Discussion-2 The importance of data types when designing a relational database

<u>Discussion Topic:</u>

Discuss the importance of data types when designing a relational database. Give an example.

<u>My Post:</u>

Hello Class,

In relational databases, information is organized in tables referred to as schemas, the table rows act as records, and the columns act as attributes. To manage and manipulate these tables, developers use Structured Query Language (SQL), with MySQL being the most popular relational database management system (RDBMS). In the context of MySQL, selecting the appropriate data types for table columns, in other words, the data attributes, is essential for the RDBMS and the applications relying on it, as it directly impacts data integrity, storage efficiency, query performance, and the accuracy of data operations like calculations and comparisons.

MySQL Data Types

In the context of MySQL, data types define the nature of the information that can be accepted and stored within each specific column of a table. MySQL provides a wide range of data types, each designed to fit a specific kind of data characteristics, such as storage space and allowed values. See the table below for more information.

Table 1
MySQL Data Types

Category	Data Type	Description/Purpose	Key Details / Examples (from text)
Numeric: Integer	TINYINT	Smallest integer type.	1 byte. Range: -128 to 127 (signed), 0 to 255 (unsigned).
	SMALLINT	Small integer type.	2 bytes. Range: -32768 to 32767 (signed), 0 to 65535 (unsigned).
	MEDIUMINT	Medium-sized integer type.	3 bytes. Range: -8388608 to 8388607 (signed), 0 to 16777215 (unsigned).
	INT	Standard integer type.	4 bytes. Range: -2147483648 to 2147483647 (signed), 0 to 4294967295 (unsigned).
	BIGINT	Large integer type.	8 bytes. Range: -2^63 to 2^63-1 (signed), 0 to 2^64-1 (unsigned).

	SERIAL	Alias for common primary key properties.	Equivalent to BIGINT UNSIGNED NOT NULL AUTO_INCREMENT UNIQUE. Auto-assigns unique values.
Numeric: Fixed-Point	DECIMAL(p, s)	Stores exact decimal numbers with defined precision and scale. Synonymous with NUMERIC.	p = total digits (precision), s = digits after decimal (scale). Default DECIMAL(10, 0). Use for exact calculations (e.g., currency). NUMERIC(5, 2)
	NUMERIC(p, s)	Stores exact decimal numbers. Synonymous with DECIMAL.	Same as DECIMAL.
Numeric: Floating-Point	FLOAT(p)	Stores approximate decimal numbers (single-precision).	4 bytes. Variable precision (0-23 digits). p limits display digits. Faster than NUMERIC, but potential rounding differences.
	DOUBLE(p)	Stores approximate decimal numbers (double-precision).	8 bytes. Variable precision (24-53 digits). Faster than NUMERIC, but potential rounding differences.
Numeric: Bit Value	BIT	(Mentioned in list, not detailed in text).	Stores bit values.
Date & Time	DATE	Stores date only.	Format: YYYY-MM-DD. Range: 1000-01-01 to 9999-12-31.
	TIME	Stores time of day only.	Format: hh:mm:ss. Can include fractional seconds (up to 6 digits). Range: -838:59:59.000000 to +838:59:59.000000.
	DATETIME	Stores combination of date and time.	Format: YYYY-MM-DD hh:mm:ss. Range: 1000-01-01 00:00:00 to 9999-12-31 23:59:59. Can include fractional seconds. Does not handle time zones automatically.
	TIMESTAMP	Stores combination of date and time, tracking a specific moment.	Range: 1970-01-01 00:00:01 UTC to 2038-01-19 03:14:07 UTC. Converts to/from UTC for storage/retrieval. Can include fractional seconds and timezone offset.
	YEAR	(Mentioned in list, not detailed in text).	Stores a year value.

String: Fixed Length	CHAR(n)	Fixed-length character string.	n = number of characters. CHAR alone is 1 character. Pads with spaces if input is shorter than n.
	BINARY(n)	Fixed-length binary string (byte string).	n = number of bytes. Pads with null bytes. Comparisons are based on byte values.
String: Variable Length	VARCHAR(n)	Variable-length character string.	n = *maximum* number of characters. Does not pad. Requires n.
	VARBINARY(n)	Variable-length binary string (byte string).	n = *maximum* number of bytes.
String: Large Objects	BLOB	For large binary objects (Binary Large Object).	Similar to VARBINARY but for larger data. Cannot have default values. Indexing requires prefix length.
	TEXT	For large character strings.	Similar to VARCHAR but for larger data. Cannot have default values. Indexing requires prefix length.
Boolean	BOOL/BOOLEAN	Not a native type; alias for TINYINT(1).	0 is considered false, non-zero is true. TRUE literal stored as 1, FALSE as 0.
Enumerated/Set	ENUM()	String object where the value must be *one* chosen from a predefined list of values.	Example: season ENUM('winter', 'spring', 'summer', 'autumn').
	SET()	String object where the value can be *zero or more* chosen from a predefined list.	Example: availability SET('sun', 'mon', 'tue', ...). Input/Output as comma-separated string. Duplicates removed, order follows definition.
Spatial	GEOMETRY, POINT, LINESTRING, POLYGON, MULTIPOINT, MULTILINESTRING, MULTIPOLYGON, GEOMETRYCOLLECTION	For storing geographical data.	POINT is built for coordinates.
JSON	JSON	Stores JSON documents.	Stored in binary format for faster access.

Note: The table lists the different MySQL. Data from “An introduction to MySQL data types” by Ellingwood (n.d.).

The Importance of Data Type Selection

Selecting the right data type in RDBMS is very important as it ensures that the data is stored efficiently and accurately (Sakshijain1, 2025). It impacts several aspects of RDBMS functionality, such as data integrity, storage, and query, as well as operations such as *SUM* and *AVG* calculation performed on numeric types, or *DATEDIFF* and *DATE_ADD* operations performed on temporal data types.

The following examples illustrate the impact of selecting appropriate data types versus inappropriate ones.

MySQL script below creates an events table to store concert dates:

```
CREATE TABLE Events (
    event_id INT AUTO_INCREMENT PRIMARY KEY,
    event_name VARCHAR(100),
    event_start DATETIME,     -- Using DATETIME type
    ticket_price DECIMAL(8, 2) -- Using DECIMAL for currency
);

INSERT INTO Events (event_name, event_start, ticket_price) VALUES
('Spring Gala', '2025-05-15 19:00:00', 75.50),
('Summer Concert', '2025-07-20 20:30:00', 45.00),
('Autumn Workshop', '2025-10-10 09:00:00', 120.00);
```

Note that the column *event_start* was defined having a DATETIME data type, which allows performing temporal operations, for example, finding events happening in the next 90 days.

```
SELECT event_name, event_start
FROM Events
WHERE event_start BETWEEN CURDATE() AND DATE_ADD(CURDATE(), INTERVAL 90 DAY);
```

Another example is using DECIMAL(8, 2) for *ticket_price* allows calculating the average ticket price without potential floating-point rounding errors:

```
SELECT AVG(ticket_price) FROM Events;
```

Now, if *event_start* were stored as *VARCHAR(20)* (e.g., 'May 15, 2025 7PM'), using a function similar to the *DATE_ADD* would be very difficult and prone to errors due to the text formats. Similarly, storing *ticket_price* as *VARCHAR* (e.g., ‘$75.50’, ‘$40.25’) and when using functions like *AVG()* or *SUM()*. Although MySQL implicitly converts the VARCHAR to the *DOUBLE* type, the output of the conversion is not always what is expected, as *AVG(ticket_price)* (calculating ticket price average) will compute as *0.0*. This happens because MySQL will read the first character for the values, '*$*' (which is not a number), and interpret the entire expression as not being numeric and translate *VARCHAR* data type values to the *DOUBLE* data type value of *0.0*.

For example:

When using a column:

```
score_text VARCHAR(20),
```

with the column is storing the following data:

```
‘100'
'p85'
'Failed'
'90 points'
'70'
```

and performing the following operation on the column (Computing scores average):

```
SELECT AVG(score_text)
```

The following MYSQL code line will first convert the VARCHAR data to DOUBLE data:

```
100.0  -- ‘100’
0.0   -- ‘p85’
0.0    -- ‘Failed’
90.o   --’90 Points’
70.0   -- ‘70’
```

Note that MySQL will generate warnings for the rows containing *'Failed'* and *'90 points'*.
And then it will performed the following calculation (*100.0 + 0.0 + 0.0 + 90.0 + 70.0) / 5 = 345.0 / 5 = 52.0*

This average output is probably not what was expected, or it is meaningful, because the non-numeric value *'Failed'* was converted to *0*, lowering the calculated average compared to the average of only the actual numerical scores.

-Alex

References:

Ellingwood, J. (n.d.). An introduction to MySQL data types.
https://www.prisma.io/dataguide/mysql/introduction-to-data-types

Sakshijain1 (2025, January 30). SQL data types. GeeksForGeeks. https://www.geeksforgeeks.org/sql-data-types/