02 MySQL Data types -KirkYagami

String Data Types

Data type	Description
CHAR(size)	A FIXED length string (can contain letters, numbers, and special characters). The *size* parameter specifies the column length in characters - can be from 0 to 255. Default is 1
VARCHAR(size)	A VARIABLE length string (can contain letters, numbers, and special characters). The *size* parameter specifies the maximum string length in characters - can be from 0 to 65535
BINARY(size)	Equal to CHAR(), but stores binary byte strings. The *size* parameter specifies the column length in bytes. Default is 1
VARBINARY(size)	Equal to VARCHAR(), but stores binary byte strings. The *size* parameter specifies the maximum column length in bytes.
TINYBLOB	For BLOBs (Binary Large Objects). Max length: 255 bytes
TINYTEXT	Holds a string with a maximum length of 255 characters
TEXT(size)	Holds a string with a maximum length of 65,535 bytes
BLOB(size)	For BLOBs (Binary Large Objects). Holds up to 65,535 bytes of data
MEDIUMTEXT	Holds a string with a maximum length of 16,777,215 characters
MEDIUMBLOB	For BLOBs (Binary Large Objects). Holds up to 16,777,215 bytes of data
LONGTEXT	Holds a string with a maximum length of 4,294,967,295 characters
LONGBLOB	For BLOBs (Binary Large Objects). Holds up to 4,294,967,295 bytes of data
ENUM(val1, val2, val3, ...)	A string object that can have only one value, chosen from a list of possible values. You can list up to 65535 values in an ENUM list. If a value is inserted that is not in the list, a blank value will be inserted. The values are sorted in the order you enter them
SET(val1, val2, val3, ...)	A string object that can have 0 or more values, chosen from a list of possible values. You can list up to 64 values in a SET list

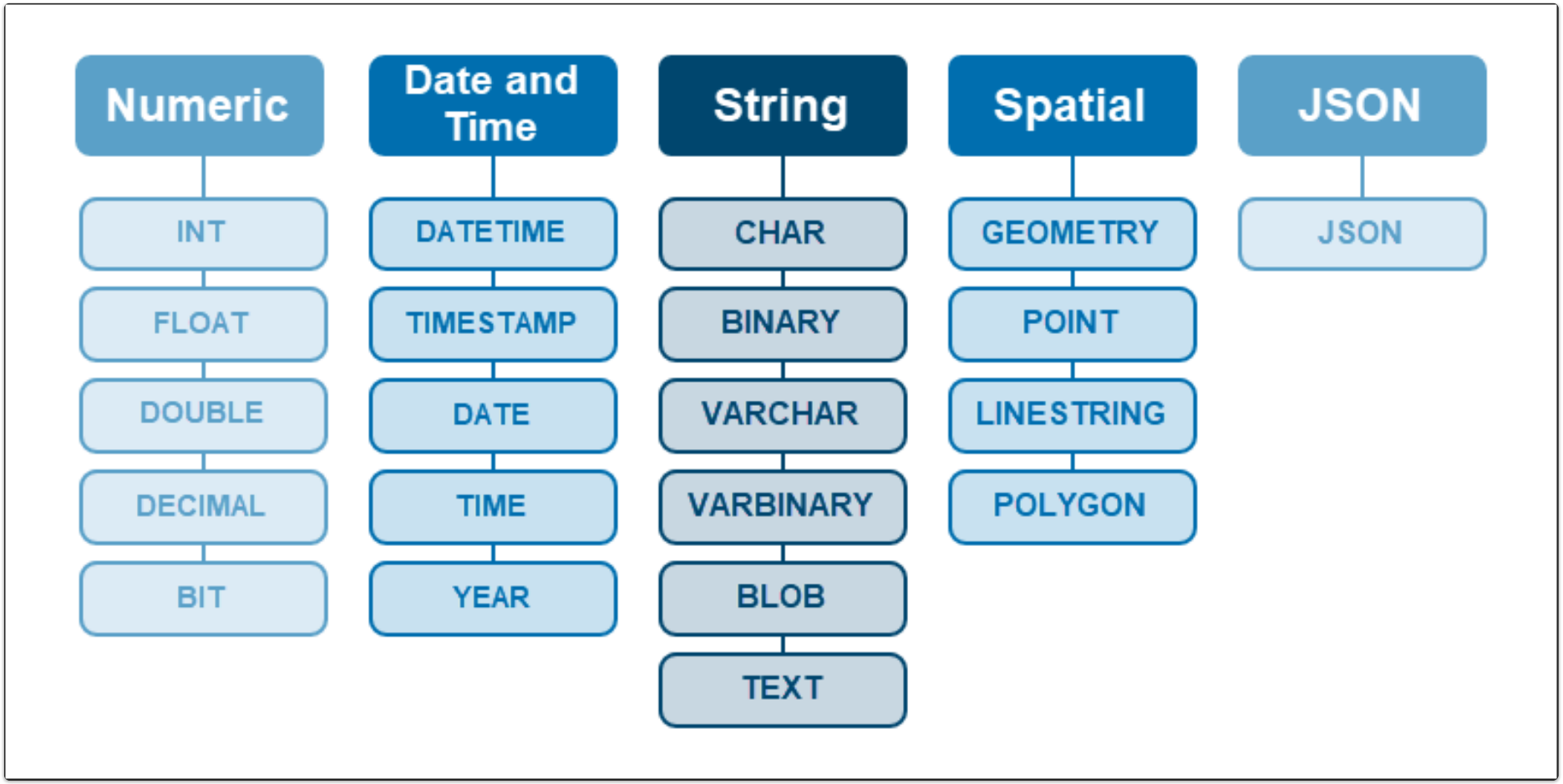
Numeric Data Types

Data type	Description
BIT(*size*)	A bit-value type. The number of bits per value is specified in *size*. The *size* parameter can hold a value from 1 to 64. The default value for *size* is 1.
TINYINT(*size*)	A very small integer. Signed range is from -128 to 127. Unsigned range is from 0 to 255. The *size* parameter specifies the maximum display width (which is 255)
BOOL	Zero is considered as false, nonzero values are considered as true.
BOOLEAN	Equal to BOOL
SMALLINT(*size*)	A small integer. Signed range is from -32768 to 32767. Unsigned range is from 0 to 65535. The *size* parameter specifies the maximum display width (which is 255)
MEDIUMINT(*size*)	A medium integer. Signed range is from -8388608 to 8388607. Unsigned range is from 0 to 16777215. The *size* parameter specifies the maximum display width (which is 255)
INT(*size*)	A medium integer. Signed range is from -2147483648 to 2147483647. Unsigned range is from 0 to 4294967295. The *size* parameter specifies the maximum display width (which is 255)
INTEGER(*size*)	Equal to INT(size)
BIGINT(*size*)	A large integer. Signed range is from -9223372036854775808 to 9223372036854775807. Unsigned range is from 0 to 18446744073709551615. The *size* parameter specifies the maximum display width (which is 255)
FLOAT(*size*, *d*)	A floating point number. The total number of digits is specified in *size*. The number of digits after the decimal point is specified in the *d* parameter. This syntax is deprecated in MySQL 8.0.17, and it will be removed in future MySQL versions
FLOAT(*p*)	A floating point number. MySQL uses the *p* value to determine whether to use FLOAT or DOUBLE for the resulting data type. If *p* is from 0 to 24, the data type becomes FLOAT(). If *p* is from 25 to 53, the data type becomes DOUBLE()
DOUBLE(*size*, *d*)	A normal-size floating point number. The total number of digits is specified in *size*. The number of digits after the decimal point is specified in the *d* parameter
DOUBLE PRECISION(*size*, *d*)	
DECIMAL(*size*, *d*)	An exact fixed-point number. The total number of digits is specified in *size*. The number of digits after the decimal point is specified in the *d* parameter. The maximum number for *size* is 65. The maximum number for *d* is 30. The default value for *size* is 10. The default value for *d* is 0.
DEC(*size*, *d*)	Equal to DECIMAL(size,d)

Note: All the numeric data types may have an extra option: UNSIGNED or ZEROFILL. If you add the UNSIGNED option, MySQL disallows negative values for the column. If you add the ZEROFILL option, MySQL automatically also adds the UNSIGNED attribute to the column.

Date and Time Data Types

Data type	Description
DATE	A date. Format: YYYY-MM-DD. The supported range is from '1000-01-01' to '9999-12-31'
DATETIME(*fsp*)	A date and time combination. Format: YYYY-MM-DD hh:mm:ss. The supported range is from '1000-01-01 00:00:00' to '9999-12-31 23:59:59'. Adding DEFAULT and ON UPDATE in the column definition to get automatic initialization and updating to the current date and time
TIMESTAMP(*fsp*)	A timestamp. TIMESTAMP values are stored as the number of seconds since the Unix epoch ('1970-01-01 00:00:00' UTC). Format: YYYY-MM-DD hh:mm:ss. The supported range is from '1970-01-01 00:00:01' UTC to '2038-01-09 03:14:07' UTC. Automatic initialization and updating to the current date and time can be specified using DEFAULT CURRENT_TIMESTAMP and ON UPDATE CURRENT_TIMESTAMP in the column definition
TIME(*fsp*)	A time. Format: hh:mm:ss. The supported range is from '-838:59:59' to '838:59:59'
YEAR	A year in four-digit format. Values allowed in four-digit format: 1901 to 2155, and 0000. MySQL 8.0 does not support year in two-digit format.



1. Introduction to MySQL Data Types

MySQL supports a variety of data types for storing different kinds of information. These data types can be broadly categorized into three main groups:

a) Numeric Types
b) String Types
c) Date and Time Types

Understanding these data types is crucial for efficient database design, optimal storage usage, and query performance.

2. Numeric Data Types

2.1 Integer Types

MySQL provides several integer types, varying in storage size and range:

- TINYINT: 1 byte, range from -128 to 127 or 0 to 255 (unsigned)
- SMALLINT: 2 bytes, range from -32,768 to 32,767 or 0 to 65,535 (unsigned)
- MEDIUMINT: 3 bytes, range from -8,388,608 to 8,388,607 or 0 to 16,777,215 (unsigned)
- INT: 4 bytes, range from -2,147,483,648 to 2,147,483,647 or 0 to 4,294,967,295 (unsigned)
- BIGINT: 8 bytes, range from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 or 0 to 18,446,744,073,709,551,615 (unsigned)

Backend storage: Integers are stored in binary format, using two's complement for signed numbers.

Best practices:

- Choose the smallest data type that can accommodate your data to save storage space.
- Use UNSIGNED when you only need non-negative values to effectively double the positive range.

Use cases:

- TINYINT: Small enumerated types, boolean flags (0 or 1)
- SMALLINT: Counters, small quantities
- INT: Most general-purpose integer needs, primary keys
- BIGINT: Very large numbers, timestamps

2.2 Fixed-Point Types

- DECIMAL(M,D): Exact fixed-point number. M is the total number of digits (precision), and D is the number of digits after the decimal point (scale).

Backend storage: DECIMAL values are stored as binary strings, with each 9-digit portion taking 4 bytes.

Best practices:

- Use DECIMAL for financial calculations where precision is crucial.

- Specify both precision and scale to avoid unexpected rounding.

Use cases:

- Currency amounts
- Tax rates
- Scientific calculations requiring exact precision

2.3 Floating-Point Types

- FLOAT: 4 bytes, single-precision floating-point number
- DOUBLE: 8 bytes, double-precision floating-point number

Backend storage: These use the IEEE 754 floating-point standard.

Best practices:

- Use floating-point types when exact precision is not required.
- Be aware of potential rounding errors in calculations.

Use cases:

- Scientific calculations where minor imprecisions are acceptable
- Storing very large or small numbers where the exact decimal places are not critical

3. String Data Types

3.1 CHAR and VARCHAR

- CHAR(M): Fixed-length string, always occupies M bytes ($1 \leq M \leq 255$)
- VARCHAR(M): Variable-length string, 1 to 65,535 bytes

Backend storage:

- CHAR: Stored with fixed length, right-padded with spaces
- VARCHAR: Stored with a length prefix (1 or 2 bytes) followed by the actual data

Best practices:

- Use CHAR for fixed-length strings (e.g., state abbreviations, ISO codes)
- Use VARCHAR for variable-length strings to save space

Use cases:

- CHAR: Country codes, zip codes
- VARCHAR: Names, addresses, product descriptions

3.2 TEXT Types

- TINYTEXT: Up to 255 bytes
- TEXT: Up to 65,535 bytes
- MEDIUMTEXT: Up to 16,777,215 bytes
- LONGTEXT: Up to 4,294,967,295 bytes

Backend storage: Stored separately from the main table, with a pointer in the table row.

Best practices:

- Use TEXT types for large strings that exceed VARCHAR limits
- Be aware of performance implications when querying or sorting large TEXT columns

Use cases:

- TINYTEXT: Short comments or descriptions
- TEXT: Longer product descriptions, articles
- MEDIUMTEXT/LONGTEXT: Very large textual content like blog posts or documentation

3.3 BINARY and VARBINARY

Similar to CHAR and VARCHAR, but for binary data.

Backend storage: Stored as-is, without character set conversion.

Best practices:

- Use for data that needs byte-level comparisons

- Suitable for storing binary hashes or encrypted data

Use cases:

- Storing encrypted passwords
- File fingerprints or checksums

3.4 BLOB Types

Binary Large Object types, similar to TEXT types but for binary data:

- TINYBLOB, BLOB, MEDIUMBLOB, LONGBLOB

Backend storage: Like TEXT types, stored separately with a pointer in the table row.

Best practices:

- Use sparingly, as they can significantly impact performance
- Consider storing large files in the filesystem and keeping only the file path in the database

Use cases:

- Storing small images or documents directly in the database
- Backup data or serialized objects

4. Date and Time Data Types

4.1 DATE, TIME, and DATETIME

- DATE: 3 bytes, stores date in 'YYYY-MM-DD' format
- TIME: 3 bytes, stores time in 'HH:MM:SS' format
- DATETIME: 8 bytes, combines date and time 'YYYY-MM-DD HH:MM:SS'

Backend storage: Stored as compact binary representations.

Best practices:

- Use the appropriate type based on whether you need date, time, or both

- Be aware of time zone implications, especially with DATETIME

Use cases:

- DATE: Birth dates, event dates
- TIME: Store only time component, e.g., store hours
- DATETIME: Event timestamps, log entries

4.2 TIMESTAMP

4 bytes, stores datetime values, but automatically converts to UTC for storage and back to the current time zone for retrieval.

Backend storage: Stored as seconds since the Unix epoch.

Best practices:

- Use for tracking record creation/modification times
- Understand the automatic time zone conversion behavior

Use cases:

- Tracking when records were last updated
- Storing event times that need to be displayed in different time zones

4.3 YEAR

1 byte, stores year values from 1901 to 2155.

Best practices:

- Use when only the year is relevant to save space compared to DATE

Use cases:

- Copyright years
- Graduation years

5. Special Data Types

5.1 ENUM

A string object with a value chosen from a predefined list of permitted values.

Backend storage: Stored as integers representing the index of the chosen value.

Best practices:

- Use for columns with a fixed set of possible values
- Be cautious about changing the definition after data is inserted

Use cases:

- Status flags (e.g., 'active', 'inactive', 'pending')
- Categories with a fixed set of options

5.2 SET

Similar to ENUM, but can store multiple choices from the predefined list.

Backend storage: Stored as a bit field.

Best practices:

- Use when multiple selections from a fixed list are needed
- Be aware of the maximum limit of 64 members

Use cases:

- Multi-select options (e.g., product features, user permissions)

6. JSON Data Type

Stores JSON (JavaScript Object Notation) documents.

Backend storage: Stored in a binary format that allows for fast read access to document elements.

Best practices:

- Use for semi-structured data that doesn't fit well into relational tables
- Utilize JSON functions for efficient querying and manipulation

Use cases:

- Storing configuration settings
- Flexible attribute storage for products or user profiles

7. Spatial Data Types

MySQL supports various spatial data types for geographic information:

- GEOMETRY, POINT, LINESTRING, POLYGON, etc.

Backend storage: Uses Well-Known Binary (WKB) format.

Best practices:

- Use spatial indexes for efficient querying
- Understand the coordinate system and units you're working with

Use cases:

- Storing geographic locations, shapes, or paths
- Mapping applications, GIS systems

8. General Best Practices for Data Types
9. Choose the smallest data type that can reliably store your data.
10. Use UNSIGNED for integer columns that will never contain negative values.
11. For string columns, use VARCHAR unless you have a specific reason to use CHAR.
12. Be cautious with TEXT and BLOB types due to their performance implications.
13. Use appropriate date and time types to leverage MySQL's built-in functions and indexing capabilities.
14. Consider using ENUM or SET for columns with a predefined list of values.
15. Use JSON type for flexible schema requirements, but don't overuse it at the expense of proper relational design.
16. Always consider indexing implications when choosing data types.

17. Performance Considerations

- Smaller data types generally lead to faster queries and less I/O.
- Fixed-length types (like CHAR) can be faster for writes but may waste space.
- Large text fields (TEXT, BLOB) can slow down queries, especially if not properly indexed.
- Using appropriate data types allows MySQL to use type-specific optimizations.

10. Storage Engine Implications

Different MySQL storage engines may handle data types slightly differently:

- InnoDB: The default and recommended engine, handles all data types well.
- MyISAM: Older engine, may have some specific optimizations for certain data types.
- Memory: In-memory tables have limitations on TEXT and BLOB types.

Understanding the interplay between data types and storage engines can help in optimizing for specific use cases.

Conclusion:

Choosing the right data types is fundamental to good database design in MySQL. It affects storage efficiency, query performance, and data integrity. Always consider the nature of your data, the operations you'll perform on it, and the scale of your application when selecting data types. Regular review and optimization of data types can lead to significant improvements in database performance and storage utilization.