MySQL Data Types

Contents

[1. MySQL Data Types 2](#_Toc185709931)

[String Data Types 2](#_Toc185709932)

[Integer Data Types 7](#_Toc185709933)

[Decimal, Float, Double, and Boolean 9](#_Toc185709934)

[Enums (Enumerations) 12](#_Toc185709935)

[Date, Time, Timestamp, and Blob Data Types 14](#_Toc185709936)

# MySQL Data Types

MySQL provides several data types, categorized into five main types, with this lecture focusing on four: **Strings**, **Numbers**, **Date and Time**, and **Blobs**. Advanced types like geospatial data types are not covered.

## String Data Types

The three main string data types in MySQL are **CHAR**, **VARCHAR**, and **TEXT**.

**1. CHAR**

* **Definition:** Fixed-length string data type.
* **Declaration:** CHAR(X) where X is the length (0–255).
  + Example: CHAR(50)
* **Characteristics:**
  + Always uses the defined length for storage, regardless of the actual string size.
  + Padding with spaces if the string is shorter than X.
  + If a string exceeds X, it is truncated or raises an error.
* **Space Usage:**
  + A column defined as CHAR(4):
    - Stores "A" as A (4 bytes).
    - Stores "AB" as AB (4 bytes).
    - Stores "ABCD" as ABCD (4 bytes).
  + Always occupies the full declared size.

**2. VARCHAR**

* **Definition:** Variable-length string data type.
* **Declaration:** VARCHAR(X) where X is the maximum length (0–65,535).
  + Example: VARCHAR(255)
* **Characteristics:**
  + Only uses storage space equivalent to the length of the stored string plus 1 byte for storing the string length.
  + Truncates or raises an error if the string length exceeds X.
* **Space Usage:**
  + A column defined as VARCHAR(4):
    - Stores "A" as 1, A (2 bytes).
    - Stores "AB" as 2, AB (3 bytes).
    - Stores "ABCD" as 4, ABCD (5 bytes).
  + The first byte stores the string length; the rest stores the string itself.
* **Comparison with CHAR:**
  + More efficient for variable-length data.
  + Does not pad with spaces, saving storage.

**Key Differences Between CHAR and VARCHAR**

| **Feature** | **CHAR** | **VARCHAR** |
| --- | --- | --- |
| **Length** | Fixed | Variable |
| **Max Size** | 0–255 characters | 0–65,535 characters |
| **Padding** | Adds spaces to reach max size | No padding; uses actual length |
| **Storage** | Always uses defined size | Uses actual size + 1 byte |
| **Use Case** | Fixed-length fields (e.g., codes) | Variable-length fields (e.g., names) |

**How Data is Stored**

* **CHAR:**
  + Takes the exact amount of space as defined by the length (X).
* **VARCHAR:**
  + Stores the string length in 1 byte, followed by the actual string.
  + Saves space for shorter strings but slightly increases overhead.

**Examples:**

1. Empty String:
   * CHAR(4): 4 bytes (e.g., ).
   * VARCHAR(4): 1 byte (length 0).
2. String "A":
   * CHAR(4): 4 bytes (e.g., A ).
   * VARCHAR(4): 2 bytes (1, A).
3. String "ABCD":
   * CHAR(4): 4 bytes.
   * VARCHAR(4): 5 bytes (4, ABCD).

**Advanced Data Types**

* Geospatial data types (e.g., latitude, longitude) are used for specialized storage but are not the focus of this class.

**Summary**

* **CHAR** is suitable for fixed-length fields, ensuring consistent space usage but may waste storage for shorter strings.
* **VARCHAR** is more flexible, storing variable-length strings efficiently but with a slight overhead.
* Proper selection of data types depends on the specific use case and expected data patterns.

**Handling Large Strings in MySQL (Understanding Size Storage)**

When dealing with large strings in **VARCHAR**, MySQL uses a dynamic mechanism to store the size of the string, adjusting based on the total size of the data. This ensures efficient storage while maintaining clarity for parsing.

**Key Concepts**

1. **Size Storage in VARCHAR**
   * MySQL stores the **length** of the string in bytes before storing the actual string.
   * The number of bytes used to store the length depends on the value of the size:
     + For sizes ≤ 255: 1 byte is used to store the size.
     + For sizes ≥ 256: 2 bytes are used to store the size.
2. **How MySQL Determines the Number of Bytes for Size**
   * MySQL evaluates the total size of the data:
     + If the **total size** of the data (length + string) is ≤ 256 bytes, **1 byte** is used for the size.
     + If the **total size** of the data exceeds 256 bytes, **2 bytes** are used for the size.
3. **Example Calculations**
   * **Case 1:** String length = 255 characters.
     + To store the size (255):
       - Requires 1 byte.
     + Total bytes = 1 (size) + 255 (string) = **256 bytes**.
   * **Case 2:** String length = 256 characters.
     + To store the size (256):
       - Requires 2 bytes.
     + Total bytes = 2 (size) + 256 (string) = **258 bytes**.
4. **How MySQL Parses the Data**
   * MySQL distinguishes between 1-byte and 2-byte size storage based on the **total size of the data**:
     + If the **total size ≤ 256 bytes**, the **first byte** is interpreted as the size.
     + If the **total size > 256 bytes**, the **first two bytes** are interpreted as the size.

**Summary of Key Rules**

* **String size ≤ 255:** Size is stored in 1 byte.
* **String size ≥ 256:** Size is stored in 2 bytes.
* **Parsing Logic:**
  + Total size ≤ 256: Interpret the first byte as size.
  + Total size > 256: Interpret the first two bytes as size.
* **Example Total Sizes:**
  + String of length 255 → 256 bytes total.
  + String of length 256 → 258 bytes total.

**Text Data Types in Databases**

1. **Overview of Text Data Types:**
   * Text data types are similar to dynamic arrays, allowing flexible and dynamic storage of strings.
   * Text columns can store large strings and are ideal for data like paragraphs, blog posts, or lengthy content.
2. **Types of Text Data Columns:**
   * **TinyText:**
     + Maximum size: 255 bytes (255 characters).
     + Suitable for small strings.
   * **Text:**
     + Maximum size: 64 kilobytes.
     + Can store moderate-sized strings.
   * **MediumText:**
     + Maximum size: 16 megabytes.
     + Suitable for larger strings, e.g., documents or long descriptions.
   * **LongText:**
     + Maximum size: 4 gigabytes.
     + Can store massive strings, e.g., DNA sequences or extensive JSON responses.
3. **Use Cases for Text Data Types:**
   * Storing large content such as blog posts, articles, and extensive textual data.
   * Storing JSON responses or similar large data payloads.
   * Storing data unsuitable for fixed or predictable sizes.
4. **Challenges and Limitations:**
   * **No Indexing Support:**
     + Text columns do not support indexing, making queries slow.
     + Searching for a specific value (e.g., name = "Naman") requires a row-by-row scan, significantly increasing query time.
   * **Performance Concerns:**
     + Databases are not optimized for handling extremely large data sizes (e.g., 4GB strings).
     + Query performance degrades with the size of the data stored.
   * **Storage Recommendations:**
     + For very large data (e.g., DNA sequences, large files), databases should not be the primary storage.
     + Instead, use file storage solutions like **Amazon S3** or other file storage systems optimized for large datasets.
5. **Best Practices:**
   * Avoid using text data types for small and frequently queried data (e.g., names or IDs).
   * For large datasets:
     + Store textual data in file stores and keep metadata in the database.
     + Use databases for manageable and indexable data.
   * Use text data types only when dynamic storage of large strings is necessary, and frequent querying isn't required.
6. **Additional Insights:**
   * Text and BLOB (Binary Large Object) types are similar internally.
     + Use **BLOB** for binary data and **TEXT** for string data.
   * While databases provide the capability to store massive data, they are not inherently optimized for such use cases.

**Summary:**

* Text data types are flexible but come with trade-offs in indexing and performance.
* For storing large textual or binary data, consider external file storage solutions over database columns.
* Use appropriate text data types (TinyText, Text, MediumText, LongText) based on the expected data size and access patterns.

## Integer Data Types

1. **Overview of Integer Data Types:**
   * Integers are used to store whole numbers in databases and programming.
   * Two primary variants of integers:
     + **Signed Integers:** Can store both positive and negative values (e.g., -128 to 127).
     + **Unsigned Integers:** Can only store non-negative values (e.g., 0 to 255).
2. **Unsigned vs. Signed Integers:**
   * **Signed Integers:**
     + Allow both positive and negative numbers.
     + Example range (for 1 byte): -128 to 127.
   * **Unsigned Integers:**
     + Only positive numbers.
     + Example range (for 1 byte): 0 to 255.
3. **Types of Integer Data Types in SQL:**
   * **TinyInt:**
     + **Size:** 1 byte.
     + **Range (Signed):** -128 to 127.
     + **Range (Unsigned):** 0 to 255.
   * **SmallInt:**
     + **Size:** 2 bytes.
     + **Range (Signed):** -32,768 to 32,767.
     + **Range (Unsigned):** 0 to 65,535.
   * **MediumInt:**
     + **Size:** 3 bytes.
     + **Range (Signed):** -8,388,608 to 8,388,607.
     + **Range (Unsigned):** 0 to 16,777,215.
   * **Int (or Integer):**
     + **Size:** 4 bytes.
     + **Range (Signed):** -2,147,483,648 to 2,147,483,647.
     + **Range (Unsigned):** 0 to 4,294,967,295.
   * **BigInt:**
     + **Size:** 8 bytes.
     + **Range (Signed):** -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.
     + **Range (Unsigned):** 0 to 18,446,744,073,709,551,615.
4. **Key Use Cases:**
   * **TinyInt:**
     + Useful for small numbers like status codes or flags (e.g., 0 for false, 1 for true).
   * **SmallInt:**
     + Suitable for slightly larger ranges, like age or small counters.
   * **MediumInt:**
     + Used when the range of SmallInt is insufficient, but Int is overkill.
   * **Int (or Integer):**
     + Commonly used for general-purpose numeric data.
     + Often used for IDs when the dataset size is within the 4-byte range.
   * **BigInt:**
     + Ideal for large numbers, such as IDs in systems with massive data, or for financial calculations.
5. **Storage and Efficiency:**
   * The size of the integer type determines how many numbers can be stored.
   * Choosing the right type helps optimize storage:
     + For small datasets or low ranges, use TinyInt or SmallInt.
     + For massive datasets, use BigInt to avoid running out of range.
6. **Best Practices:**
   * Use **BigInt** for unique IDs, especially in systems with large-scale data (e.g., 64-bit IDs).
   * Use **SmallInt** or **TinyInt** where the range suffices to save storage.
   * Consider the signed/unsigned requirement:
     + Use **unsigned** when negative numbers are unnecessary, as it doubles the range for positive numbers.

**Summary:**

* Integer types vary by size and range: TinyInt, SmallInt, MediumInt, Int, and BigInt.
* Use the smallest integer type that fits the data to save storage.
* Use BigInt for scalability in identifiers and applications requiring very large numbers.

## Decimal, Float, Double, and Boolean

**Overview of Data Types**

* Data types are broadly categorized into:
  + **String Data Types**
  + **Numeric Data Types**
  + **Date and Time Data Types**
  + **Blob Data Types**
* Previous discussions covered:
  + **String Data Types** (e.g., VARCHAR, CHAR)
  + **Integral Numeric Data Types** (TINYINT, SMALLINT, MEDIUMINT, BIGINT, etc.)

**Decimal (Fixed-Point) Data Types**

1. **Definition:**
   * Used for storing **exact values** with defined precision and scale.
   * Useful for financial data where precision is critical (e.g., bank transactions).
2. **Characteristics:**
   * Defined as DECIMAL(P, S):
     + **P (Precision):** Total number of digits in the number.
     + **S (Scale):** Number of digits after the decimal point.
   * Example:
     + For 3.141, P = 4, S = 3.
     + For DECIMAL(5, 2), numbers like 123.45 or 12.34 are valid, but 123.456 is invalid.
3. **Advantages:**
   * **Exact Value Storage:** Guarantees no loss of precision.
   * Ideal for scenarios like:
     + Financial transactions (e.g., currency).
     + Scientific calculations where exact values are mandatory.
4. **Limitations:**
   * Maximum **Precision** and **Scale:** 1 to 65.
   * Cannot handle extremely large or small numbers that exceed 65 digits.

**Floating-Point Data Types (FLOAT and DOUBLE)**

1. **Definition:**
   * Used for storing **approximate values** with large ranges.
   * Follows the **IEEE 754** standard.
2. **Characteristics:**
   * **FLOAT:** 4 bytes, less precision but more efficient.
   * **DOUBLE:** 8 bytes, higher precision compared to FLOAT.
   * Stores numbers in scientific notation (e.g., 6.5 x 10^-19).
3. **Advantages:**
   * Can handle very small or very large numbers efficiently.
   * Useful for applications where precision is less critical:
     + Scientific calculations (non-critical precision).
     + Large datasets requiring less storage.
4. **Limitations:**
   * Does **not store exact values** (e.g., 1.999999 may be approximated to 2.0).
   * Not suitable for critical financial or legal data.
5. **Comparison:**
   * **Decimal vs. Float/Double:**
     + Use DECIMAL for exact values.
     + Use FLOAT/DOUBLE for ranges where exact precision is unnecessary.

**Boolean Data Type**

1. **Definition:**
   * Represents **True/False** or **1/0** values.
2. **Characteristics in MySQL:**
   * Internally stored as a **TINYINT** (1 byte).
     + **True = 1**
     + **False = 0**
   * Though a boolean could theoretically be represented using a single bit, MySQL uses 1 byte for simplicity.
3. **Applications:**
   * Logical flags (e.g., is\_active, is\_deleted).
   * Conditions or switches in programming.
4. **Note:**
   * The 1-byte size means it is less storage-efficient than a 1-bit representation, but this is a design choice in MySQL.

**Key Insights and Use Cases**

1. **When to Use Decimal:**
   * Financial systems requiring precision (e.g., transactions, accounting).
   * Scenarios where even minor approximations can lead to significant errors.
2. **When to Use Float/Double:**
   * Scientific computations where precision is acceptable.
   * Situations requiring large ranges or very small numbers (e.g., probabilities, scientific constants).
3. **Boolean:**
   * Used for binary states (on/off, yes/no).
   * Efficient for logical checks and conditions.
4. **Why Use Both Float/Double and Decimal?**
   * **Float/Double**: Handles very large ranges efficiently with approximations.
   * **Decimal**: Handles precise numbers but is limited in range and digits.

**Practical Examples**

* **Decimal:**
  + Banking: Store 1.999999 as 1.9999 exactly to avoid financial discrepancies.
* **Float/Double:**
  + Scientific Data: Store values like 6.5 x for research purposes.
* **Boolean:**
  + User Status: Track is\_logged\_in as True or False.

## Enums (Enumerations)

**What are Enums?**

* **Definition:** Enumerations (enums) are a data type that allows you to define a fixed set of allowed values for a variable.
* **Purpose:** Helps prevent errors like typos or incorrect values by limiting input to predefined options.

**Usage Example:**

* **Scenario:** Working at BookMyShow to track seat status.
  + Possible statuses: booked, available, reserved, undefined.
* **Problem with strings:**
  + Typos like BOKED or case sensitivity (Booked vs. booked) can cause bugs.
  + Requires extra handling in code to manage these issues.

**How Enums Solve the Problem:**

* Enums restrict values to predefined options (e.g., booked, reserved, available, undefined).
* **Declaration Syntax**

|  |
| --- |
| enum SeatStatus { booked, reserved, available, undefined }  SeatStatus status; |

* + Only one of the specified values can be assigned to status.
  + Any other value results in an error.

**Advantages of Enums:**

1. **Error Prevention:** Avoids invalid inputs and typos.
2. **Code Readability:** Ensures values are meaningful and consistent.
3. **Efficiency in Programming Languages:**
   * Internally stored as integers (e.g., booked = 1, reserved = 2).
   * Space-efficient as only integers are stored.

**Enums in MySQL**

**How MySQL Stores Enums:**

* Unlike programming languages, MySQL **does not store enums as integers**.
* Internally, enums in MySQL are stored as **strings**.

**Disadvantages of Enums in MySQL:**

1. **Space Inefficiency:**
   * Strings are stored for each enum value, leading to higher space usage.
   * For instance, storing booked repeatedly uses 6 bytes each time.
2. **Avoidance Recommendation:**
   * Enums in MySQL are not ideal due to their inefficiency.

**Alternative to Enums in MySQL:**

* **Using Separate Table for Status:**
  1. Create a dedicated table for statuses

|  |
| --- |
| CREATE TABLE SeatStatus (      id INT PRIMARY KEY,      status\_name VARCHAR(50)  ); |

* 1. Reference this table in your main table

|  |
| --- |
| CREATE TABLE Seats (      seat\_id INT PRIMARY KEY,      status\_id INT,      FOREIGN KEY (status\_id) REFERENCES SeatStatus(id)  ); |

* + **Advantages:**
    - Ensures consistency without duplicating string data.
    - Space-efficient as integers are stored instead of strings.
  + **Disadvantages:**
    - Slightly more complex queries (requires joins).

**Best Practices for Enums in MySQL:**

* **Do not use MySQL enums if:**
  + The set of values might change frequently.
  + Space efficiency is critical.
* **Use an external table for enums instead.**

**Summary:**

* Enums are powerful for restricting input values to predefined options.
* In programming languages, enums are typically stored as integers, making them efficient.
* MySQL enums are stored as strings, which is inefficient for space.
* For better design in databases, use a separate table for enums and reference it with foreign keys.

## Date, Time, Timestamp, and Blob Data Types

**Date and Time Data Types**

**1. DATE Data Type:**

* **Purpose:** Stores only the date (no time).
* **Format:** YYYY-MM-DD (e.g., 2023-12-21).
* **Storage:** Uses **4 bytes**.
* **Limitation:** Cannot store time.

**2. TIME Data Type:**

* **Purpose:** Stores only the time (no date).
* **Format:** HH:MM:SS (e.g., 14:30:00).
* **Storage:** Uses **4 bytes**.
* **Limitation:** Cannot store date.

**3. DATETIME Data Type:**

* **Purpose:** Stores both date and time.
* **Format:** YYYY-MM-DD HH:MM:SS (e.g., 2023-12-21 14:30:00).
* **Storage:** Uses **8 bytes**.

**4. TIMESTAMP Data Type:**

* **Purpose:** Stores both date and time.
* **Format:** Similar to DATETIME.
* **Storage:** Uses **4 bytes**.
* **How it works:**
  + Stores the number of milliseconds since **January 1, 1970 (Unix epoch)**.
  + Efficient but has limitations due to its 4-byte size.
* **Problem with TIMESTAMP:**
  + **2038 Problem:**
    - By the year **2038**, the number of milliseconds since 1970 will exceed the maximum value storable in 4 bytes.
    - This leads to **integer overflow**, similar to the **Y2K bug**.
  + **Y2K Bug Parallel:**
    - Storing years as 2 digits (e.g., 99 for 1999) caused problems in the year 2000 (00 interpreted as earlier than 99).

**Blob Data Types**

**What is a Blob?**

* **Definition:** Blob stands for **Binary Large Object**, used to store binary data like files.
* **Purpose:** Technically allows storing files (e.g., images, videos, PDFs) in a database.
* **Blob Types and Sizes:**
  1. **Tiny Blob:** Up to **255 bytes**.
  2. **Blob:** Up to **65 KB**.
  3. **Medium Blob:** Up to **16 MB**.
  4. **Long Blob:** Up to **4 GB**.

**Why Not Use Blobs in Databases?**

1. **Space and Performance Issues:**
   * Databases are not optimized for storing large binary data.
   * Storing files directly in a database can increase storage and retrieval time.
2. **Query Limitations:**
   * Databases are better suited for **metadata** and **querying**.
   * Files (blobs) are not typically queried directly.

**Recommended Approach:**

* Store files in an **external file system** (e.g., **AWS S3**, **Google Cloud Storage**, or local file systems).
* Store the **path to the file** in the database instead of the file itself.
* **Advantages:**
  + Efficient file handling and retrieval.
  + Better performance and scalability.

**When to Use Blobs:**

* Only in cases where storing files directly in the database is unavoidable due to specific use cases.

**Comparison of Date, Time, and Blob Data Types**

| **Data Type** | **Purpose** | **Storage** | **Key Points** |
| --- | --- | --- | --- |
| DATE | Stores only the date | 4 bytes | No time information. |
| TIME | Stores only the time | 4 bytes | No date information. |
| DATETIME | Stores both date & time | 8 bytes | Larger storage but no overflow issues like TIMESTAMP. |
| TIMESTAMP | Stores both date & time | 4 bytes | Efficient, but affected by the **2038 problem**. |
| BLOB | Stores binary data (files) | Variable | Use only when necessary; prefer file systems for file storage. |

**Summary**

1. Choose appropriate data types based on requirements:
   * Use DATE, TIME, or DATETIME for temporal data.
   * Avoid TIMESTAMP for long-term storage due to its **2038 limitation**.
2. For binary data:
   * Avoid using BLOB unless absolutely necessary.
   * Store files externally and save the file path in the database for better performance and scalability.