SQL and MySQL Data Types

Databases have tables, which have columns, which have types and other properties. This chapter summarises MySQL column data types and properties.

SQL data types and MySQL

Look in any computer language manual and you will find a list of data types like this:

- *Character stream, or string*: one or more alphanumeric characters likely to be meaningful or printable in text, e.g., a name.
- *Binary stream*: a byte stream that may encode text or other information *e.g.*, a digital photo, engineering drawing, or word processing document.
- *Number*: a negative or positive numeric value, small or large, rational or irrational, for example 27, -1, 3.14159, 6.28 + 3i. A numeric subtype of special interest is the two-valued numeric called boolean (True/False, Yes/No, Living/Dead, or whatever is appropriate to the problem domain).
- *Datetime*: dates, times, timestamps.

Databases are concerned at the most fundamental level with data types, because every column of every table must be of a predefined base type or user-defined type (UDT) derived from a base type. UDTs are not yet available in MySQL.

SQL92 defines a set of base types. Every SQL92 implementation delivers a slightly different subset of these types. In designing your tables, choose types that best fit your requirements in the interests of efficiency and performance. MySQL offers these:

- *Character string*, or string for short: a sequence of characters, as short as the initial 'A', or as long as a sequence of four billion characters; MySQL has nine string column types: CHAR, VARCHAR, NVARCHAR, four TEXT types, ENUM, SET.
- *Binary stream* or object: a character stream without optimisation for rendering as text. MySQL has two BINARY and four Binary Large Object (BLOB) types. Since version 5.7.8 MySQL also supports the structured binary JSON type allowing use of MySQL as a *document store* of collections (containers) of JSON-encoded documents rather than as a relational database.
- *Numeric*: representation of a number as an integer, a floating point value, or a value to a fixed decimal precision; MySQL has 11 numeric data types: BIT, TINYINT, SMALLINT, MEDIUMINT, INT, BIGINT, three FLOATs, DOUBLE, DECIMAL.
- *Datetime*: a value representing the time in milliseconds since a reference date. MySQL has five datetime types: DATE, TIME, DATETIME, TIMESTAMP, and YEAR.
- *OpenGIS*: types for representing spherical geometric values.

Column type modifiers in MySQL

When you *CREATE* or ALTER a column, you may also specify properties the MySQL manual calls *field attributes*, for example UNSIGNED for only positive values, or NULL to accept null values. But *attribute* is a formal SQL synonym for *column*, so using it also for column properties like NULL makes them attributes of attributes. Confusing at best. We refer to these properties as *type modifiers*, because that's what they do.

The general column type modifiers are

- NOT NULL: the column rejects NULL values, *e.g.*, `lastname` CHAR(20) NOT NULL,
- NULL: the column accepts NULL values, for example `middlename` CHAR(2) NULL,
- DEFAULT `x`: default column value=`x`, *e.g.*, `cntry_code CHAR(2) DEFAULT 'US'`.
- UNSIGNED: numeric type accepts values >= 0 only.
- AUTO_INCREMENT: in numeric columns only, automatically assign the next available value to a column in a new row
- *Maximum display width*: (N) after the name where N limits display width, *e.g.*, `qty INT(4)`. See *Chapter 6* (CREATE TABLE) for display width modifier syntax. The feature is deprecated as of version 8.0.17.

MySQL string types

Before version 5.7.8, MySQL had four broad string datatypes: *char*, *binary*, *hybrid*, *enumerated*; 5.7.8 added the JSON datatype. [N][VAR]CHAR strings use character sets and collations; [N] variants use the NATIONAL charset, `utf8` in MySQL. BINARY types are byte strings with no character set, no collation; comparison and sorting are bytewise. Hybrid strings ([VAR]CHAR BINARY) use character sets and corresponding BINARY (`_bin`) collations. Enumerated strings (ENUM, SET) store single or multiple predefined strings. JSON strings are `utf8mb4` and `utf8mb4_bin`. See Table 4-1.

MySQL character and binary columns have *four* lengths:

- *maximum character length*, the largest number of characters the column can store,
- *actual character length* of any one string in one cell,
- *character storage required* for a column in a given row,
- *byte storage required*, character storage * bytes-per-character for the charset.

In CHAR and BINARY columns, the first three lengths are the same. That is, the column definition statement `name CHAR(50)` determines that every `name` cell will be exactly 50 characters long, and `data BINARY(50)` determines that every `data` cell will be exactly 50 bytes long irrespective of character set. CHAR and BINARY columns are *fixed-length*. But VARCHAR, VARBINARY, TEXT and BLOB columns are *variable-length*: actual length depends on the number of characters stored. *Table 4-1* lists MySQL string types.

Length and silent column specification changes

String and binary storage length variability affects performance. The MySQL server can compute the actual position of any given cell in a table with no variable-length string columns simply by multiplying row count and column size. Very quick.

Character column length

But in a table with VARCHAR, VARBINARY, TEXT or BLOB columns, the DBMS must compute cell position from unsigned length integers stored with the data in each variable-length cell. The CHAR column padding no longer has a purpose, and it slows access, so if you specify both a CHAR column and a VARCHAR(n>3), VARBINARY, TEXT or BLOB column, MySQL silently coerces the CHAR column to VARCHAR. Thus if you write

```
CREATE TABLE clientpics( name CHAR( 50 ), picture MEDIUMBLOB );
DESCRIBE client;
```

you will see that MySQL has automatically changed your CHAR column to VARCHAR.

Table 4-1: String and binary column types in MySQL

Type Name	*Max length*	*Meaning*	*Required chars / row*	*Trimmed on retrieval*	*Case-sensitive sort*
CHAR(n)	n, <= 255	<= n chars	n	Yes	With BINARY operator
NCHAR(n)	n, <= 255	<= n chars utf8 charset	n	Yes	With BINARY op
VARCHAR(n)	n, <= 65532 (255 pre 5.0.3)	<= n chars	Actual length + 1 or 2	Yes to 5.0.2, then No	With BINARY op.
NVARCHAR(n)	like VARCHAR	VARCHAR(n) utf8 charset	like VARCHAR	like VARCHAR	like VARCHAR
BINARY(n) (since 4.1.2)	n, <= 255 bytes	<= n bytes padded with \0. CHAR BYTE(1) ≡ BINARY(1)	n	No	Yes
VARBINARY(n) (since 4.1.2)	n, <= 65532 (255 pre 5.0.3)	<= n bytes, no padding	Actual length +1 or 2	Yes to 5.0.2, then No	Yes
TINYTEXT	255	<= 255 chars	Actual length+1	No	With BINARY op.
TEXT	65,535	<= 65,535 chars	Actual length+2	No	With BINARY op.
MEDIUMTEXT	16,777,215	<= 16,777,215 chars	Actual length+3	No	With BINARY op.
LONGTEXT	4,294,967,295	<= 4,294,967,295 chars	Actual length+4	No	With BINARY op.
TINYBLOB	255	<= 255 bytes	Actual length+1	No	Yes
BLOB	65,535	<= 65,535 bytes	Actual length+2	No	Yes
MEDIUMBLOB	16,777,215	<= 16,777,215 bytes	Actual length+3	No	Yes
LONGBLOB	4,294,967,295	<= 4,294,967,295 bytes	Actual length+4	No	Yes
ENUM ("val1","val2",...)	65,535	Enumerated strings, one value / cell	1 or 2	--	--
SET ("val1","val2",...)	65,535	Enumerated strings, multiple values / cell	1, 2, 3, 4 or 8	--	--
JSON	max_allowed_ packet	Determined internally	Determined internally	No	No

On the other hand, MySQL will coerece VARCHAR columns with lengths of four bytes or less in the opposite direction, from VARCHAR to CHAR, because the better performance of fixed-length storage offsets the tiny extra space cost of padding short strings..

Since the CHAR column type is so much faster to read and write, is it preferable to VARCHAR? All other things being equal, yes, though if you must store large variable amounts of text, you need a VARCHAR or TEXT column (VARCHAR is a little faster, and supports DEFAULT). Even in this case there may be an opportunity for optimisation. If a subset of columns will often be queried without reference to variable-length columns, performance may be better with two key-related tables: one for the fixed-length columns, one for the variable-length columns.

Maximum string length

With CHAR, VARCHAR, BINARY and VARBINARY, you specify maximun column length when you create or alter the table. Maximum lengths of other string and binary columns are predefined. For example in

```
CREATE TABLE clientpics( clientID INTEGER, picture MEDIUMBLOB );
```

the `picture` column will store an image up to 16,777,215 bytes long because that is the MEDIUMBLOB specification. The actual amount of storage used by all string columns except CHAR, however, varies from row to row. The storage occupied at any one time by a column in any given row will be the length of data stored in that cell, plus 1-4 bytes.

BLOB and TEXT column types

There are four TEXT types and four corresponding BLOB types with maximum lengths of 255, 65,535, 16,777,215 and 4,294,967,295 bytes. The difference between a TEXT and BLOB column of the same size is that the BLOB column sorts case-sensitively. Beware that MySQL has a hard `max_allowed_packet` limit of 1GB *bytes*, so larger LONGBLOB/TEXT values need to be chunked for transmission; therefore it's more efficient to store pointers to such files in the table, and store the actual file drectly on disk.

MySQL has supported FULLTEXT indexing and searching for VARCHAR and TEXT columns in MYISAM tables since version 3.23 and in INNODB tables since 5.6.4 (*Chapter 7*). Specify it with the FULLTEXT keyword in CREATE | ALTER TABLE and CREATE INDEX (*Chapter 6*), and do full-text searches with the *MATCH function*.

TEXT comparisons are case-insensitive; use the BINARY operator for case sensitivity, as in `SELECT ... WHERE BINARY mytext=…`. To force case insensitivity in a comparison with a BLOB, use `CONVERT(colname USING latin1) COLLATE latin1_ swedish_ci` (see CAST, CONVERT in *Chapter 8*, and Character sets and collations in *Chapter 3*)

ENUM and SET column types

ENUM and SET types predefine lists of allowable string values. They are string types because their specifications are strings, but MySQL stores them as unsigned integers for compactness and efficiency, and can return integer values from them.

The ENUM column is stored an an unsigned SMALLINT, so it permits a list of up to 65,535 permissible strings (preferably from a category that relates to the table). In practice no-one ever approaches 65,535—long before that, you'll have to move the ENUM list into its own table, creating a foreign key that references it.

The advantage of an ENUM column is that MySQL automatically handles the range for you. An ENUM column defined as `ENUM( 'Strongly Agree', 'Agree', 'Do not Know', 'Disagree', 'Strongly Disagree')` would accept only one of those values and no others, no coding required.

SET columns are very similar, except that you may specify more than one item from the list--to a maximum of 64 since SETs are stored in TINYINTs, and there is the controversy: SET columns violate a basic principle of relational theory, the rule of atomic values.

Is the violation merely formal? Why not numeric vectors too? Indeed some OPENGIS types are vectors in all but name. Version 5.7.8 introduced JSON structure storage. Relational "laws" aren't absolute. We suggest you use SET columns for non-exclusive selections from small categories.

For ENUM and SET columns,

- column values can be set and retrieved as strings or as the unsigned integers that MySQL encodes them with,
- when you display them as strings, declaration case determines display case, and
- declaration order determines both sort and display order.

For ENUM columns,

- values are assigned unsigned integer values sequentially beginning with 1, and
- Setting an ENUM column to an unsupported value sets the column value to an empty string with numeric value 0.

For SET columns,

- set the column to multiple values by writing them as '`val1,val2,...`',
- bit 0 represents the first set member, bit 1 the second, up to 64 bits, and
- MySQL ignores assignments of unsupported values to a SET column.

Here is an example:

```
CREATE TABLE paytypes (
  paytypeID INTEGER NOT NULL,
  paytype ENUM( "CASH", "CREDIT", "CHECK", "MONEY ORDER" ),
  journal SET( "distribute", "defer", "partnerdiscount" )
);
INSERT INTO paytypes VALUES(1, "CASH", 0), (2, "CREDIT", "PARTNERDISCOUNT" ),
  (3,"CREDIT CARD","distribute,defer");
SELECT paytype,paytype+0,journal,journal+0 FROM paytypes;
+-------------+-----------+------------------+-----------+
| paytype     | paytype+0 | journal          | journal+0 |
+-------------+-----------+------------------+-----------+
| CASH        |         1 |                  |         0 |
|             |         0 | partnerdiscount  |         4 |
| CREDIT CARD |         3 | distribute,defer |         3 |
+-------------+-----------+------------------+-----------+
```

The ENUM column `paytype` accepts one of four values, and the SET column `journal` accepts up to three values. Either can be displayed as a string or as an integer. MySQL ignores attempts to assign an undefined value to either column.The order in which you specify the list items dictates the sort order of the ENUM or SET column. Sometimes the desired ordering of data is irrational. You might want a list of Canadian provinces to appear in a geographical sequence, say, east to west; this would be difficult with either strings or numbers, but you could do it with an ENUM column.

Should you ever need to know the actual stored value in an ENUM column, you can retrieve it by dong arithmetic on it:

```
SELECT enumColumn + 0 FROM table_name;
```

ENUM can provide a convenient solution for making legacy data more efficiently query-able. For example, if you inherit a system with a table storing the month portion of dates as three-character month names, the following simple table change

```
ALTER TABLE legacyTable MODIFY month
ENUM('JAN','FEB','MAR','APR','MAY','JUN','JUL','AUG','SEP','OCT','NOV','DEC');
```

breaks no old queries, yet permits automatic numeric ordering, so

```
SELECT * from legacyTable ORDER BY year, month, day;
```

displays the three-character month name, yet sorts `month` by number.

JSON column type

Version 5.7.8 introduced JSON columns for optimised access to structured data, A JSON *object* is a comma-separated list of key-value pairs inside curly braces, *e.g.*, `{"name":"Paavo","age":34}`. A JSON *array* is a comma-separated list of JSON values inside square brackets, *e.g.*, `[17,"Bob",1,null,'2015-8-1']`. Yes this data type also breaks the atomicity rule.

The MySQL JSON implementation treats single-quoted strings erratically, so use double quotes around strings. JSON objects and arrays may contain other JSON objects or arrays; MySQL numeric, date time, string or binary scalars; JSON booleans or nulls; and/or raw bit collections (OPAQUE). Before storing a JSON string, MySQL validates and normalises it, discarding duplicate keys and their associated values if any. Verify JSON value validity with JSON_ TYPE() before storing it. Strings are `utf8mb4` with `utf8mb4_bin` collation. Beware: column data size is limited by the `max_allowed_packet` system variable.

JSON values may be compared with =, <, <=, >, >=, <>, !=, and <=>. They're structured, so we need a syntax for finding a path to what we're looking for inside them:

- *$* refers to the target JSON value or "document"; all paths begin with it
- *.keyname* refers to the object member named *keyname*
- *[n]* immediately after a path selectng an array refers to the *n*th array element
- *name:value* denotes a key:value pair
- *.[*]* refers to all document members, *[*]* to all array members
- *prefix**suffix* refers to all paths beginning with *prefix* and ending with *suffix*

Sort precedence is BLOB, BIT, OPAQUE, DATETIME, TIME, DATE, BOOLEAN, ARRAY, OBJECT, STRING, INTEGER, DOUBLE, NULL. Within a given precedence, smaller sorts ahead of larger, shorter ahead of longer. For comparison, non-JSON values are converted to JSON. Comparison with NULL yields UNKNOWN. For aggregation, non-NULL values are converted to a numeric type. For JSON processing functions see *Chapter 8*.

MySQL numeric column types

Table 4-2 sets out the numeric data types available in MySQL. When you look at so many numeric column specifications, you may wonder, why so many? Why such bizarre boundaries? Who thinks up this weird stuff?

A bit is the smallest unit a computer can handle, the binary digit, 0 or 1. This fine grain of detail distinguishes only 0 from 1, false from true. MySQL supports BIT columns, but they're not single bits—rather they're *bit strings* written as `b'value'`, *e.g.*, `b'00101'`.

A TINYINT column occupies a single byte, which can store 256 values. Why 256? A byte is a series of 8 bits. The first bit can be in one of two states, for each of which the second bit can be in one of two states ... and so on for eight bits:

```
00000000
00000001
```

```
00000010
00000011
00000100
...
11111111
```

so there are or 2^8 possible values, 0 through 2^8-1=255. Add a second byte to give 16 bits, and we can store 2^{16} or *65,536* values in the SMALLINT. Add a third byte and we are at 2^{24}, or *16,777,216*, the MEDIUMINT. Add 8 more bits and we have 2^{32} or *4,2949,4967,296* values in the INTEGER datatype. Add four more bytes gets us the BIGINT with 2^{128}= *18,446,744,073,709,551,616* possible values.

A DECIMAL stores decimal numbers exactly (see below). A FLOAT (4 bytes) or DOUBLE (8 bytes) can store a wider numeric range per byte, but at the cost of precision.

When the numeric type supports the type modifier UNSIGNED, using it restricts all possible values to non-negative numbers. Otherwise the range centres round zero.

Numbers can store a vast range of values in a very small space, so database designers rely on them heavily, particularly for indexes: the smaller the file, the quicker the search.

Table 4-2: Numeric column types in MySQL

Type Name	*Meaning*	*Signed range*	*Unsigned range*	*Bytes*
BIT(n)	n bits		0 to 2^{n-1}, n up to 64	1-2
TINYINT [BOOL=TINYINT(1)]	one-byte integer	-128 to 127	0 to 255	1
SMALLINT	two-byte integer	-32,768 to 32,767	0 to 65,535	2
MEDIUMINT	three-byte integer	-8,388,608 to 8,388,607	0 to 16,777,215	3
INT	four-byte integer	-2,147,483,648 to 2,147,483,647	0 to 4,294,967,295	4
BIGINT	eight-byte integer	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	0-18,446,744,073,709,551,615	8
FLOAT, FLOAT(4)	four-byte float	Smallest non-zero values ±1.175494351e-38 Largest non-zero values ±3.402823466e+38		4
DOUBLE, REAL FLOAT(8),	double precision	Smallest non-zero values ±2.225073858507201e-308 Largest non-zero values ±1.797693138623157e+308		8
DECIMAL(n,d) NUMERIC(n,d)	decimal value with n digits and d decimals	depends on n and d. Since 5.0.6, 1 <= n <= 65, 1 <= d <= 30.		n

The numeric type that isn't there

Some other databases offer a *money* data type. MySQL does not. Because storage of FLOAT values is seldom exact (*Chapter 9*, Floats and rounding), usually the best choice for money columns will be DECIMAL with two decimal points. For example DECIMAL(10, 2) gives you eight places to the left of the decimal point and two to the right, with no rounding errors.

If performance is critical, you might instead prefer to store money values as integers, without decimals, using the smallest unit available in the given currency (cents versus dollars or euros, for example). Since MySQL performs operations on integers much more quickly than on strings, this approach will deliver the best possible performance. On the other hand, you will have to perform the conversion to larger units yourself (for example to convert cents to dollars, divide by 100; few people present project budgets in pennies).

Using AUTO_INCREMENT

The AUTO_INCREMENT column type modifier turns out to be especially useful for Primary Key (PK) columns. Recall that the purpose of a primary key is to uniquely identify a table's rows. Put another way, any additional attempt to add meaning to a primary key violates the rule of atomic values (*Chapter 1*, Codd's 12 rules). From the human viewpoint, the primary key is useless; many database applications never expose PKs in the user interface. The users of your application should never need to know any particular PK value. Even if you display them, in what way will it help your user to know that Johann Bach's customerID is 2138?

You might think that in some cases the PK can do double duty, for example, in the invoices table you need an invoice number, so why not use the PK?

Ask your accountants. Whenever you insert a row into a table with a surrogate (*Appendix A*) `auto_increment` column, the engine sends you the next number in the sequence. But what if you soon delete the row? On the next insert the PK value will increment again, thereby creating a hole in the sequence. Many accountants and managers won't tolerate missing numbers in a sequence. Resist the temptation to make the PK do double duty. The primary key has precisely one purpose, which has nothing to do with its actual value. What better way to create such values, then, than to have the system manufacture them? It's the system's job to guarantee uniqueness.

Every table must have a primary key. The advantages of `auto_increment` are enormous. The only issue is size: should the `auto_increment` be a smallint, mediumint or bigint?

Even if you agree that all primary keys should be numeric, you may disagree that they should be numbered automatically, for various reasons.

You can create a table whose purpose is to store the next valid primary key for a given table in your database, on any key generation rule you please so long as the results are unique. Suppose you have two tables, `customers` and `customers_pk`, each having a column `CustomerID`. The `customers_pk` table will always have only one record. Imagine that the current value of `CustomerID` is 492.

Client `C1` now wants to add a new customer. She fills out a form in your application, then clicks the Save button. MySQL grabs the value 492 from `customers_pk` and immediately replaces it with 493. It now inserts the `customer` record, using the value 492 as its primary key. An instant later client `C2` clicks her Save button. MySQL grabs the new value 493 and MySQL advances the value for next use to 494. It is impossible for two users to call on `customer_pk` at exactly the same time: one must arrive first, or MySQL will arbitrarily break the tie.

MySQL date and time columns

MySQL has five datetime types: DATE, DATETIME, TIME, TIMESTAMP and YEAR.

DATETIME

A DATETIME column stores date and time independently of `time_zone` settings. Since 5.6.4, you may specify microsecond digits as DATETIME(n) with 0<=n<=6. The default display format is `yyyy-mm-dd hh:mm:dd[.fraction]`. The range is `1000-01-01 00:00:00` through `9999-12-31 23:59:59`. To prevent errors from dates outside that range, set `allow zero datetime=TRUE` in the connection string. As of 5.6.5, DATETIME[(n)] columns defined with DEFAULT CURRENT_TIMESTAMP[(n)] auto-initialise to the current date and time, and auto-update with ON UPDATE CURRENT_TIMESTAMP[(n)].

DATE

A DATE column stores date values independently of `time_zone` settings. The default is NULL unless the column is defined as NOT NULL; then the default is 0. Default display is `yyyy-mm-dd [.fraction]`; the range is `1000-01-01` through `9999-12-31`. Before Enterprise edition 5.0.42 and Community edition 5.0.45, DATE-DATETIME comparisons ignored time; now they coerce DATE values to time=`00:00:00` unless cast with `CAST(value AS DATE)`.

TIME

A TIME column stores time of day independently of `time_zone` settings. Since 5.6.4, you may specify microsecond digits as TIME(n) with 0<=n<=6. The actual range, `-838:59:59` through `838:59:59`, permits storage of *elapsed time* values. Default display is `hhh:mm:ss[.fraction]`.

TIMESTAMP

Timestamps display as DATETIME values, range from 1970 through 2037. They convert to UTC based on the `time_zone` setting, converting back on retrieval. Before 5.6.4 tthey were 4-byte integers. Before 5.5.3 TIMESTAMP(*n*) specified display width but since 5.6.4 TIMESTAMP(*n*) specifies 0<=*n*<=6 microsecond digits. Before 8.0 created with `sql_mode =MAXDB`), they behave like DATETIME columns. Otherwise …

Before 5.6.5: They could be NULLed only if created as TIMESTAMP NULL and were otherwise autodeclared NOT NULL. They took zero or valid DATETIME values. *One* TIMESTAMP column per table could be defaulted on INSERT or UPDATE. If there was no explicit default, the first TIMESTAMP was autoassigned DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP, and subsequent TIMESTAMPs were autoassigned DEFAULT '0000-00-00 00:00:00'.

Since 5.6.5: MySQL accepts any number of TIMESTAMP columns with any combination of DEFAULT CURRENT_TIMESTAMP and ON UPDATE CURRENT_TIMESTAMP. DEFAULT <`constant`> and ON UPDATE <`constant`>.

Since 5.6.6: If the system variable `explicit_defaults_for_timestamp` is set ON, TIMESTAMPs not explicitly declared NOT NULL accept NULLs, no TIMESTAMP column is autoassigned DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP, and TIMESTAMPs declared NOT NULL without a DEFAULT clause have no default value (if `sql_mode` is `strict`, an error occurs).

Since 8.0.22, `CAST(timestamp_val AT TIME ZONE {[INTERVAL] '+00:00' | 'UTC' } AS DATETIME[(0…6)])`.returns `timestamp_val` as a DATETIME value.

CURRENT_ TIMESTAMP, CURRENT_ TIMESTAMP() and NOW() are synonyms.

YEAR

This is a one-byte column with a range from 1901 through 2155.MySQL accepts a display argument YEAR(n) but YEAR(2) was deprecated in 5.5.27, removed in 5.7.5, and YEAR(4) was deprecated in 8.0.19.

Datetime formats and conversions

MySQL stores dates in `yyyy/mm/dd` format but can decipher a wide variety of datetime formats. Since 5.6.4 it accepts fractional microseconds in DATETIME, TIMESTAMP and TIME values. Sometimes MySQL will even forgive the absence of quotes. Flexibility is terrific, but the rule of thumb is: provide enough formatting to make every part of every date and time value completely unambiguous. For a detailed list of all the possible datetime formats, see DATE_ FORMAT() in *Chapter 8*. Be aware that DATETIME and TIMESTAMP values may display identically yet reflect different underlying values, so conversions from one datetime type to another may lose some of the stored information.

MySQL OpenGIS column types

MySQL has supported a subset of the *OpenGIS* specification (*www.opengis.org/*) for SQL representation and storage of geometric data since version 4.1. The OpenGIS specification rests on a simple object model (Table 4-3) with five abstract and nine instantiable classes:

Table 4-3: OpenGIS Object Model

Class	*Dim*	*Instantiable?*	*Description*	*WKB Type*
Geometry		No		
Point	0	Yes	X and Y coordinates	1
Curve	1	No		
LineString	1	Yes	Line with interpolation between points	2
Line	1	Yes	Linestring with two coordinate pairs	
LinearRing	1	Yes	Simple, closed curve	
Surface	2	No	Has one exterior boundary, zero or more internal boundaries. If simple, the boundary is a set of LinearRings	

Class	*Dim*	*Instantiable?*	*Description*	*WKB Type*
Polygon	2	Yes	Zero or more internal boundaries, one external, all LinearRings defining holes, no LinearRings cross. If it has holes, exterior is not connected.	3
GeometryCollection		Yes	Collection of like Geometry objects	7
MultiPoint	0	Yes	Collection of Points, simple if 0 identical	4
MultiCurve	1	No	Collection of Curves, simple if all are, closed if all elements are. If closed, boundary is empty, otherwise a point is in its boundary if in the boundaries of an odd number of elements.	
MultiLineString	1	Yes		5
MultiSurface	2	No	Two have no intersecting interiors and boundaries which intersect at a finite number of points at most.	
MultiPolygon	2	Yes	Regular closed set of Points. No. of interior connected components = no. of Polygon values. Each curve in the boundary is in the boundary of one Polygon element. No two Polygon interiors intersect, no two Polygon elements cross or touch at an infinite number of points. Has no cut lines, spikes, or punctures, and is not connected if it has > 1 Polygon with an unconnected interior.	6

Table 4-4: Geometry Object Attributes

Property	*Description*	*Points*	*Curves*	*Surfaces*
type	Name of instantiable class	point		
SRID	Spatial Reference Identifier, identifies the object's coordinate space			
coordinates	Zero or more pairs (X Y) of 8-byte coordinates	(X Y)	(X1 Y1, …, Xn Yn)	
interior	space occupied by the object			
boundary	interface between the object's interior and exterior	empty if no coordinates	Empty if closed, otherwise its endpoints	Set of closed curves. If simple, the set of closed curves which are its exterior and interior boundaries.
exterior	space not occupied by the object			

Property	*Description*	*Points*	*Curves*	*Surfaces*
MBR	Minimum Bounding Rectangle (Envelope) formed by minimum and maximum bounding coordinates `((MINX MINY,MAXX MINY,MAXX MAXY,MINX MAXY,MINX MINY))`			
simple	Boolean	No coordinates	If it does pass through any point on it more than once	
closed	Boolean	No value	When first coordinate pair = last cordinate pair	
empty	Boolean	Yes		
dimension	-1, 0, 1 or 2	0	1	2

OpenGIS column types

Four column types hold simple OpenGIS values: GEOMETRY, POINT, LINESTRING and POLYGON. Four hold collections: MULTIPOINT, MULTILINESTRING, MULTIPOLYGON, GEOMETRYCOLLECTION.

MySQL accepts two representations of Geometry values: *Well Known Text* (WKT) and *Well Known Binary* (WKB). A WKT representation begins with the data type name followed by parentheses containing comma-separated coordinate pairs, for example

```
LINESTRING(0 0, 5 10, 10 15, 20 20)
MULTIPOLYGON(((0 0, 1 0, 1 1, 0 1, 0 0)),((2 2, 3 4, 5 5, 7 7, 9 9)))
```

A WKB representation uses a BLOB consisting of hex representations of:

- Byte order: 1-byte integer, 0 = little-endian byte or Network Data Representation (NDR), 1= big-endian byte or External Data Representation (XDR)
- WKB type: 4-byte integer, Table 4-4, rightmost column,
- 8-byte X coordinate,
- 8-byte Y coordinate.

See *www.opengis.org* for how to represent more complex geometry values.

NULL

NULL indicates that data is missing. You can define any column except the primary key as nullable, thus permitting empty values in that column. To exclude NULLs, define the column NOT NULL. Remember that NULLs slow down processing and commit you to testing column values against IS NULL as well as the values of interest.

Column types and indexes

An index is a data structure that databases use for instant access to table data. Without indexes, every lookup or list would require tiresomely long sequential searches. An index is defined by a key, a search specification that you expect to use frequently. Actually, it's a sorted mini-lookup-table consisting of a sort key plus the primary key value of each

row, which MySQL will use to locate the actual data record (unless the index contains all the data the request has asked for–an optimisation that is worth designing for).

At a minimum, you index the primary key and every foreign key (FK) of a table (the InnoDB storage engine does the latter automatically). These columns are most always numerics; by choosing the correct size you can ensure that you will never run out of keys.

For a foreign key with a very small number of possible values, an index is less useful, and may even be slower than simply reading the table. For example, a table of payment methods might include a dozen or fewer rows. It may be just as efficient to store the payment method word itself.

You may index on single columns, on leading portions of string columns, on multiple columns, and since version 8.0 with MySQL functions (*Chapter 6*).. The one-column index, for example on `clients.companyname` to facilitate searches on campany name, may seem most intuitive, yet may in fact be least useful because MySQL can generally use just one index at a time for a given table in a given query context. Defining indexes is mostly an optimisation issue (*Chapter 9*).

A PK is by definition unique. When creating other indexes, you may specify the UNIQUE qualifier. If you do, MySQL prevents duplicate entries in the indexed column(s). This is much like a Primary Key, but may provide additional "business" logic for your application. For example, the official ISO list of countries identifies each with both a number and a unique two-letter code.

Indexes are valuable tools, they speed up searches, but each index exacts a performance cost—MySQL must perform an additional physical update for each defined index. A rule of thumb: index only those columns needed to optimise frequent joins and queries.

Referential integrity

If table B has a foreign key column referencing a key in table A, we must not allow deletion of a key value A referenced by a row in table B. This is the *orphaned row* problem. Referential Integrity (RI) means that when you specify the rules for foreign keys, the RDBMS does the logical work for you, without application code from you except of course graceful error handling. MyISAM tables do not offer RI. InnoDB tables do (*Chapter 7*, INNODB).

In some situations we need the opposite functionality. We might have `ProjectQuotes` and `ProjectQuoteDetails` tables storing contractors' job estimates. Estimates are fictions. Some day we may wish to delete unrealised quotes and their details. This capability is called *cascading deletion*. InnoDB implements it (see *Chapter 6*, **Foreign Keys**).

Further reading

JR Groff, PN Weinberg, AJ Oppel. "SQL: The Complete Reference, Third Edition