Additional task:

1. **SQL Data type.**
2. **DATETIME vs TIMESTAMP vs CURRENT_TIMESTAMP.**

Submitted By Khairul Basar

SQL Data Types: A Comprehensive Discussion

SQL data types define the type of data that can be stored in a column. Choosing the correct data type is critical for optimizing storage, ensuring data integrity, and enhancing query performance. Let's dive into the different types of SQL data types, their usage scenarios, and real-world examples.

---

1. Numeric Data Types

Numeric data types store numeric values. They can be further divided into:

A. Integer Types

- **Examples:** `TINYINT`, `SMALLINT`, `MEDIUMINT`, `INT`, `BIGINT`
- **When to Use:**
  - To store whole numbers like counts, IDs, or statuses.
  - Choose the size based on the range of expected values.
- **Example Use Case:**
  - **User ID (Primary Key)**: Use `INT` if the range of IDs is expected to be large, or `SMALLINT` if the table will have fewer entries (e.g., <32,000 rows).

```
CREATE TABLE users (
    user_id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    age TINYINT NOT NULL
);
```

B. Decimal/Fixed-Point Types

- **Examples:** `DECIMAL(p, s)` or `NUMERIC(p, s)`
- **When to Use:**

    - To store exact numeric values, such as currency or financial data, where precision is critical.
- **Example Use Case:**
    - **Bank Account Balance**: Store balances with 2 decimal points of precision.

```sql
CREATE TABLE bank_accounts (
    account_id INT AUTO_INCREMENT PRIMARY KEY,
    balance DECIMAL(10, 2) NOT NULL
);
```

C. Floating-Point Types

- **Examples:** `FLOAT`, `DOUBLE`
- **When to Use:**
    - To store approximate numeric values like scientific data or sensor measurements.
- **Example Use Case:**
    - **Temperature Sensor Data**: Store temperature readings where precision beyond a few decimal points isn't crucial.

```sql
CREATE TABLE temperature_readings (
    sensor_id INT NOT NULL,
    reading DOUBLE NOT NULL
);
```

---

2. String Data Types

A. Character Types

- **Examples:** `CHAR(n)`, `VARCHAR(n)`
- **When to Use:**
    - Use `CHAR(n)` for fixed-length strings (e.g., ISO country codes).
    - Use `VARCHAR(n)` for variable-length strings (e.g., names, email addresses).
- **Example Use Case:**
    - **Customer Emails**: Use `VARCHAR` since email lengths vary.

```sql
CREATE TABLE customers (
    customer_id INT AUTO_INCREMENT PRIMARY KEY,
    email VARCHAR(255) NOT NULL
);
```

B. Text Types

- **Examples:** `TINYTEXT`, `TEXT`, `MEDIUMTEXT`, `LONGTEXT`
- **When to Use:**
  - Use for storing large text blocks, such as comments or articles.
- **Example Use Case:**
  - **Blog Posts**: Use `TEXT` to store content that exceeds `VARCHAR`'s length limits.

```sql
CREATE TABLE blog_posts (
    post_id INT AUTO_INCREMENT PRIMARY KEY,
    content TEXT NOT NULL
);
```

C. Binary Data

- **Examples:** `BINARY(n)`, `VARBINARY(n)`, `BLOB`
- **When to Use:**
  - Use for storing binary data like images, files, or multimedia.
- **Example Use Case:**
  - **Profile Pictures**: Store image data as `BLOB`.

```sql
CREATE TABLE user_profiles (
    user_id INT AUTO_INCREMENT PRIMARY KEY,
    profile_picture BLOB
);
```

---

3. Date and Time Data Types

Examples: `DATE`, `DATETIME`, `TIMESTAMP`, `TIME`, `YEAR`

- **When to Use:**
  - Use `DATE` for storing only dates (e.g., birth dates).
  - Use `DATETIME` or `TIMESTAMP` for storing both date and time (e.g., order timestamps).
  - Use `TIME` for time-only data (e.g., shift start/end times).
- **Example Use Case:**
  - **Order History**: Use `DATETIME` to store when an order was placed.

```
CREATE TABLE orders (
    order_id INT AUTO_INCREMENT PRIMARY KEY,
    order_date DATETIME NOT NULL
);
```

---

4. JSON Data Type

- **When to Use:**
  - To store semi-structured data that doesn't fit neatly into a relational schema (e.g., dynamic configurations or logs).
- **Example Use Case:**
  - **Product Attributes**: Use `JSON` to store attributes like size, color, or materials.

```
CREATE TABLE products (
    product_id INT AUTO_INCREMENT PRIMARY KEY,
    attributes JSON
);
```

---

5. Spatial Data Types

- **Examples:** `GEOMETRY`, `POINT`, `LINESTRING`, `POLYGON`
- **When to Use:**
  - For applications that involve geographical data, such as maps or geofencing.
- **Example Use Case:**
  - **Store Locations**: Use `POINT` to store latitude and longitude coordinates.

```
CREATE TABLE store_locations (
    store_id INT AUTO_INCREMENT PRIMARY KEY,
    location POINT NOT NULL
);
```

How to Choose a Data Type?

1. Optimize Storage

- Use the smallest possible data type that can store the data to save storage space.
  - Example: Use `TINYINT` for storing a `status` field with values 0-255.

2. Ensure Data Integrity

- Choose data types that restrict invalid entries.
  - Example: Use `DATE` for a birth date to prevent invalid dates like "2025-02-30."

3. Improve Query Performance

- Avoid large text fields when searching.
  - Example: Use `VARCHAR` instead of `TEXT` for searchable columns like names.

4. Consider Future Scalability

- Anticipate data growth and select types accordingly.
  - Example: Use `BIGINT` for IDs if you expect the table to exceed 2 billion rows.

Comparison: `DATETIME` vs `TIMESTAMP` vs `CURRENT_TIMESTAMP`

In SQL, the choice between `DATETIME`, `TIMESTAMP`, and `CURRENT_TIMESTAMP` depends on the use case and how you want to handle dates and times, particularly regarding time zones, automatic updates, and storage precision. Here's a detailed breakdown:

---

1. DATETIME

- **Definition**: A date and time without time zone conversion.
- **Range**: From `1000-01-01 00:00:00` to `9999-12-31 23:59:59`.
- **Storage**: Requires **8 bytes**.
- **Time Zone**: It is time-zone agnostic, meaning it stores the value exactly as provided without converting based on the server's or client's time zone.

When to Use

- Use `DATETIME` when:
    1. You don't need time zone adjustments.
    2. You need a large date range (beyond the year 2038 limit of `TIMESTAMP`).
    3. You don't require automatic updates to the column.

Real Example

Scenario: An event management system that stores the start time of events in various countries without considering time zones.

```
CREATE TABLE events (
    event_id INT AUTO_INCREMENT PRIMARY KEY,
    event_name VARCHAR(255) NOT NULL,
    start_time DATETIME NOT NULL
);

INSERT INTO events (event_name, start_time)
VALUES ('Conference', '2025-06-15 10:00:00');
```

- **Why `DATETIME`?**The event's start time is fixed and doesn’t require adjustments based on time zones. For instance, the conference starts at `10:00 AM` local time regardless of where it's queried.

---

2. TIMESTAMP

- **Definition**: A date and time with automatic time zone conversion based on the server's or client’s time zone.
- **Range**: From `1970-01-01 00:00:01 UTC` to `2038-01-19 03:14:07 UTC`.
- **Storage**: Requires **4 bytes**.
- **Time Zone**: Adjusted automatically to/from UTC based on the server’s or client’s time zone.

When to Use

- Use `TIMESTAMP` when:
    1. You need to store time-zone-sensitive information.
    2. You need automatic time zone conversion for consistent global queries.
    3. You want the column to automatically update on record modification (with `ON UPDATE CURRENT_TIMESTAMP`).

Real Example

Scenario: A chat application that logs message timestamps in UTC, with conversion for users in different time zones.

```sql
CREATE TABLE messages (
    message_id INT AUTO_INCREMENT PRIMARY KEY,
    user_id INT NOT NULL,
    message_text TEXT NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

INSERT INTO messages (user_id, message_text)
VALUES (1, 'Hello from New York!');
```

- **Why `TIMESTAMP`?**If the server stores all timestamps in UTC, users in different time zones can query the messages in their local time.

3. CURRENT_TIMESTAMP

- **Definition**: A **function** that provides the current date and time based on the server's time zone.
- **Behavior**:
    - When used as a **default value**, it automatically assigns the current timestamp during insertion.
    - When combined with `ON UPDATE`, it updates the timestamp whenever the row is modified.

When to Use

- Use `CURRENT_TIMESTAMP` when:
    1. You need to auto-populate date and time during insertion (`DEFAULT CURRENT_TIMESTAMP`).
    2. You need to track when a record was last updated (`ON UPDATE CURRENT_TIMESTAMP`).

Real Example

Scenario: An e-commerce system where you track order creation and update times.

```
CREATE TABLE orders (
    order_id INT AUTO_INCREMENT PRIMARY KEY,
    order_status VARCHAR(50) NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP
);

INSERT INTO orders (order_status)
VALUES ('Pending');

UPDATE orders
SET order_status = 'Shipped'
WHERE order_id = 1;
```

- **Why `CURRENT_TIMESTAMP`?**

    - `created_at`: Tracks when the order was placed.
    - `updated_at`: Updates automatically to the current timestamp whenever the row is modified.