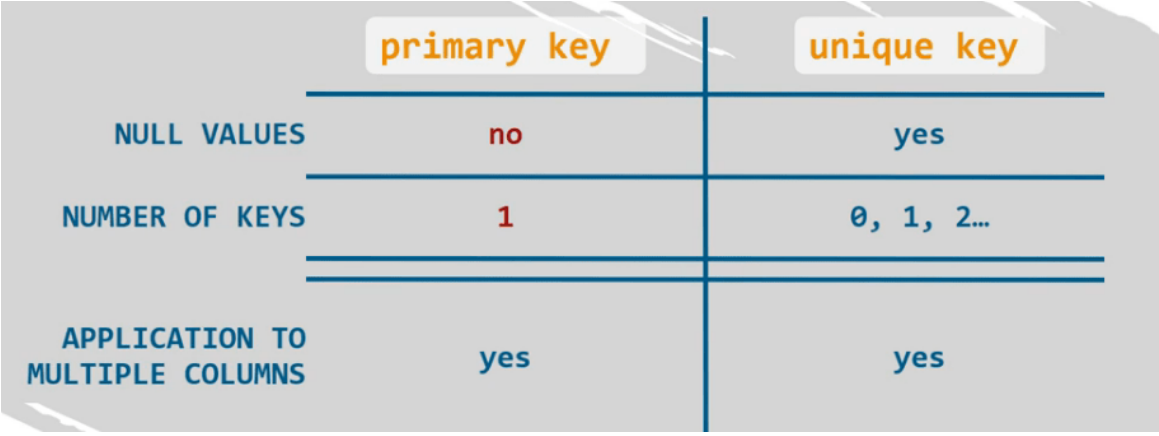
Aspect	Primary Key	Foreign Key
Definition	Uniquely identifies a row in a table.	Links two tables by referencing a primary key.
Uniqueness	Must be unique.	Can have duplicates in the child table.
NULL Values	Cannot contain NULL values.	Can contain NULL values if allowed.
Number per Table	Only one primary key per table.	A table can have multiple foreign keys.

Summary

- A **Foreign Key** links two tables and ensures data consistency and integrity.
- It references the **Primary Key** of another table.
- It supports cascading actions like `ON DELETE CASCADE` or `ON UPDATE SET NULL`.
- It's essential for modeling real-world relationships like `Customers` and `Orders` or `Students` and `Enrollments`

Primary Vs Unique Key :





String :

Types of String Data Types in SQL

1. CHAR

- **Description**: Stores fixed-length strings. If the stored string is shorter than the defined size, it is padded with spaces.
- **Usage**: When the data always has a fixed length.
- **Size**: Up to 255 characters.

Example:

```
sql
Copy code
CREATE TABLE Users (
    Username CHAR(10)
);
-- 'Alex' is stored as 'Alex      ' (padded with spaces)
```

---

2. VARCHAR

- **Description**: Stores variable-length strings. It only uses as much space as the actual string length, plus 1 or 2 bytes for length storage.
- **Usage**: When the data length varies.
- **Size**: Up to 65,535 characters (depending on row size and database).

Example:

```
sql
Copy code
CREATE TABLE Users (
    Email VARCHAR(100)
);
-- 'Alex' is stored as 'Alex' (no padding)
```

ENUM

- **Description**: Stores a predefined list of string values. It allows only one value from the list.
- **Usage**: When the column should have a restricted set of string values.
- **Size**: Up to 65,535 distinct values.

Example:

```
sql
Copy code
CREATE TABLE Products (
    Size ENUM('Small', 'Medium', 'Large')
);
-- Size can only be 'Small', 'Medium', or 'Large'.
```

INTEGER :

1. TINYINT

- **Description**: A very small integer.
- **Range**:
  - **Signed**: -128 to 127
  - **Unsigned**: 0 to 255
- **Size**: 1 byte
- **Usage**: For very small numeric values like flags, status codes, or boolean-like values.

Example:

```
CREATE TABLE Example (
    Status TINYINT
);
-- Status can store values like -1, 0, 1
```

---

2. SMALLINT

- **Description**: A small integer.
- **Range**:
  - **Signed**: -32,768 to 32,767
  - **Unsigned**: 0 to 65,535
- **Size**: 2 bytes
- **Usage**: For slightly larger numbers, such as short identifiers or small counters.

Example:

```
CREATE TABLE Example (
    Age SMALLINT
);
-- Age can store values like 25, 30, 45
```

3. MEDIUMINT

- **Description**: A medium-sized integer.
- **Range**:
  - **Signed**: -8,388,608 to 8,388,607
  - **Unsigned**: 0 to 16,777,215
- **Size**: 3 bytes
- **Usage**: For numbers that are larger than `SMALLINT` but smaller than `INT`.

Example:

```
CREATE TABLE Example (
    Population MEDIUMINT
);
-- Population can store values like 1,000,
```

4. INT (INTEGER)

- **Description**: A standard integer data type.
- **Range**:
  - **Signed**: -2,147,483,648 to 2,147,483,647
  - **Unsigned**: 0 to 4,294,967,295
- **Size**: 4 bytes
- **Usage**: For most general-purpose numeric fields, like IDs or counters.

Example:

```
CREATE TABLE Example (
    UserID INT
);
```

```
-- UserID can store large numbers for identification
```

5. BIGINT

- **Description**: A large integer.
- **Range**:
  - **Signed**: -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
  - **Unsigned**: 0 to 18,446,744,073,709,551,615
- **Size**: 8 bytes
- **Usage**: For very large numbers, like financial data, scientific computations, or large counters.

Example:

```
CREATE TABLE Example (
    NationalDebt BIGINT
);
-- NationalDebt can store massive numbers
```

Comparison Table of Integer Data Types

Data Type	Size	Signed Range	Unsigned Range	Use Case
TINYINT	1 byte	-128 to 127	0 to 255	Flags, small numeric values
SMALLINT	2 bytes	-32,768 to 32,767	0 to 65,535	Small counters, short IDs
MEDIUMINT	3 bytes	-8,388,608 to 8,388,607	0 to 16,777,215	Medium-sized values like population
INT	4 bytes	-2,147,483,648 to 2,147,483,647	0 to 4,294,967,295	Most common for IDs or counters
BIGINT	8 bytes	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	0 to 18,446,744,073,709,551,615	Large numbers like financial data

Choosing the Right Integer Data Type

- Use **TINYINT** for small values such as binary flags or status indicators.

- Use **SMALLINT** for small-range numeric data such as age or short counters.
- Use **MEDIUMINT** when values are expected to fall between the ranges of `SMALLINT` and `INT`.
- Use **INT** for most general-purpose numeric data.
- Use **BIGINT** for very large values, such as global counters, financial data, or scientific measurements.

precision and scale 45.3343

precision : Number of data types 5

scale : values after decimal 3

1. Fixed-Point Data Types

Fixed-point numbers are used when precise decimal values are required. They store numbers with an exact number of digits before and after the decimal point. These are commonly used for financial calculations.

Fixed-Point Data Types in SQL

1. `NUMERIC(p, s)` **or DECIMAL(p, s)`**
   - **Description**: Used to store exact numeric values with fixed precision.
   - **Precision (`p`)**: Total number of digits (both before and after the decimal point).
   - **Scale (`s`)**: Number of digits after the decimal point.
   - **Storage**: Depends on the precision and scale.

Examples

Table with Fixed-Point Values:

```
sql
Copy code
CREATE TABLE FixedPointExample (
    Price DECIMAL(10, 2),  -- Total 10 digits, 2 digits after the decimal po
int
    InterestRate NUMERIC(5, 3)  -- Total 5 digits, 3 digits after the decima
l point
);
```

Inserting Data:

```
sql
Copy code
INSERT INTO FixedPointExample (Price, InterestRate)
VALUES
(1000.50, 5.123),   -- Valid: Matches the precision and scale
(12345.67, 0.045);  -- Valid: Matches the precision and scale
```

Output:

Price	InterestRate
1000.50	5.123
12345.67	0.045

2. Floating-Point Data Types

Floating-point numbers are used when you need to store very large or very small numbers, and precision can be approximate. These are typically used for scientific calculations or measurements.

Floating-Point Data Types in SQL

1. `FLOAT(p)`
   - **Description**: Approximate numeric data type. Precision depends on the hardware.
   - **Precision (`p`)**: Number of binary digits (not decimal digits) for the mantissa. The actual precision in decimal digits is platform-dependent.
   - **Storage**: Varies depending on precision.
2. `REAL`
   - Alias for `FLOAT` in some SQL implementations.
3. `DOUBLE` **or** `DOUBLE PRECISION`
   - **Description**: Similar to `FLOAT`, but allows for double the precision.

Examples

Table with Floating-Point Values:

```
sql
Copy code
CREATE TABLE FloatingPointExample (
    Temperature FLOAT(24),  -- Approximate precision with single-precision f
```

```
loat
    Distance DOUBLE         -- Double-precision float
);
```

Inserting Data:

```
sql
Copy code
INSERT INTO FloatingPointExample (Temperature, Distance)
VALUES
(36.6, 12345.67890),  -- Stores approximate value
(98.123456, 0.00005678);
```

Output:

Temperature	Distance
36.6	12345.6789
98.12346	5.678E-05

Comparison Between Fixed-Point and Floating-Point

Feature	**Fixed-Point (`DECIMAL` or `NUMERIC`)**	**Floating-Point (`FLOAT`, `REAL`, `DOUBLE`)**
Precision	Exact and user-defined	Approximate, hardware-dependent
Use Case	Financial data, precise calculations	Scientific data, very large or small values
Performance	Slower due to precision guarantee	Faster for large datasets
Range	Limited by precision and scale	Much larger range
Storage	Fixed size, depends on precision and scale	Varies depending on precision and type

Choosing the Right Data Type

- Use **fixed-point (`DECIMAL` or `NUMERIC`)** when precision is critical, such as for prices, interest rates, and financial data.
- Use **floating-point (`FLOAT`, `DOUBLE`)** when working with scientific or approximate values, such as temperatures, distances, or measurements.

Other DataTypes:

Comparison of Date/Time Data Types

Data Type	Description	Storage	Example
DATE	Only date	3 bytes	`2024-12-11`
TIME	Only time	3 bytes	`14:30:00`
DATETIME	Date and time	8 bytes	`2024-12-11 14:30:00`
TIMESTAMP	Date and time with auto TZ	4 bytes	`2024-12-11 12:00:00`
YEAR	Year only	1 byte	`2024`
INTERVAL	Duration/difference	Varies	`INTERVAL '5 DAYS'`

Tables :