02 DataTypes -KirkYagami X Nikhil Sharma

MySQL Data Types

1. Numeric Data Types

Integer Types

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

Type	Storage	Min Value (Signed)	Max Value (Signed)	Min Value (Unsigned)	Max Value (Unsigned)
TINYINT	1 byte	-128	127	0	255
SMALLINT	2 bytes	-32,768	32,767	0	65,535
MEDIUMINT	3 bytes	-8,388,608	8,388,607	0	16,777,215
INT	4 bytes	-2,147,483,648	2,147,483,647	0	4,294,967,295
BIGINT	8 bytes	-2^63	2^63-1	0	2^64-1

When to use each type:

- **TINYINT**: For small number ranges like flags, small counters, or boolean values
- **SMALLINT**: For medium-sized numbers like product quantities or ages
- **MEDIUMINT**: Less commonly used, but good for larger counts that don't need INT
- **INT**: The standard choice for most IDs and counters
- **BIGINT**: For very large numbers or when future growth might exceed INT limits

Examples:

```
CREATE TABLE products (
    product_id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    product_name VARCHAR(100) NOT NULL,
    in_stock TINYINT UNSIGNED DEFAULT 0,
    quantity SMALLINT UNSIGNED DEFAULT 0,
    price DECIMAL(10,2) NOT NULL,
    is_available TINYINT(1) DEFAULT 1
);
```

In this real-world example:

- `product_id` : INT is perfect for IDs (UNSIGNED means only positive values)
- `in_stock` : TINYINT is used for a small count (0-255 items in stock)
- `quantity` : SMALLINT handles reasonable inventory counts
- `is_available` : TINYINT(1) works like a boolean (0=false, 1=true)

Decimal Types

For precise numeric values (like money):

- **DECIMAL(M,D)** or **NUMERIC(M,D)**: M = total digits, D = decimal places

- **FLOAT**: Single-precision floating-point (4 bytes)
- **DOUBLE**: Double-precision floating-point (8 bytes)

Examples:

```
CREATE TABLE financial_transactions (
    transaction_id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    amount DECIMAL(10,2) NOT NULL,
    tax_rate DECIMAL(5,4) NOT NULL,
    exchange_rate FLOAT NOT NULL,
    large_calculation DOUBLE NOT NULL
);
```

In this example:

- `amount`: DECIMAL(10,2) for currency (e.g., 12345678.90)
- `tax_rate`: DECIMAL(5,4) for percentages (e.g., 0.0825 for 8.25%)
- `exchange_rate`: FLOAT for approximate values where precision isn't critical
- `large_calculation`: DOUBLE for scientific calculations

When to use DECIMAL vs FLOAT/DOUBLE:

```
-- BAD: Using FLOAT for money (potential precision errors)
CREATE TABLE account_bad (balance FLOAT);
INSERT INTO account_bad VALUES (100.00);
INSERT INTO account_bad VALUES (100.10);
-- May not equal exactly 200.10 due to floating-point imprecision

-- GOOD: Using DECIMAL for money (exact precision)
CREATE TABLE account_good (balance DECIMAL(10,2));
INSERT INTO account_good VALUES (100.00);
INSERT INTO account_good VALUES (100.10);
-- Will equal exactly 200.10
```

2. String Data Types

Fixed-Length vs Variable-Length

- **CHAR(N)**: Fixed-length strings (always uses N bytes)
- **VARCHAR(N)**: Variable-length strings (uses only what's needed plus 1-2 bytes)

Examples:

```
CREATE TABLE users (
    user_id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    username VARCHAR(50) NOT NULL,
    password_hash CHAR(64) NOT NULL,  -- SHA-256 hash is always 64 chars
    postal_code CHAR(5),              -- US ZIP codes are always 5 chars
    bio VARCHAR(500)                  -- Variable user descriptions
);
```

This example shows:

- `username`: VARCHAR because usernames have varying lengths

- `password_hash`: CHAR because cryptographic hashes have fixed length
- `postal_code`: CHAR for data with known, fixed length
- `bio`: VARCHAR for longer text with varying lengths

Text Types (for larger text):

Type	Maximum Length	Storage Required
TINYTEXT	255 bytes	Length + 1 byte
TEXT	65,535 bytes	Length + 2 bytes
MEDIUMTEXT	16 MB	Length + 3 bytes
LONGTEXT	4 GB	Length + 4 bytes

Example:

```
CREATE TABLE blog_posts (
    post_id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    title VARCHAR(200) NOT NULL,
    summary TEXT,
    content MEDIUMTEXT,
    meta_keywords VARCHAR(255)
);
```

In this blog system:

- `title`: VARCHAR for shorter, variable-length titles
- `summary`: TEXT for paragraph-length summaries
- `content`: MEDIUMTEXT for full blog posts (could be very long)
- `meta_keywords`: VARCHAR for a list of keywords

Binary String Types:

- **BINARY(N)** and **VARBINARY(N)**: Like CHAR/VARCHAR but for binary data
- **BLOB** types: Binary equivalent of TEXT types

Example:

```
CREATE TABLE files (
    file_id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    file_name VARCHAR(255) NOT NULL,
    file_data MEDIUMBLOB,
    file_hash BINARY(32)  -- MD5 hash (always 32 bytes)
);
```

3. Date and Time Data Types

Type	Format	Range	Storage
DATE	YYYY-MM-DD	1000-01-01 to 9999-12-31	3 bytes
TIME	HH:MM:SS	-838:59:59 to 838:59:59	3 bytes
DATETIME	YYYY-MM-DD HH:MM:SS	1000-01-01 00:00:00 to 9999-12-31 23:59:59	8 bytes
TIMESTAMP	YYYY-MM-DD HH:MM:SS	1970-01-01 00:00:01 to 2038-01-19 03:14:07	4 bytes

Type	Format	Range	Storage
YEAR	YYYY	1901 to 2155	1 byte

Examples:

```
CREATE TABLE appointments (
    appointment_id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    patient_name VARCHAR(100) NOT NULL,
    appointment_date DATE NOT NULL,
    appointment_time TIME NOT NULL,
    created_at DATETIME DEFAULT CURRENT_TIMESTAMP,
    last_modified TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP
);
```

In this appointment system:

- `appointment_date` : DATE stores just the date (YYYY-MM-DD)
- `appointment_time` : TIME stores just the time (HH:MM:SS)
- `created_at` : DATETIME captures when the record was created
- `last_modified` : TIMESTAMP automatically updates when the record changes

DATETIME vs TIMESTAMP:

- TIMESTAMP values are converted from the current time zone to UTC for storage and back to the current time zone for retrieval
- TIMESTAMP uses less storage but has a limited range (1970-2038)
- DATETIME has a wider range but doesn't handle time zones automatically

```
-- Example showing automatic updates with TIMESTAMP
CREATE TABLE audit_log (
    log_id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    action VARCHAR(50) NOT NULL,
    created_at DATETIME DEFAULT CURRENT_TIMESTAMP,
    system_time TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

4. Enumeration and Set Types

ENUM

ENUM allows you to specify a list of possible values for a column, storing data very efficiently.

```
CREATE TABLE surveys (
    survey_id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    question VARCHAR(255) NOT NULL,
    response ENUM('Strongly Disagree', 'Disagree', 'Neutral', 'Agree', 'Strongly Agree')
);

-- How to insert using ENUM
INSERT INTO surveys (question, response) VALUES
('I found the product easy to use', 'Agree');
```

SET

SET is like ENUM but allows multiple values from the predefined list:

```
CREATE TABLE users_preferences (
    user_id INT UNSIGNED PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    notifications SET('email', 'sms', 'push', 'in_app') DEFAULT 'email',
    favorite_categories SET('electronics', 'books', 'clothing', 'food', 'sports')
);

-- Store multiple values in a SET
INSERT INTO users_preferences (user_id, name, notifications, favorite_categories)
VALUES (1, 'John Smith', 'email,sms', 'electronics,books');

-- Query to find users who want email notifications
SELECT * FROM users_preferences WHERE FIND_IN_SET('email', notifications) > 0;
```

5. JSON Data Type

MySQL 5.7+ supports a native JSON data type for storing and working with JSON documents:

```
CREATE TABLE product_details (
    product_id INT UNSIGNED PRIMARY KEY,
    basic_info JSON,
    tech_specs JSON
);

-- Inserting JSON data
INSERT INTO product_details VALUES (
    1001,
    '{"name": "Smartphone X", "color": "Black", "storage": "128GB"}',
    '{"screen": "6.5 inch", "battery": "4000mAh", "camera": "48MP"}'
);

-- Extracting values from JSON
SELECT
    product_id,
    JSON_EXTRACT(basic_info, '$.name') AS product_name,
    JSON_EXTRACT(tech_specs, '$.camera') AS camera
FROM product_details;

-- Using the -> operator (shorthand for JSON_EXTRACT)
SELECT
    product_id,
    basic_info->'$.name' AS product_name
FROM product_details;
```

6. Spatial Data Types

MySQL supports GIS (Geographic Information System) data types:

- **GEOMETRY**: Base type
- **POINT**: Single x,y coordinate
- **LINESTRING**: Line of points
- **POLYGON**: Area defined by points
- **MULTIPOINT/MULTILINESTRING/MULTIPOLYGON**: Collections of the above

```sql
CREATE TABLE stores (
    store_id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    store_name VARCHAR(100) NOT NULL,
    location POINT NOT NULL,
    delivery_area POLYGON
);

-- Adding a spatial index
CREATE SPATIAL INDEX idx_location ON stores(location);

-- Inserting point data
INSERT INTO stores (store_name, location) VALUES
('Downtown Store', ST_GeomFromText('POINT(40.7128 -74.0060)'));

-- Finding stores within a certain distance (e.g., 5km)
SELECT store_name FROM stores
WHERE ST_Distance_Sphere(location, ST_GeomFromText('POINT(40.7200 -74.0100)')) <= 5000;
```

7. Special Data Types

BIT

The BIT data type is used to store bit field values:

```sql
CREATE TABLE system_settings (
    setting_id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    setting_name VARCHAR(50) NOT NULL,
    flags BIT(8) DEFAULT b'00000000'
);

-- Setting the 1st and 3rd bits
INSERT INTO system_settings (setting_name, flags) VALUES
('Security Settings', b'00000101');
```

8. Practical Examples

Example 1: E-commerce Database

```sql
CREATE TABLE customers (
    customer_id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    first_name VARCHAR(50) NOT NULL,
    last_name VARCHAR(50) NOT NULL,
    email VARCHAR(100) UNIQUE NOT NULL,
    phone CHAR(10),
    birth_date DATE,
    membership_level ENUM('Bronze', 'Silver', 'Gold', 'Platinum') DEFAULT 'Bronze',
    is_active TINYINT(1) DEFAULT 1,
    created_at DATETIME DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE orders (
    order_id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    customer_id INT UNSIGNED NOT NULL,
    order_date DATETIME DEFAULT CURRENT_TIMESTAMP,
    total_amount DECIMAL(10,2) NOT NULL,
```

```
    shipping_address JSON NOT NULL,
    payment_method ENUM('Credit Card', 'PayPal', 'Bank Transfer') NOT NULL,
    order_status ENUM('Pending', 'Processing', 'Shipped', 'Delivered', 'Cancelled') DEFAULT
'Pending',
    FOREIGN KEY (customer_id) REFERENCES customers(customer_id)
);

CREATE TABLE products (
    product_id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    sku CHAR(8) UNIQUE NOT NULL,
    name VARCHAR(100) NOT NULL,
    description TEXT,
    price DECIMAL(10,2) NOT NULL,
    stock_quantity SMALLINT UNSIGNED DEFAULT 0,
    categories SET('Electronics', 'Clothing', 'Home', 'Books', 'Sports') NOT NULL,
    specifications JSON,
    image_data MEDIUMBLOB,
    is_featured TINYINT(1) DEFAULT 0,
    created_at DATETIME DEFAULT CURRENT_TIMESTAMP,
    last_updated TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP
);
```

Example 2: Blog Platform

```
CREATE TABLE blog_users (
    user_id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    username VARCHAR(50) UNIQUE NOT NULL,
    password_hash CHAR(60) NOT NULL, -- For BCrypt hash
    email VARCHAR(100) UNIQUE NOT NULL,
    bio TEXT,
    avatar MEDIUMBLOB,
    role ENUM('Reader', 'Author', 'Editor', 'Admin') DEFAULT 'Reader',
    joined_date DATETIME DEFAULT CURRENT_TIMESTAMP,
    last_login TIMESTAMP
);

CREATE TABLE blog_posts (
    post_id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    author_id INT UNSIGNED NOT NULL,
    title VARCHAR(200) NOT NULL,
    slug VARCHAR(200) UNIQUE NOT NULL,
    content MEDIUMTEXT NOT NULL,
    excerpt TEXT,
    status ENUM('Draft', 'Published', 'Archived') DEFAULT 'Draft',
    comment_status ENUM('Open', 'Closed') DEFAULT 'Open',
    published_at DATETIME,
    created_at DATETIME DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
    tags JSON,
    views INT UNSIGNED DEFAULT 0,
    FOREIGN KEY (author_id) REFERENCES blog_users(user_id)
);

CREATE TABLE comments (
    comment_id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    post_id INT UNSIGNED NOT NULL,
    author_name VARCHAR(100) NOT NULL,
```

```
    author_email VARCHAR(100) NOT NULL,
    author_ip VARCHAR(45) NOT NULL,
    content TEXT NOT NULL,
    is_approved TINYINT(1) DEFAULT 0,
    parent_id INT UNSIGNED DEFAULT NULL,
    created_at DATETIME DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (post_id) REFERENCES blog_posts(post_id),
    FOREIGN KEY (parent_id) REFERENCES comments(comment_id)
);
```

9. Best Practices for Choosing Data Types

1. **Use the smallest data type that can reliably store your data**
   - If a SMALLINT can handle your range of values, don't use INT
   - This saves storage space and improves performance
2. **Be cautious with variable-length types**
   - VARCHAR is efficient for variable-length data
   - But CHAR is faster for fixed-length data
3. **Consider future growth**
   - Will this user ID column ever exceed 2 billion records? If possible, use BIGINT
   - Will product names ever exceed 100 characters? Maybe use VARCHAR(255) instead
4. **Use DECIMAL for money**
   - Never use FLOAT or DOUBLE for currency or precise calculations
   - DECIMAL(10,2) is common for standard currency values
5. **Choose the right TEXT/BLOB type**
   - Don't use LONGTEXT if TEXT will suffice
   - Large TEXT/BLOB fields can impact performance
6. **Use ENUM and SET for constrained values**
   - When values come from a fixed list, ENUM is very efficient
   - Consider SET when multiple selections are possible
7. **Consider normalization**
   - Storing JSON or complex data might be convenient but can hinder queries
   - Consider breaking out frequently searched attributes into separate columns
8. **Use specialized types appropriately**
   - For spatial data, use spatial types rather than storing coordinates in separate columns
   - For dates and times, use appropriate types instead of storing as strings
9. **Plan for NULL values**
   - Decide whether each column should allow NULL values
   - Use DEFAULT values instead of NULL when appropriate